A General Model for Component-based Software

by

Baoming Song

A Thesis Submitted to the
Faculty of Computer Science
In Partial Fulfillment of the Requirements
for the Degree of

Master of Computer Science

Approved:

Dr. P. Cox, Supervisor

Dr. T. Smedley, Faculty of Computer Science

Dr. R. Giles, Acadia University

Dalhousie University – DalTech

Halifax, Nova Scotia 2000

Table of Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
AWT	Java Abstract Windows Toolkit
CBVPE	Component-based Visual Programming Environment
COM	Component Object Model from Microsoft
COMEL	Component Object Model Exemplary Language
CORBA	Common Object Request Broker Architecture
CSCK	Component Software Construction Kit
CSP	Communicating Sequential Processes
DCOM	Distributed Component Object Model from Microsoft
EJB	Enterprise JavaBean from SUN
ECOOP	the European Conference on Object-Oriented Programming
HTML	HyperText Markup Language
GIF	Graphical Interchangeable Format
GUI	Graphical User Interface
GUID	Globally Unique Identifier
IDE	Integrated Development Environment
IDL	Interface Definition Language
I/O	Input/output
JAR	Java Archive File
JFC	Java Foundation Class
MIDL	Microsoft’s Interface Description Language
OLE	Object Link and Embed
OMA	Object Management Architecture
OMG	Object Management Group
OOP	Object-oriented Programming
ORB	Object Request Broker
RAD	Rapid Application Development
RFP	Request For Proposal

RPC	Remote Procedure Call
SOM	IBM’s System Object Model
TCP/IP	Transmission Control Protocol/Internet Protocol
UML	Unified Modeling Language
VCE	Visual Composition Editor in VisualAge for Java
VPE	Visual Programming Environment
VPL	Visual Programming Language

Acknowledgments

I cannot thank enough my supervisor Dr. Phil Cox for his guidance and his long time spent on revising my thesis. Without his brilliant ideas and his help, this thesis would not be finished.

I would like to thank Dr. T. Smedley and Dr. R. Giles for their serving as members in my examining committee. I sincerely appreciate their time, interest, and suggestions.

The financial support from the Faculty of Computer Science, Dalhousie University is gratefully acknowledged.

Further, I would like to thank my wife Fei Tan for her incredible patience, her love, and her strong support during my study at the Faculty of Computer Science.

Abstract

Component technologies have become the buzzword in today's software engineering communities. Component technologies empower software engineers to produce a higher quality, more reliable, and more maintainable software solutions in a shorter time and within a limited budget.

This thesis presents a general model for component-based software. The model precisely specifies component-based software with sound basis mathematics. It captures the essence of currently most popular component technologies like JavaBean, COM, and CORBA. It will help people understand concepts of component technologies more easily and also it could be used as a standing point to develop a formal testing and verification methodology for component technologies.

To verify the applicability of the general model, a prototype for the general model has been presented in this thesis. The prototype is implemented as a visual programming integrated development environment that takes full advantages of component-based technology and visual programming concepts. The prototype has proved that the general model for component-based software is applicable.

CHAPTER ONE

Introduction

1.1 Issues of Software Reuse

Software reuse has long been one of the major issues in the world of software engineering. The reason is obvious. Software reuse can dramatically increase the productivity of the software community, ease maintenance, and improve product reliability. Although most people would agree upon the importance of reuse, it is only today that it has become a main goal in software engineering. As a result, many software reuse technologies have been developed over the past few years (Jacobson, 1997; Leach, 1997).

Software reuse was first realized in the late 50's or early 60's when the concept of libraries was developed which allowed collections of pre-compiled, reusable subroutines to be linked into a program for performing specialized tasks. The area in which libraries have succeeded best is probably numerical analysis, where a large number of FORTRAN libraries are available on various platforms and are used in many engineering projects. But there are not many successful stories in other areas. The difficulty of encapsulating high-level functionality in subroutines is responsible for the failure of library-based reuse.

The inception of object-oriented programming (OOP) languages has made software reuse more feasible and practical. Its features include abstraction, polymorphism, delegation, dynamic binding, encapsulation, and inheritance (Budd, 1997). At its first appearance, people realized software reuse by inheritance. Now software reuse with OOP has shifted to object composition and genericity. In OOP, object composition is dynamically defined at runtime and achieves software reuse by defining new classes of objects as structures containing instances of other previously defined classes. This makes the object-oriented design more reusable. Genericity provides parametric polymorphism, allowing the same code to be used with respect to different types. For example, it is realized by using templates in C++. A type of element in a linked list can use templates to allow the programmer to declare a list of integer by using integer as a parameter, to declare a list of string by using string as a parameter, and so on. More importantly, object-oriented application frameworks have taken advantage of the features of OOP, and have evolved into one of the most powerful reuse techniques. In addition to reusing code, frameworks reuse the design (Lewis et al., 1995).

Another popular reuse technique in the object-oriented programming community is design patterns. Design patterns represent a recurring solution to a software development problem within a particular context. They have frequently been used to guide the creation of abstractions in the software design phase, necessary to accommodate future changes and yet maintain architectural integrity. These abstractions help us de-couple the major components of the system so that each component may vary independently (Gamma et al., 1994).

The current trend in software engineering is towards component-based development. Building software with components promises more efficient and effective software reuse and higher productivity. A system can be designed and implemented by assembling components, customizing or extending them as needed; and publishing components in a form that can be applied to design and construct others, based purely on interface specifications (Chappell, 1997; Szyperski, 1998).

1.2 Why Move to Component-based Software Engineering?

To increase software reuse is not the sole driving factor that supports the increasing interest in component-based software development. Other key driving factors include (Tran et al., 1997):

- To survive in the competitive software market, software companies have to deliver higher quality and more reliable software in a shorter time and within a limited budget.
- Today software markets have increasingly been expecting large-scale, more complex software projects. Traditional software development technologies and methodologies are inadequate for managing such projects which normally involve several software companies.
- Users expect software to be easy to maintain in order to decrease maintenance and operating expenses.
- Users require that software from different vendors will work together. Such integration requires strict adherence to standards, creating extra difficulties for software developers which can be addressed by a component-based approach to development.
- Techniques and approaches have been developed to make component-based software development more applicable and feasible.

As a consequence, component-based software development offers the following advantages over conventional software development (Chappell, 1997; Tran et al., 1997):

- Component-based software development can increase the productivity of software developers. Component-based software is constructed by assembling existing reusable components. This process is much faster than writing an application from scratch.

- Component-based software development offers higher quality, more reliable software. The main reason is that reusable components have been tested and therefore their quality can be assured.
- Component technology can ease software maintenance. Component-based software means that a large software application can be made of many small components. A task for maintaining a large software application can be partitioned to many smaller and easier tasks for maintaining components.
- Component technology makes it easier to manage software development. Component partitioning enables parallel development, allowing several organizations to be involved in development of larger and more complex software.
- Because component technology implies some base set of standards for infrastructure service, a large application can depend on these standards thereby saving considerable time and effort.

1.3 Objectives of This Research

As component-based development has become more and more important in the information technology world, many concepts and conventions have been introduced. These concepts and conventions are often easily misunderstood and misled. An obvious demand exists to precisely specify issues such as what is a component, how they are used, and how they interact with each other. Unfortunately these issues are addressed only for particular component models, for instance, the component object model (COM). There is a lack of a general definition for these concepts. As a basis for addressing this issue, our first objective is to provide an overview of component-based software technologies and formal approaches to component-based software. We will then propose a general model for component-based software consisting of formal definitions of what components are, how they are used, and how they interact with each other.

Another challenge in component-based software development is how to assemble components effectively and efficiently. As a component is ready, it must be deployed into a component models or frameworks. These component models or frameworks provide a systematic method of connecting components, where components are assembled by instantiating and connecting component instances and by customizing component resources. Component assembly can be done in several ways. One choice is to use a traditional textual programming language; however, this is likely to be complicated since the task involves describing networks. One way of simplifying the assembly process is by using visual programming or visual assembly tools. Once a component model or framework has been set up, visual programming environments or visual assembly tools could be devised that allow components to be plugged together graphically. It has been noted that visual programming has played an important role in component-based software development (Carrel-Billiard and Akerley, 1998). There are already visual component-based tools available on the software market. Most of these tools, however, are targeted to particular component models. For example, VisualAge for Java from IBM uses JavaBeans model. It is not easy for people to learn and understand the concepts of the general component model using these tools. A tool or development environment is definitely needed to clearly and visually present the general component model and concepts of component-based software. Therefore, our second objective of this research is to investigate the importance of visual programming in component-based software development, and to present a prototype of our general model implemented using visual programming concepts. This prototype will offer a visual integrated component development environment that enables people to learn and understand the general model and concepts of component-based software.

The rest of this thesis is organized as follows. Chapter 2 provides an overview of component-based software systems and technologies addressing issues such as the definition and characteristics of components. Several commercial component models such as COM, JavaBean, CORBA (common object request broker architecture) are introduced and compared. We follow with discussing the relationship between component-based software and visual programming in Chapter 3. An overview of formal methods for component-based software systems is given in Chapter 4. In Chapter 5, we propose a general model for component-based software. This model is discussed with the reference to an example, and compared with other component models. Chapter 6 presents a prototype of component-based visual programming environment based on our model. The evaluation of the prototype is discussed in Chapter 7. We conclude in Chapter 8 with a summary of this research and recommendations for the future work.

CHAPTER TWO

Overview of Component-Based Software

The concept of component has been around in the computer hardware industry for a long time. To build a computer, hardware engineers no longer design tiny, basic elements from scratch. They simply plug off-the-shelf components such as chips, boards, or cards together. Component-based development has brought a number of benefits to hardware engineers such as reusability, maintainability, flexibility, and integration readiness. Due to the constraint of time and budget, software engineers have sought similar techniques for software development leading in recent years to various techniques for building software from components. Component models like COM and CORBA (Common Object Request Broker Architecture) allow software engineers to plug together components in different languages and platforms. End-users are also benefiting from these technologies: for example, spreadsheet, word processing, drawing and database applications often use a component model to embed editable data from one application into the files created and managed by another (D'Souza et al., 1997; Jacobson et al., 1997; Szyperski, 1998).

2.1 What Is a Component?

The word "component" is used very broadly and often loosely throughout the software industries. Generically, a *component* is defined as *a computational unit* (Shaw and Garlan, 1996). Components can be things like clients and servers, databases, filters, and layers in a hierarchical system. Shaw and Garlan (1996) have classified components according to their structural properties as in Table 2.1.

The interactions among components are also of identifiable kinds. Some of the most common interactions are summarized in Table 2.2. Components as a computational unit and interactions among these components have formed the foundation of *software architecture*, an emerging discipline in software engineering (Shaw and Garlan, 1996).

Component Types	**Characteristics**	**Examples**
Pure computation	Simple input/output relations, no retained state	Math functions, filters, transforms
Memory	shared collection of persistent structured data	database, file system, symbol table, hypertext
Manager	state and closely related operations	abstract data type, many servers
Controller	governs time sequences of other's events	scheduler, synchronizer
Link	Transmits information between entities	communication link, user interface (GUI)

Table 2.1 Component Classification According to Shaw and Garlan (1996)

Although Shaw and Garlan (1996) define components in a very generic way, their definition is not widely accepted in the component software community. This is partly because their definition is more abstract and at a higher level. Intuitively, most people think of software components as analogous to hardware components, picturing software engineers assembling them into software applications much the same way as hardware engineers assemble a computer out of hardware components. Various researchers have attempted to define software components to fit this view, resulting in several different descriptive but informal models.

Interaction Types	Characteristics	Examples
Procedure call	Single thread of control passes among definitions.	ordinary (single name space), remote (separate name spaces)
Dataflow	Independent processes interact through streams of data; availability of data yields control	Unix pipes
Implicit event propagation	Computation is invoked by the occurrence of an event; no explicit interaction among processes	event systems, automatic garbage collection
Message passing	Independent process interact by explicit, discrete handoff of data; may be synchronous or asynchronous	TCP/IP
Shared data	Components operate concurrently (probably with provisions for atomicity) on the same data space	blackboard systems, multiuser database
Instantiation	Instantiator uses capabilities of instantiated definition by providing space for state required by instance	use of abstract data types
Knowledge passing	Rather than receive and send data and references to components, actually transmit an object, complete with the code that defines its behaviors	Agent and Mobile Agent

Table 2.2 Interactions among Components According to Shaw and Garlan (1996)

One of the most popular definition of a component was offered by a working group at ECOOP (the European Conference on Object-Oriented Programming) (Szyperski, 1998):

> "*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*"

This definition emphasizes component composition. As a unit of composition, each component has its specified interface that determines how it can be composed with other components.

Sterling (1998) extended the above definition then distinguished three aspects of a component:

- *A specification that describes what the component does and how it should be used.*
- *A packaging perspective in which a component is considered as a unit of delivery.*
- *An integrity perspective in which a component is considered as an encapsulation boundary.*

They then defined a component simply as:

> "*A software package which offers services through interfaces*".

A similar definition is given by D'Souza et al. (1997):

> "*A component is a coherent package of software that can be independently developed and delivered as a unit, and that offers interfaces by which it can be connected, unchanged, with other components to compose a larger system.* "

Like Sterling's definition, the above definition also reiterates that a component is a software package. D'Souza et al. (1997) have examined a component from a coding point of view, pointing out that a component package should include:

- *A list of Imports (import interface) that refers to other components on which this one depends.*
- *External Specification (export interface) that is a description of what it provides for users, and what they need to provide to make it work. In some components, part of the specification may be available from the executing components themselves such as JavaBeans components in which a reflection mechanism is available for this purpose.*
- *Executable Code that exists in binary format. The code can be coupled to the code of other components if it built according to a suitable and consistent component model.*
- *Validation Code that is used to help decide whether a proposed connection with other components is acceptable.*
- *Design that includes all the documents and source code associated with the work of satisfying the specification. It may not be available for users in some cases.*

Booch (1997) has provided the following rather informal definition of a component:

> "*A component is a non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function … [It] can be used in assembling a well-defined architecture [or application] … A component conforms to and provides the physical realization of a set of interfaces that specify some logical abstraction (i.e., system behavior).*"

This definition lists most characteristics of a component like nearly independent, reusable, and a set of interfaces.

A definition of component from http://webopedia.internet.com/TERM/c/component.html is that a component is

> "*a small binary object or program that performs a specific function and is*

> *designed in such a way to easily operate with other components and applications.*"

Different from other definition, this definition states that a component is a binary object or program. This implies that a component should be available in its executable code rather than its source code. This definition is consistent with COM component definition from Microsoft which we will discuss in Section 2.2.2.

Although these definitions differ in detail, their proposers would probably agree that a component is an independent software package that provides functionality. Moreover, they all emphasize the importance of well-defined interfaces. The interface could be an export interface through which a component provides functionality to other components or an import interface through which a component gains services from other components. All these definitions also emphasize the "black box" nature of a component: that is, a software engineer can use one to create a larger system without any knowledge of how it is implemented.

2.2 CORBA, COM, and JavaBeans

Given the definition of a component as described in section 2.1, there are two important questions to be answered: how is a component developed and how is the component applied in software development? A component model will address both questions. A component model provides a standard way to develop and use components and is expressed by a set of conventions. These conventions include (Anderson, 1998):

> *"the standard structure of a component's interfaces, the mechanism by which a component interacts with other components, patterns for asking a component about its features, and a means for browsing active components"*

The three main component models currently in commercial use are CORBA from the Object Management Group (OMG), COM from Microsoft, and JavaBeans from Sun Microsystems. In this section we will investigate these three component models and summarize their differences and similarities.

2.2.1 CORBA and CORBA components

Common Object Request Broker Architecture (CORBA), was proposed by the Object Management Group (OMG) to support open distributed communication between objects across a wide variety of platforms, languages, and implementations. Although "Object" is used in CORBA's name for historical reasons, CORBA actually provides a model for components, as defined in section 2.1 above. Therefore, CORBA could more properly be considered a distributed component model or framework (Harmon, 1998; Szyperski, 1998).

The way in which OMG works on CORBA is to issue Request for Proposals (RFPs) on all aspects of component technology then ask for the specifications of each part of CORBA from all member companies of OMG to fit into a broadly common Object Management Architecture (OMA). Since the goal for CORBA was to build a standard way to allow open intercommunication between different platforms and programming languages, at the very beginning, OMG has carefully settled CORBA on a source code standard, rather than a binary one like COM. This makes it much easier for member companies and individual vendors of CORBA-compliant products to add value to the CORBA standard.

OMA is a conceptual model that defines a set of facilities at a high level of abstraction as shown in Figure 2.1. CORBA is the core part of the OMA, serving as a common communication bus for all components that sit in heterogeneous environments and supporting a set of facilities. These facilities include object services, application interfaces, domain interfaces and common facilities. The following is a brief description of these facilities (Schmidt, 1998; Szyperski, 1998).

- **Object Services**: these services provide a variety of largely infrastructure services. Two examples are the naming service which allows other software to find components based on names and the trading services which allows other software to find components based on their properties.
- **Common Facilities**: these facilities are targeted for end-user applications. They provide interfaces for applications like email, compound documents, and so on.
- **Domain Interfaces**: these interfaces are targeted for specific application domains.
- **Application Interfaces**: they are developed specifically for a given application.

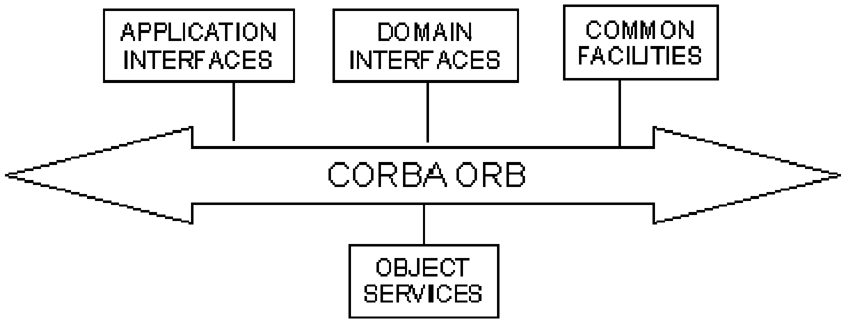


Figure 2.1 OMA Model

Given the high level view of OMA model, we discuss how the CORBA component defines and how these components interact with each other.

A CORBA component can be considered as an object with visible operations defined by an interface, and is invoked by an object reference, as shown in Figure 2.2. The object reference is also an abstraction (Yang and Duddy, 1996). An interface is the key part of CORBA component, determining the operations that other components may perform using the object reference. The component can only be accessed through operations defined for that interface. All the implementation details are hidden from interfaces (encapsulation). The interaction between components must carry out via interfaces.

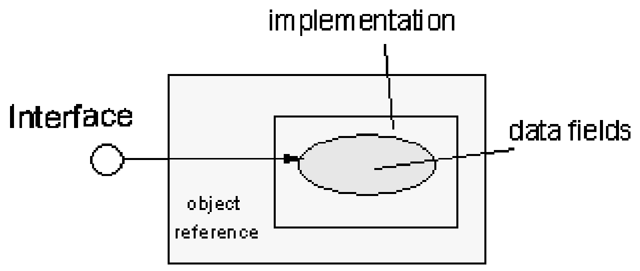


Figure 2.2 CORBA Component

All CORBA components must have interfaces defined in the CORBA IDL (Interface Definition Language). Different programming languages have standardized bindings to the IDL. Programmers either manually write IDL, then compile it into the source-code for the corresponding programming language in order to write their implementations; or use a programming language compiler that offers direct generation of IDL.

IDL is language and platform neutral, and declarative. It specifies a component's attributes (or public variables), the parent classes it inherits from, the exceptions it raises, typed events, programs for generating globally unique identifiers for the interfaces, and the methods an interface supports including the input and output parameters and their data types. It comprises the following main elements (Orfali et al., 1996; Yang and Duddy, 1996):

- **Modules** provide a namespace to group a set of class description together. A module has a scoped name that consists of one or more identifiers;
- **Interfaces** define a set of methods for a component that other components can invoke the component (see operation below). An interface can declare one or more

exceptions. It also may have attributes which can be changed by get and set operations;

- **Operation** is the CORBA-equivalent of a method. It represents a service that other components can invoke. The IDL defines the operation's signature, that is, the method's parameters and the result that it returns;
- **Data types** are used to describe the accepted values of CORBA parameters, attributes, exceptions, and return values.

CORBA component support inheritance through interface inheritance. There is no support on overriding or specialization of operations or methods as a class in OOP does. Interface also support aggregation mechanism (Jacobson et al., 1997).

Recently, CORBA defined mappings for the Java language, and aligned closely with JavaBeans and Enterprise JavaBeans for its component model. In fact, the Java Transaction Service is defined based on the CORBA model. OMG has also generated specifications for bindings between CORBA components and Microsoft COM components (www.omg.com).

CORBA is a quite powerful component model. The problem with CORBA, however, is that it requires significant overhead because of the encapsulation of objects written in different languages and the need for communication between those objects and other objects on other platforms.

Other CORBA-related component models have been developed. IBM’s System Object Model (SOM) is a component model created for OS/2. It is based on CORBA to which it adds some extensions, such as a binary standard like COM.

The OpenDoc component model was developed specifically to support GUI components and compound documents by Component Integration Labs (Apple, 1993) but has been discontinued by Apple (Szyperski, 1998). OpenDoc was built on top of IBM’s SOM model. Since it is based on CORBA, it has CORBA's characteristics, that is, source code standard, cross-platform and cross-language support.

OpenDoc is completely object-oriented. The model is therefore implemented by component classes, called parts. Compound documents with parts contain various media, such as text, graphics, table and so on. OpenDoc provides a uniform way to manipulate these media through familiar "cut & paste" and "drag and drop" operations.
Because of its pure object orientation OpenDoc allows inheritance and aggregating other components (Jacobson et al., 1997).

OpenDoc has the same capability as OLE (object linking and embedding) from Microsoft. The main difference between them is that OpenDoc is an open component model, whereas OLE is a proprietary standard. Unlike OLE which must adopt the paradigm enforced by COM, OpenDoc adopts the familiar object-oriented paradigm so that developers can implement OpenDoc components without much difficulty.

2.2.2 COM

COM (component object model) is Microsoft's component model. Unlike CORBA, which is an effort of a group of companies, COM is solely developed by Microsoft and therefore it is a proprietary component model. As a result, COM is only applied on Microsoft's own platforms, Windows 95 and NT. It is a high level standard and serves as a foundation on which all component models on Windows 95 and NT are based, such as OLE, ActiveX control.

COM is a binary standard, which means it does not bind to any programming language; so it is language independent. Developers can use any programming languages (C++ and Visual Basic at most cases) to implement COM components then use these components to develop COM component-based software on Microsoft's platforms as long as these components meet COM specifications.

COM specifies a standard way to allow COM components to communicate with each other. There are two key elements for COM: COM interfaces and a system for registering and passing messages between COM interfaces. All interactions with a COM component are through COM interfaces in the system. On the binary level, an interface is represented as a pointer to a pointer (pointer held in the first field of the interface node) to a table of interface functions (Op1, Op2, …), as shown in Figure 2.3 (Szyperski, 1998; www.microsoft.com/com).

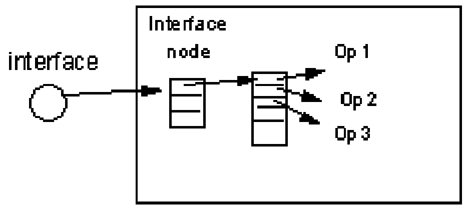


Figure 2.3 COM Component

From the implementation point of view, COM interfaces are defined in Microsoft's Interface Description Language (MIDL). Luckily developers do not need to write MIDL directly. Instead, the MIDL compiler will automatically link the source code written in C++ or Visual Basic to an MIDL interface on Microsoft platforms.

Whereas CORBA components use object reference, a COM component is identified by globally unique identifiers (GUID) which is a 128-bit integer. This guarantees that no two components can share the same identifier. Moreover, all COM components must support the most crucial interface, *IUnknown*. This interface provides a uniform way to allow a user to know what interfaces that the specific COM component actually supports. It also supports methods to manage component instances.

COM defines incoming interfaces as those interfaces that receive calls from other components and outgoing interfaces as those interfaces that through which other components are called. Just like JavaBeans use its event, COM uses outgoing interfaces to define events. Making requests and sending events by outgoing interfaces are similar to JavaBeans event processing as discussed in Section 2.2.3.

Unlike CORBA, COM does not support interface inheritance. However, it offers two mechanisms for object composition: containment and aggregation (Szyperski, 1998; Harmon, 1997).

Based on COM, several other component models have been developed for different purposes. OLE model is developed to primarily deal with compound documents. For example, OLE allows Microsoft Office users to "cut and paste" elements between different applications such as Excel, Word. In addition, it provides several services such as structured storage, uniform data transfer, and OLE automation and scripting (Jacobson et al., 1997).

Active X, another COM interface standard, was developed to support Internet and distributed computing, similar to Java Applets. Much like JavaBeans, ActiveX uses events to communicate with each other, and also have properties attribute.

COM+ enhanced COM by integrating with Microsoft's J++ approach (Microsoft's Java). It defines a virtual machine model for components, similar in many respects to the Java virtual machine (Kirtland, 1997).

DCOM is an extension to COM and supports Windows NT cross platform communications. In other words, DCOM is simply COM plus an ORB (object request broker). Microsoft's ORB is a straightforward implementation of a RPC (remote procedure call) that runs on the top of TCP/IP.

2.2.3 JavaBeans

According to the JavaBeans specification from Sun Microsystems (1997), a Java Bean is

> "*a reusable software component that can be manipulated visually in a builder tool*."

JavaBean components can be visual beans such as individual AWT components or Swing components for GUI design and also be non-visual beans such as database connectivity components (Flanagan, 1997).

JavaBeans is a Java -based component model. It depends on the features of Java language and is therefore language dependent. JavaBeans are portable because of the portability of Java. More importantly, the Java programming language provides an interface mechanism for JavaBeans. It supports interface inheritance, and interface registration. Unlike CORBA and COM, the Java interface is automatically (implicitly) registered, thus possibly saving developers' time (Harmon, 1998).

The main aspects of JavaBeans include:

- **Events:** Events provide a way for one component to notify other components that something interesting has happened. Instances of JavaBeans can be potential sources or listeners for specific types of events.
- **Properties**: Properties are conceptually nothing more than attributes or data fields of an object in object-oriented languages. It is used for both customization and programming. Changing properties usually trigger events or results in an immediate change in a JavaBean component.
- **Introspection**: JavaBeans allows programmers to discover the component interface in terms of the events it can signal, its property values that can be read and set, the methods it implements.

- **Customization**: Using the assembly tool, a bean instance can be customized by setting its properties.
- **Persistence**: JavaBeans can remember all aspects of their states between uses. For example, a JavaBean component can be implemented to maintain a database connection during a session.

Communication between JavaBeans components is handled by Java delegation event model. The event model specifies how a component sends a message to other components without knowing the exact methods that the other component implements. Figure 2.4 shows event processing in JavaBeans. A component interested on receiving events is an event listener or event sink. An event source maintains a list of listeners that is interested in being notified when events occur. The listener indicates its interest in an event by registering itself to the source or an adapter. When an event occurs on event source, the event source will notify all the listeners that the event has occurred. In order for an event source to invoke a method on a listener, all listeners must implement the required method. An adapter is used to simplify the implementation. Registering an adapter guarantees that a listener only implements those methods that the listener is interested in.

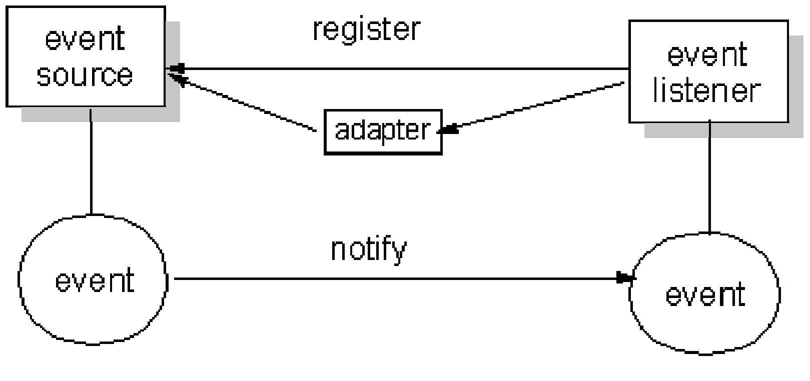


Figure 2.4 Event Processing in JavaBeans

Enterprise JavaBeans (EJB) is an extension of the JavaBean model. The key driving factor for introducing enterprise JavaBeans is twofold: first, there is a need to link JavaBeans with non-Java objects; and second, server components are needed to support a client/server computing architecture.

EJB retains the basic JavaBean component system, but adds to it the functionality of encapsulating a JavaBean component as a CORBA component. As such, all CORBA services can be available by using EJB in whatever CORBA environment is used (http://java.sun.com/products/ejb/docs.html).

2.2.4 Relationships and Comparisons

The relationships between these component models are shown in Figure 2.5. The models at the bottom of the diagram form the foundations for the models above them.

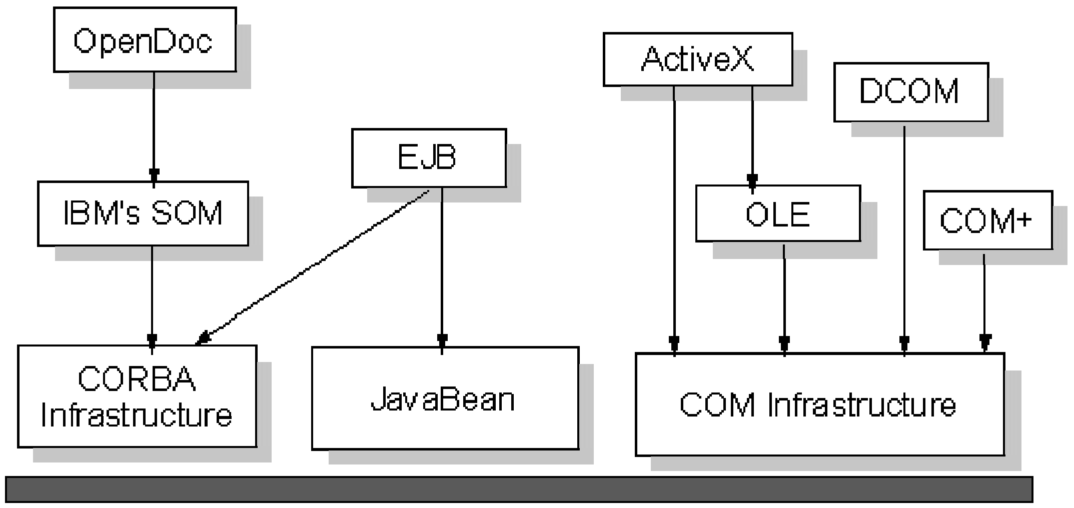


Figure 2.5 Component Models and Their Relationship

We now compare three component models, JavaBeans, COM and CORBA in various aspects.

The three leading component models, CORBA, COM, and JavaBeans, provide essentially similar architectures for component-based software. First of all, all three models emphasize the component interface through which services are provided. However, there exists a subtle difference with respect to how the interface is defined and how it is registered. CORBA and COM use IDL to define interfaces to components and component systems, the interfaces are explicitly defined and registered, whereas JavaBeans depend on an interface that the Java language provides which is implicitly registered. Secondly, the three models specify that the component is a software package or module that is a “black-box”. The only difference lies in how the internal workings are implemented. A CORBA component can contain any programming language implementation. A COM component is normally implemented by C++ or Visual Basic, although Microsoft has claimed it can be implemented by any language. JavaBeans are implemented in Java, and can consist of several classes. Thirdly, the three models address the connection between components in different ways. COM uses interface pointers; CORBA uses IDL; JavaBeans use Java delegation event model. Finally, the three models have different variability mechanisms that allow components to be specialized via their interfaces, such as inheritance and aggregation (Jacobson et al., 1997). Other differences are related to detail of platform support, distribution mechanism and self-description as summarized in Table 2.3.

	JavaBeans	**COM**	**CORBA**
Component	Module containing multiple classes	Module containing multiple classes or other implementation	Module containing any implementation
Interface	Java language	OLE IDL, which defines interfaces as collection of functions	OMG IDL
Connection	Via event and listener.	Via interface pointers; Need to support *IUnknown* interface	Via Interface Definition Language
Variability mechanism	Inheritance and aggregation	Genericity, containment and aggregation	Inheritance and aggregation
Platform	Multiple platforms	Windows 95/NT	Multiple platforms
Implementation Language	Java	Any languages, but primarily use C++ and Visual Basic	Any languages
Distribution Mechanism	EJB, Internet, RMI (remote method invocation)	DCOM, Internet	An ORB
Self-description	Support via introspection	No	No

Table 2.3 Comparison of Three Component Models

2.3 Characteristics of Component and Comparison with Object

After discussing the above three most popular component models, we now summarize the characteristics of a component. To be a good component, the following characteristics should also be offered (Anderson, 1998; Szyperski, 1997; Chappell, 1997). It should be pointed out that a component does not have to have all of these characteristics to be considered a component.

- **High Cohesion**: Cohesion describes the degree of sticking things together tightly. High cohesion for a component implies that a component is a meaningful unit.
- **Low Coupling:** Coupling describes the degree of relationship between components. Low coupling means that a component should be minimally depend on other components. If a component in a software system changes, the impact on other components in the system should be minimal, and vice versa.
- **Reusability**: After a component has been designed, implemented, and tested, it should be reused as much as possible.
- **Well-defined Services and Black-box Nature**: A good component should be a black box with well-defined services. A black box means that its internal workings are both hidden and isolated from its interfaces so that software engineers are able to build a software system using a component without any knowledge of how it is implemented.
- **Reliability**: A component should be tested individually. A reliable component is a prerequisite for developing high quality and reliable component-based software.
- **Distributability:** A component can be executed across a remote machine. CORBA, COM, and JavaBeans have provided this facility, although they use different mechanisms.
- **Interoperability**: A component can possibly request the component's services from any platform. CORBA and JavaBeans support multiple platforms, thus making components interoperable.
- **Cross-language Support:** An interaction between components should not depend on the programming languages in which a component is written.
- **Executability:** A component should be executed by anyone without having to make its source code available.
- **Self-description**: A component should be able to describe its public interfaces, any properties that can be customized, and the events that it generates. It is also possible to retrieve the information from the executing components themselves. For example, the JavaBean component model provides the introspection mechanism to achieve this.

The characteristics of components as summarized above makes it clear that components and objects have many similarities. Actually the concepts of component and object are often confused during component-based software development. To better understand component-based software systems, we summarize their similarities and clarify their differences as follows (D'Souza et al., 1997; Jacobson et al., 1997; Szyperski, 1998).

Logically speaking, components and objects are the same. They both are used to provide a concrete representation of real-world problems. Both support encapsulation (information hiding) and provide software reuse capability. On this level, a good component and a good object should share such characteristics as high cohesion, low coupling, well-defined services, and reusability.

Differences between components and objects lie in several aspects. The key difference is related to the different ways in which they are implemented. Because the concept of object is related only to object-oriented technology, the internal workings of an object has to be implemented by an object-oriented programming language such as Java, C++, Smalltalk, or Prograph. On the other hand, a component can be implemented by any technology or programming language as long as its interfaces comply with an accepted component model like COM or CORBA. A component's interfaces are separated from its internal implementation. For instance, a component can be implemented by procedural programming languages such as C, Fortran, object-oriented programming languages such as Java, C++, or Smalltalk, or even assembly languages. Moreover, since an object is an instance of a class, an object is realized only by a single class. A component, however, may be implemented as multiple objects of different classes by object-oriented programming languages.

	Component	**Object**
Programming Language	Can be implemented in any language	Only object-oriented programming languages
Interoperability	Support inter-services between different platforms	Usually does not have this capability
Intercommunication	Intercommunication can be message sending, event propagation, or interface pointers	Message sending
Persistence	Components have persistence	Usually does not have persistence
Frequency of Changing Implementation	More static	Dynamic
Dependency	Component depends on other component by import relation (Szyperski, 1998)	Class can depend on other class using inheritance or composition

Table 2.4 Comparison between Component and Object

It is possible for a component to request another component's services from different platforms, provided that a component is implemented with the same standard component model like CORBA and the component has standard interfaces for services. That is not normally true for objects.

In addition to using the message sending mechanism like the intercommunication between objects, the intercommunication between components could use event models like the Javabeans event delegation model or interface pointers as COM uses. A component must use its interface to interact with other components. It should be noted that an event model could be realized using the messaging sending mechanism.

As a component is delivered, its implementation is seldom changed. Therefore in terms of the frequency of changing implementation, a component may be more static than an object.

A component can normally remember their states between uses. However, an object usually does not have this capability.

Table 2.4 summarizes the major differences between components and objects.

2.4 Major Issues and Challenges

Component-based software has entered the mainstream of software engineering in computer software industry (Jacobson et al., 1997; Szyperski, 1998; Chappell, 1997). One of the most important challenges in component-based software development is to educate developers in the proper use of component technology and tools. This implies a need for a formal approach to precisely define components, to specify how they are used and how they interact with each other.

The second challenge in component-based software development is how to develop component-based software effectively and efficiently. Component-based software development is typical iterative and incremental (Tran et al., 1997; Barn, 1998). Each phase of development must be performed several times. This is different from the traditional software development model such as waterfall model – analyze everything, design every thing and then test everything. Moreover, component-based software development emphasizes the component assembly phase. To assemble components effectively and efficiently is of critical importance to component-based software development. An effective component assembling environment or tool is required for this end.

The third challenge for component-based software development is to make component-based software development an efficient and effective practice that does not suffer from the shortcomings of previous reuse-based efforts of the 1970's and 1980's (Barn, 1998).

Tools and methods are, therefore, required that support rigorous component modeling that separate component specification from component implementation.

2.5 Summary

In this chapter, we have presented an overview of component-based software technologies. We have discussed various definitions of components. The three most popular component models, COM, JavaBeans, and CORBA have been introduced and compared.

From the three component models, we have learned that the core concept for component development is the idea of creating an interface. Well-defined services for a component will be provided via interfaces. Interfaces also offer the way to allow components interact with each other. By registering the interfaces with a set of component system services, it becomes possible to assemble components together and deliver component-based software products quickly.

CHAPTER THREE

Component-based Software Development with Visual Programming

Component-based software development means building software by assembling or gluing components together. An Integrated Development Environment (IDE) is essential for component-based software development. In an IDE, components can be added to an application and connected together, and the application can be debugged and tested. Furthermore, the efficiency and usability of an IDE is a key factor influencing the value of component-based software development. A component-based visual programming environment as such is an IDE where the application assembly process is visualized and debugging and testing are visually interactive. Normally a graphical user interface (GUI) builder is included in a component-based visual programming environment.

PARTS from ObjectShare (formerly ParcPlace – Digitalk) and IBM's VisualAge were early examples of component-based visual programming environments (D'Souza et al., 1997). Note that the most popular visual environments, such as Visual Basic and Visual C++ from Microsoft, and Delphi from Borland, provide visual component assembly only for GUI components. We emphasize that a component-based visual programming environment is a visual building tool or environment in which general components, not just GUI components, can be assembled. This kind of environment has proliferated with the advent of the JavaBeans component model. A JavaBean component provides an icon defining its appearance in a component diagram; the mechanism of connecting one component to others; and documentation such as online help for the programmer (Sun Microsystems, 1997). During assembly, components are instantiated, and instances are connected using a uniform model –JavaBeans component model as discussed in Section

2.3.3.

There are a growing number of such environments or tools like IBM's VisualAge for Java, SunSoft's Java Studio, and others. These environments fall into two categories: either the programming process in the environment is fully visual as in Java Studio; or the environment provides only some visual programming capability as in VisualAge for Java. Environments of this kind take advantage of the benefits of both component-based software and visual programming, thereby significantly increasing software productivity. In this chapter we will briefly introduce visual programming, followed by investigating three visual component-based programming environments, VisualAge for Java from IBM (1997), PARTS for Java Technology from ObjectShare (http://www.objectshare.com), and Java Studio from Sun Microsystems (1997). We will conclude with some suggestions or comments.

3.1 Visual Programming

Visual programming is commonly defined as the use of visual expressions such as graphics, drawings, forms, animation or icons in the process of programming. It is a field that results from a marriage of work in computer graphics, programming languages, and human-computer interaction. Although the very first visual programming language was implemented in the 1960's, the success and boom of visual programming was not until the middle of the 1980's when graphics hardware became widely used.

3.1.1 Features of Visual Programming

It is a well-known quotation that "a picture is worth a thousand words". Pictures express more information than text does under some circumstances. The goals of visual programming are to improve the ways in which programmers express the representation and processing of information so that it is easy to understand and to modify logical connections and results by using visual expressions such as graphics and icons (Burnett et al., 1995).

The advent of the graphical user interface (GUI) has made it possible to move many programs from text-based to window-based. It allows users to perform operations with point-and-click actions with the aid of a mouse rather than having to memorize key combinations. The powerful graphics capabilities provided by modern operating systems provide the foundation for visual programming environments, which require the simple construction and manipulation of possibly complex diagrams.

Visual programming offers many advantages over traditional textual programming. Burnett et al. (1995) have identified the following four common characteristics that are particularly important to visual programming although textual programming may also offer some of them in different levels:

1. Fewer programming concepts: Visual programming aims to simplify the programming process by reducing some complex programming concepts like scope, variable declaration, pointers, and memory allocation, so that programmers without knowledge of these concepts can still create applications. Most visual programming languages have achieved this goal.
2. Concreteness: Visual programming manipulates concrete objects that are depicted as icons or other pictorial representations to create applications. Such representations are more concrete than textual programming. For example, a linked list can have a visual representation that shows its structure and data values.
3. Explicit depiction of relationships: Visual representation allows programmers to clearly see the relationship between program elements. For example, flow-chart-like pictorial syntax is often used in visual programming languages. This makes programs much easier to follow and to understand in contrast to traditional textual programming where programmers have to follow the program statement by statement or block by block, and especially if there is no documentation at all. The flow chart

also allows programmers to follow the logic of the program. If there is a logical mistake, it is easier to trace and make necessary changes.

4. Immediate visual feedback: Programmers can see the consequences of actions they perform in a visual programming environment. This obviously benefits both debugging and the general understanding of a program.

The major disadvantage of visual programming lies in its difficulty with handling large programs. This problem is called the *scaling-up problem*. Nine kinds of scaling-up problems have been identified by Burnett et al. (1995). One problem is how to effectively use the computer screen. Due to the complexity of large programs and the limitation of computer screen size, it is very hard to visualize everything together. The use of abstraction mechanisms in some visual languages (for instance, a block contains other blocks) has alleviated this problem.

3.1.2 Visual Programming Languages vs. Visual Programming Environments

Visual programming can be broken into two closely related areas, Visual Programming Environments (VPE) and Visual Programming Languages (VPL). VPEs are normally implemented in a VPE, but a VPE does not necessarily provide a VPL.

A VPE is defined as *a system in which the tools are graphical, using graphical techniques for manipulating pictorial elements and for displaying the structure of the program, whether it was originally expressed textually or visually* (Goldberg et al., 1995). The environment usually uses techniques such as point-and-click for action invocation or selection, and a “connect-the-dots” approach. “Connect-the-dots” means that modules are related to one another by drawing a line from one to the other. The lines specify particular relationships such as message sending, data flow etc.

A VPL is *defined as a programming language with a visual syntax* (Goldberg et al., 1995). Visual syntax means that at least some of the terminals of the language grammar are graphical, such as icons, forms or animations. The programmer writes a program by manipulating icons or other graphical representations in a visual environment. In the same visual environment, the program can subsequently be debugged and executed.

Visual programming languages may be further classified according to the type and extent of visual expression used, into icon-based languages, form-based languages and diagram languages. In purely visual programming languages, the program is compiled directly from its visual representation into machine code. And it is never translated into an interim text-based language before compiling into machine code. Prograph (Cox, Giles, and Pietrzykowski, 1989) is one of the most commercially successful purely visual programming languages. It has a number of features that are desirable for visual programming languages and environments. Moreover, some of its features shed light on designing and implementing a component-based visual programming environment. It is therefore worthwhile to discuss Prograph in more detail.

3.1.3 Prograph

Prograph is an object-oriented, data-flow, visual programming language for general purpose application development (Cox, Giles, and Pietrzykowski, 1989; Cox and Pietrzykowski, 1990). It has been available as a commercial product for over 10 years and has been used for creating a number of commercial software packages. Its original version was released on the Macintosh platform and it now exists in both the Macintosh and the Windows platforms. Prograph allows programmers to work on both high and low levels, allowing them to design and maintain from simple to rather more complicated software applications (Prograph International, 1993).

Prograph integrates the familiar notion of object-orientation with a powerful visual dataflow specification mechanism. It supports all standard Object-Oriented Programming features such as inheritance, polymorphism, and encapsulation. In Prograph, classes are organized in a single inheritance hierarchy. Each class contains the declarations for the attributes and the method of the class. All classes, methods, and attributes are explicitly public.

Prograph is dataflow in style. Dataflow means that the execution order of program is not fixed. Instructions are executed when all of their input data become available. Data flows into an operation which acts on them, producing results which flow out of the operation and on toward other operations along data links.

Prograph is purely visual. All the elements are concrete objects represented by icons. These icons provide an effective communication vehicle for programmers to design and create applications. Prograph's visual syntax allows programmers to create a program just by placing a set of built-in or user-defined primitives in a window then drawing lines to link these icons together.

In Prograph a project consists of sections that contain universal methods, persistent data objects, and classes. Classes consist of methods and attributes. Methods, in turn, are composed of cases. This characteristic encourages programmers to follow a top-down software development methodology.

Figure 3.1 shows the case window of a method. The inputs and outputs of a case are graphically represented as horizontal bars at the top and bottom of the case, respectively. Data may flow into the case through an input bar located in the top via the terminals, represented by small circles attached below the input bar. Data may flow out of a case through the output bar (bottom bar) via roots represented by small circles attached above the output bar.

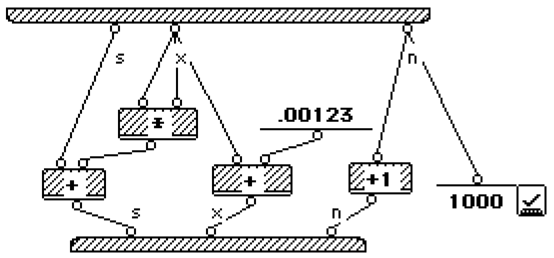


Figure 3.1 A Case in Prograph

Prograph CPX implements the Prograph language in an IDE that includes a program editor, code interpreter, compiler, graphical debugger and GUI builder. This interactive environment provides a powerful facility for developing simple programs to more complicated applications.

Note that the above presents just the basic features of Prograph and is not sufficient to illustrate all the features of the language and environment. We conclude this section with a list of the important aspects of Prograph and particularly those aspects that benefit component-based software.

- Prograph is a purely visual programming language. Despite its simple visual syntax, Prograph allows the creation of complex applications. Furthermore, Prograph avoids syntactic errors since the visual syntax is particularly well suited to syntax-directed editing.
- Prograph employs dataflow diagrams in cases. This makes the logic of the program much clear, thus allowing programmers to trace and debug the program more easily.
- Prograph is an object-oriented programming language. All the concepts of object-orientation are represented graphically. Since the concept of component-based software evolved from that of object-oriented software, the visualization concepts and design style of Prograph definitely lend themselves to the design and implementation of a component-based visual programming environment. In fact, Munch and Schurr

(1999) have recently classified Prograph as a component-based visual language by classifying operations as components, and the terminals and roots of operations as in-ports and out-ports respectively (Figure 3.1). Although this analogy does not rigorously comply with the concepts of component-based software as discussed in Chapter 2, it is very helpful for the design and implementation of a prototype of a component-based software development environment, which will be presented in Chapter 6.

- Prograph frees programmers from dealing with tedious and unnecessary levels of detail. For example, its visual nature eliminates the variable required by textual programming languages.
- Prograph provides an environment which encourages a top-down approach to software development. The design concept could be used in implementing visual programming environments for component-based software development.

3.2 Component-based Visual Programming Environment

Applying the concepts of visual programming to a component-based software development environment has a significant impact on software productivity. A component-based visual programming environment may completely visualize the whole assembly process. The "connect-the-dots" approach is normally applied in this environment where a component is represented as a particular icon a connection between components is visually expressed by drawing a line from one to another. As such, it is quite easy for programmers to follow the logic of the application and to understand the architecture of the application, thereby speeding up software development.

In this section we will introduce three component-based visual-programming environments: VisualAge for Java, Java Studio, and PARTS for Java. These environments are IDEs with visual editor, debugger and interpreter. They provide the ability to select a visual component from a palette and place it on the user interface. The programmer can modify the component's appearance and other attributes. For connecting components together, some use the drag-and-drop paradigm while others have the programmer click on the origin component for a connection and then click on the destination component.

3.2.1 VisualAge for Java

VisualAge for Java was developed by IBM in 1997. It is an integrated, visual programming environment that supports the complete cycle of Java program development. With VisualAge for Java, the programmer can build a Java platform compatible application, applets and JavaBeans components (IBM, 1997).

One of the most important parts of VisualAge for Java is the Visual Composition Editor (VCE), where programmers can develop programs by visually arranging and connecting JavaBean components. In the VCE, programmers select JavaBean components from a palette, specify their characteristics, and make connections between them. Beans can contain other beans and also connect to other beans. This kind of abstraction results in a cleaner picture of connections between components in VCE allowing programmers to design and debug programs more easily. VisualAge for Java not only provides visual tools for building the user interface of the applets or applications, but the whole applet or application itself, provided that the necessary Javabean components have been developed.

An important aspect for developing software applications in this environment is how to visually connect or "wire" JavaBean components together. Four main connections between beans have been identified and provided by VisualAge for Java (Carrel-Billiard and Akerley, 1998). These connections are represented by different colors.

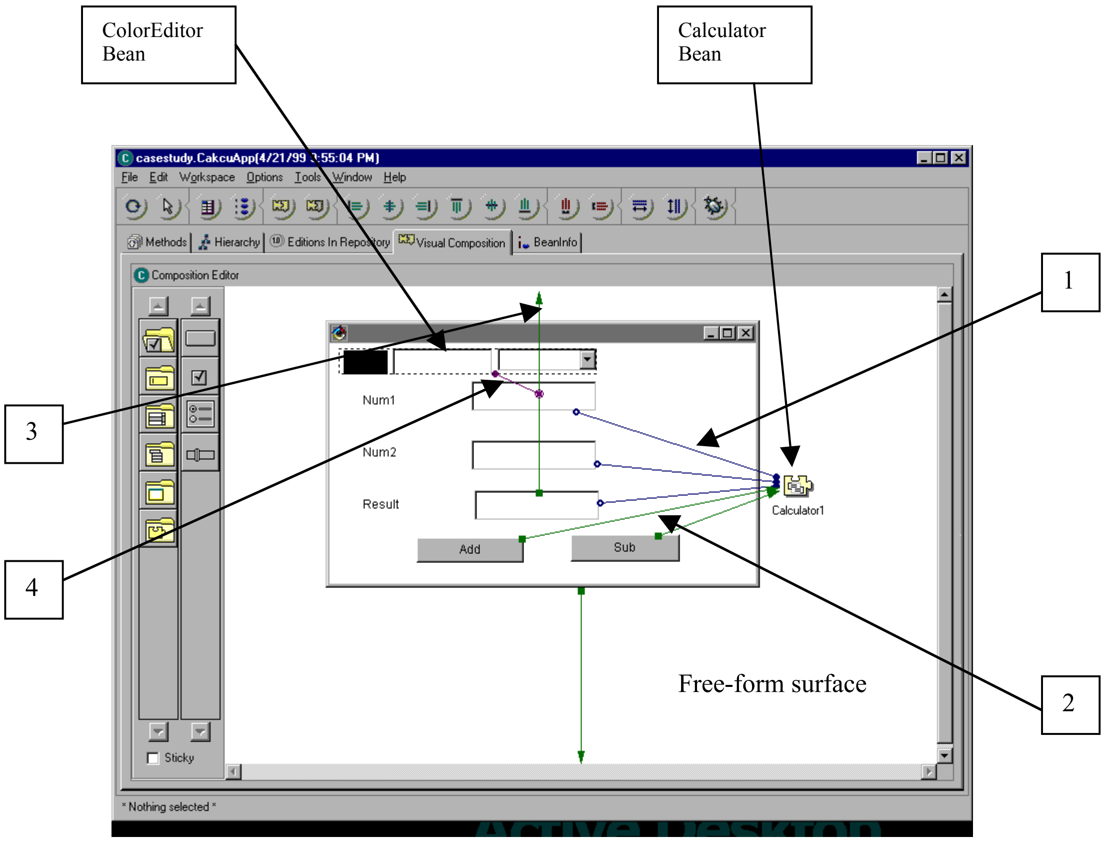


Figure 3.2 Visual Composition Editor in VisualAge for Java

We use an example to show how these connections work. The example is to construct a simple calculator. Figure 3.2 shows a snapshot of this simple calculator illustrating all Javabean components and the connections between components. In the calculator, two textfields are provided for entering numbers. The value of the result will display in a result textfield. When the *add* button is clicked, the result textfield displays the result of adding two numbers together and when *sub* button is clicked the result textfield displays the result of subtracting one number from another. Besides, the result textfield can display its value with different colors according to the user's selection. A user-defined JavaBean *Calculator* has been created by using textual programming whose functionality is to manipulate two numbers and obtain the result according to the request. A ColorEditor bean and other visual beans are used like frame and label. Four kinds of connections are identified as follows:

- **Property-to-property**: this connection links property values between two JavaBeans components together. When one component's property value changes, this connection will cause other component's property value to change. This connection never takes parameters. After the target property is initially set up, events are required to fire the connection. The connection # 1 in Figure 3.2 is a property-to-property connection which links a value of the textfield to a value in *Calculator* component. The change in the value in the textfield will cause the corresponding value in *Calculator* to change.
- **Event-to-method**: it links a source event to a target method. This connection invokes the target method whenever the source event occurs. Since the JavaBean component model is based on events, many of the behaviors of an application can be specified visually by triggering the execution of a method of one bean whenever an event is signaled by another bean. The connection #2 in Figure 3.2 is an event-to-method connection between a button and *Calculator* component. When users click the button, a method in *Calculator* for adding two numbers together will be executed.
- **Script connections or event-to-code**: this connection is used to access behavior that is not part of the bean interface. The script refers to local methods declared as private or protected or to the inherited methods declared as protected. This connection is graphically represented a line between the free-form surface and a Javabean that accesses the script. The connection # 3 is such a connection. In this case a local method *AdjustColor (Color aColor)* is created that allows the result textfield to access.
- **Parameter connections**: By passing either the value of a property of a JavaBean component or the return value from a method, this connection provides an input value to the target of a connection. There are three kinds of parameter connections: parameter-from-property, parameter-from-script, and parameter-from-method. The connection # 4 is a parameter connection. In this case, a parameter *Color* is provided from ColorEdior bean to the target of connection # 3: the local method *AdjustColor*

(Color aColor).

Note that in this environment, the programmer can access all methods, selectively show connections between components, and edit existing connections.

Although it is powerful, VisualAge for Java is still not very popular. The main reason is probably it has a complex user interface, and requires thorough knowledge of the JavaBean component model.

3.2.2 PARTS for Java Technology

PARTS for Java is a complete visual-programming environment that creates small applets to large scale distributed Java applications (http://www.objectshare.com). It fully supports the JavaBeans component model. Besides this, it also supports CORBA architecture and has a bridge for conversion between JavaBeans components and COM components. Active X controls and other COM components can be imported into Java programs, and used just like any of the other Beans. A JavaBeans component can also be published as an ActiveX that can be used in any ActiveX container in which ActiveX can be composed, such as Microsoft Visual Basic or a Web browser. This environment is one powerful visual component-based programming environment in terms of its support for several popular component models.

From aspects of visual programming, this environment is very similar to VisualAge for Java. It has the Visual Designer that is an application development tool that allows programmers to construct applications by arranging and connecting components on a workbench surface. These include visual components that are used to create user interfaces, or non-visual components that perform certain “behind-the-scene” operations. All these components are in the form of JavaBeans. But Active X controls or other COM components can be imported and automatically converted into JavaBeans components.

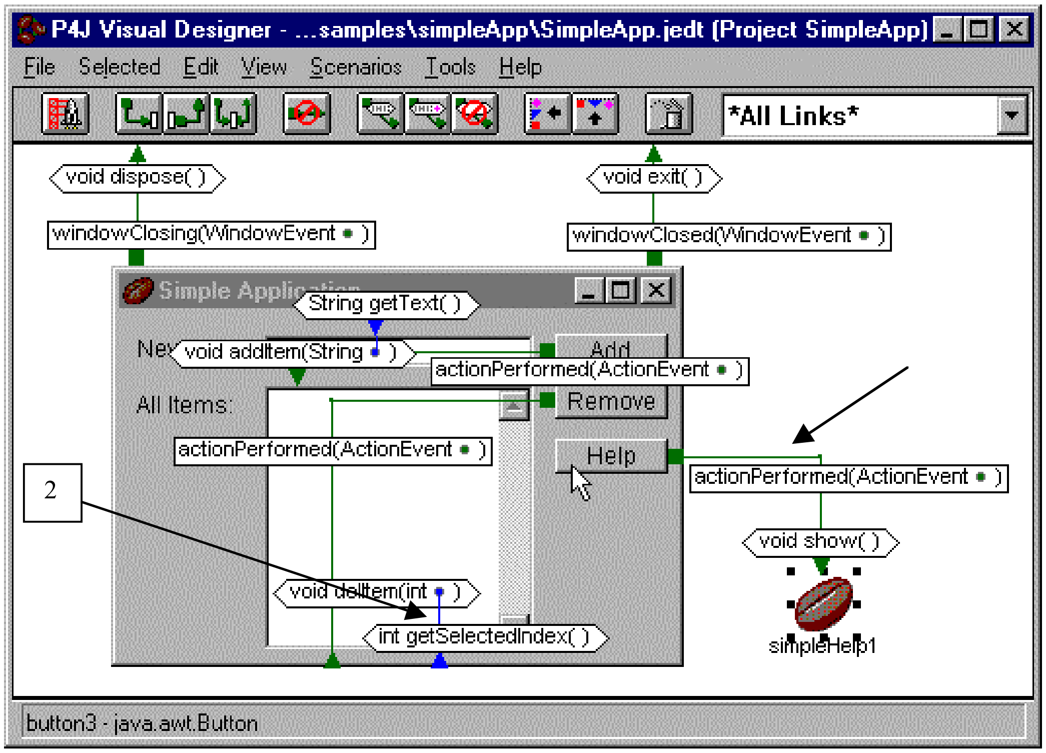

P4J Visual Designer - ...samples\simpleApp\SimpleApp.jedt (Project SimpleApp)
File Selected Edit View Scenarios Tools Help
All Links
void dispose( )
void exit( )
windowClosing(WindowEvent ●)
windowClosed(WindowEvent ●)
Simple Application
String getText( )
New
void addItem(String ●)
Add
actionPerformed(ActionEvent ●)
All Items:
Remove
actionPerformed(ActionEvent ●)
Help
actionPerformed(ActionEvent ●)
2
void show( )
void delItem(int ●)
int getSelectedIndex( )
simpleHelp1
button3 - java.awt.Button

Figure 3.3 Visual Designer in PARTS for Java

Applications are realized by assembling or "wiring" components together using links or connectors. A link represents a message sent from one component to another, usually as the result of an event, like a button click, since JavaBean uses event delegation model as discussed in Chapter 2. Figure 3.3 shows components with links.

A component in the Visual Designer has an exposed interface (events and messages) for connecting. An event is a signal that something has happened, such as the user clicking a button or pressing a key. A message is a request for an object to perform some action, such as responding to an event. Both event and message refer to a method in object-oriented languages. A link or connection provides a visual indication of the interactions between components. There are the following kinds of links in PARTS for Java:

- An event link connects a single event with a single message (equivalent to the event-to-method connection in VisualAge for Java). In Figure 3.3, link # 1 is such a connection. It links a help button to a help message. When the button is clicked, the help message will be shown.
- An argument link requests a value from another object to complete the link (equivalent to the parameter connection in VisualAge for Java). The link # 2 is this kind of connection.
- A result link cascades messages, passing the return values of one message along to subsequent message as an argument value (equivalent to parameter connection in VisualAge for Java)
- A diving link cascades messages by constructing a series of messages to the object returned by an event link

A nice feature is that a label for each event or message will be displayed as necessary. This secondary notation helps make coding and debugging the application much easier. (since this software is not available, I am not able to present the same example)

3.2.3 Java Studio

Java Studio is a visual programming environment developed by Sun Microsystems (1997). Although Java Studio has been discontinued, it is worthwhile describing it here because of its unique features. Java Studio was developed not only for end users who do not want to write textual code, but also for programmers who want to rapidly create prototypes, JavaBean components, or working applications. A set of robust JavaBeans components was shipped with Java Studio. These include GUI-building components, as well as components for string handling, simple numerical calculations, and control and data flow between components.

Java Studio provides two ways for viewing the design while creating it: the design window where components are connected or "wired" together to create the application, and the GUI window where the GUI page elements are laid out and can be tested as the application is being built. This unique feature provides “progressive evaluation” so that the programmer gets immediate feedback on design and programming decisions (Green and Petre, 1996). In the design window, each JavaBean component has an iconic representation with ports shown visually along the border of the component icon. Once a design is complete, it can be generated as a Java applet, a Java application, a JavaBean component, or packaged design that can be re-imported into this environment.

The Java Studio architecture is based on the JavaBeans specifications. A component is a so-called the *end-user bean* which wraps up a JavaBean (the skilled-user bean). Each component contains a series of ports through which the component can interact with other components, as shown in Figure 3.4. A port may either be:

- Input: it can only receive inputs
- Output: it can only send outputs
- Two-way: it is a bi-directional, input-output (I/O) port which can both send outputs and receive inputs
- Trigger: it activates an output connector and it is a specialized input connector;
- Hide/Show: it makes a component visible or invisible;
- Enable/Disable: it enables a component to perform its actions or prevents it from performing its actions.

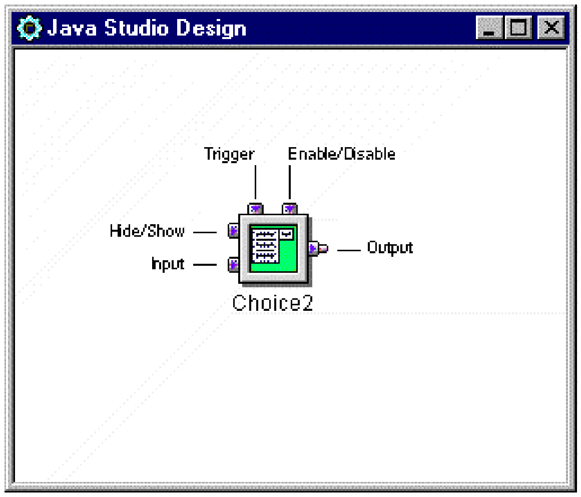


Figure 3.4 JavaBeans Component for Java Studio

Specifically, a port is used to map JavaBeans events, methods, and bound properties. Ports can receive or send messages. The message type could be numerical value, string, boolean value, action, or variable.

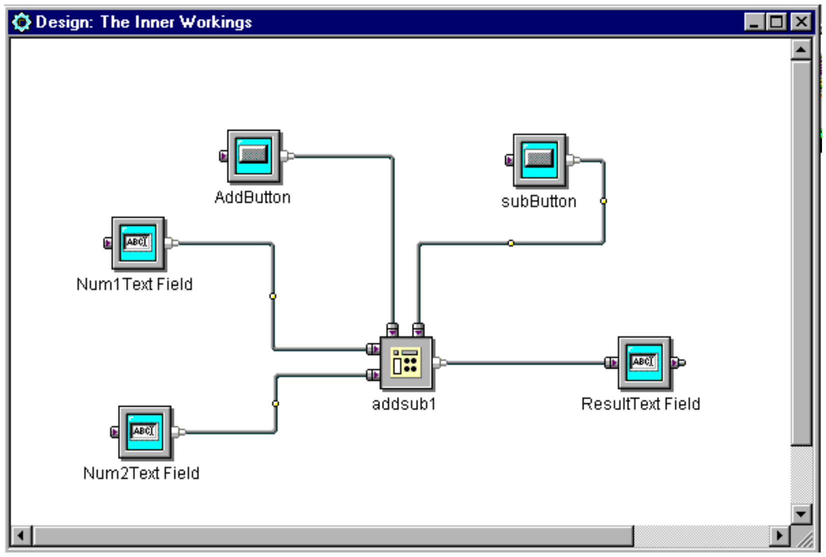


Figure 3.5 Design Window in Java Studio

Figure 3.5 shows a simple calculator in the design window. The calculator is the same as one discussed in Section 3.2.1 except that there is no selected color display for the value of the result textfield. A package named *addsub1* is implemented to contain the functionality of the calculator and it only exposes four ports: two inputs, two triggers, and one output. Two textfield components are used to input numbers and connect to the input ports of *addsub1* via their output ports, respectively. Two button components are used to trigger events and connect to trigger ports of *addsub1* via their output ports, respectively. A textfield component is used to display result and connects to output port of *addsub1* via its input port.

The visual implementation for *addsub1* is shown in Figure 3.6. All components employed are shipped with Java Studio. It should be noted that four external connector components are applied to implement ports of the package. The properties of these connectors depend on the ports to which they connect.

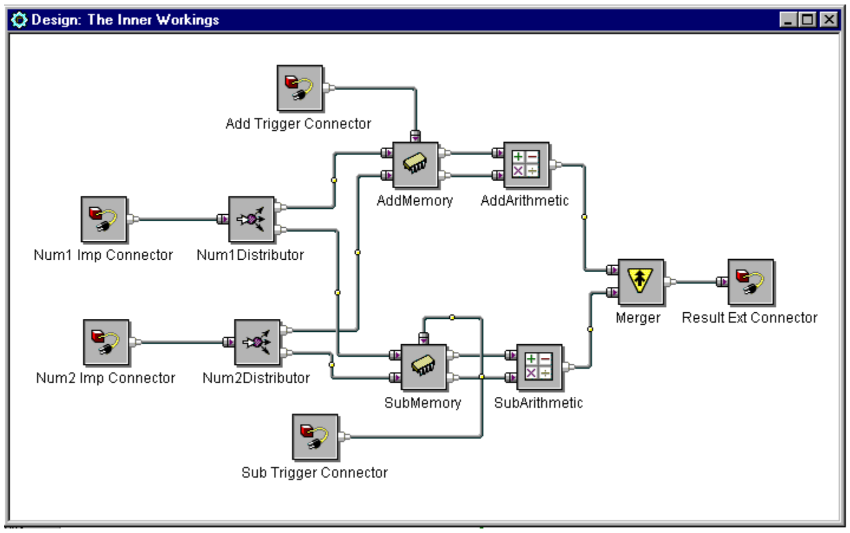


Figure 3.6 Package in Java Studio

As demonstrated above, Java Studio provides a fully visualized component-based visual programming environment where the task of programming simply involves visually connecting the components together. Networking between components is quite clear, which allows programmers to follow the logic without any difficulty. It is simple enough for non-programmers to write an application. However it may be not powerful enough for professionals to write large applications. Furthermore, it is impossible to write a recursive program using Java Studio.

3.3 Summary

After investigating the above visual programming environments, we conclude that:

- Applying the principles of visual programming to a component-based software development environment could significantly ease and speed up developing component-based software, thereby increasing software productivity.
- The features of Prograph have implications for developing a component-based visual programming environment.
- The most popular commercial component-based visual development environments are those that are based on the JavaBean component model. Furthermore, some of these environments have the built-in capability to convert between JavaBean components and other components including CORBA and COM components.
- The popular "connect-the-dots" approach has been applied in most visual component-based environments. The advantage of this approach is that it provides a clear picture of the logic of the program. It makes programs conceptually simple, and easy to debug. But if there are too many connections, this approach may make the window look like a "spider-net". However, this problem can be solved using abstraction. For example, if a component can contain other components, the connections will be reduced to the certain level.
- The success of component-based visual programming environments largely depends on whether or not they provide powerful functionality or are easy to learn. Two extreme examples are Java Studio and VisualAge for Java. Java Studio's "poor" functionality probably resulted in its recent withdrawal from commercial distribution.

VisualAge for Java is not popular partly because of its complexity.

- It should be pointed out that in the above visual programming development environments, recursion is not realized. The lack of this feature could be considered as a shortcoming from a programming point of view.

CHAPTER FOUR

Overview of Formal Models for Component-based Software Systems

By a “formal approach” we mean an approach with a sound basis in mathematics. A formal approach allows system functionality to be precisely specified. As the complexity of building software systems is continually growing, it becomes more important to use a formal approach to attain a reasonable level of dependability and trust in software systems. Formal approaches allow us to analyze programs, to precisely describe the behaviors of programs, and to verify program properties. A formal model for component-based software is of critical importance because it provides a basis for the understanding of the underlying concepts of component models, component certification techniques, component testing etc. In this chapter, we will introduce Petri Nets, a graphical-based formal approach. Following that we will discuss the formalism of component models. Two formal models for COM will be briefly presented.

4.1 Petri Nets

There are five types of formal approaches: model-based, logic-based, algebraic, process algebra and net-based (graphical) approaches (Liu et al., 1997). Two examples of such formal approaches which could be used for specifying component-based software are Z notation which is model-based and Petri Nets which are net-based.

The *model-based approach* formalizes a system by explicitly defining states and operations that change the system from one state to another. Model-based systems employ standard mathematical notations such as sets and relations. Since this approach normally has rich constructs, the specification and formalization of software can be fairly concise (Sommerville, 1992). Three of the most popular model-based approaches are Z, VDM, and the B-method (Sommerville, 1992; Liu et al., 1997). This kind of approach is attractive for modeling component-based software because of its expressive power. Particularly, Z has been used for modeling COM components (Sullivan et al., 1997), which we will introduce in Section 4.2.2.

The *Net-based approach* uses graphical notations with a formal semantics to model systems. Because of their natural pictorial representation, graphical notations are easier to comprehend and, hence, more accessible to non-specialists (Liu et al., 1997). Petri Nets and State Charts belong to this category. Since component-based software has a close relationship with visual programming, this approach could shed some light on visually modeling component-based software in terms of its graphical notations.

Petri Nets have been thoroughly researched since they were invented by Carl Adam Petri in the beginning of the 60’s. Originally defined to generalize automata, Petri Nets have found wide applicability in computer science, in such fields as performance evaluation, operating systems, and software engineering. In particular, Petri Nets have proved to be useful for describing the concept of concurrently occurring events (Schach, 1997).

Petri Nets can be considered as both a formal and a graphically appealing language. A Petri Net consists of four parts: a set of places P, a set of transitions T, an input function I, and an output function O. A place is graphically represented by a circle. A transition is graphically represented by a square. An input or output function is graphically represented by an arrow. Consider the Petri Net shown in Figure 4.1.

The set of places P is $\{p_1, p_2, p_3, p_4, p_5\}$. The set of transitions T is $\{t_1, t_2\}$. The input functions for the two transitions, represented by the arrows from places to transitions, are

$$I(t_1) = \{p_2, p_4\}$$

$I(t_2) = \{p_2, p_5\}$

The output functions for the two transitions, represented by the arrows from transitions to places, are

$O(t_1) = \{p_1\}$
$O(t_2) = \{ p_3, p_3\}$

Note that unlike a transition in an automaton which connects one state to another state, a transition in a Petri net connects a set of places to another set of places.

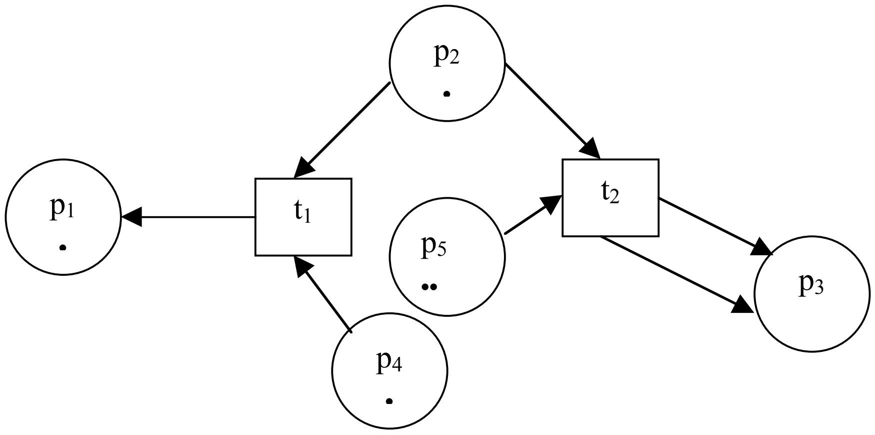


Figure 4.1 Petri Net

An important feature of Petri Net is that it can assign tokens to the places of the Petri Net, called a *marking* of the Petri Net. Places may contain zero or more tokens that are graphically represented by a black circle inside a place. The initial assignment of tokens is determined by the initial state of system. A set of markings can be represented by a vector. For example, the set of initial marking in Figure 4.1 can be represented as M = {1,1,0,1,2}.

The markings are used to express the state of the Petri Net at different times. They change during execution of the net as the tokens "travel" through it. The number and distribution of the tokens control the execution of the Petri Net. A transition is enabled if each of its input places contains at least as many tokens as there exists links from that place to the transition. When a transition is enabled it may fire. When a transition fires, all enabling tokens are removed from its input places, and a token is deposited in each of its output places.

It should be pointed out that the above describes basic Petri Nets. Higher level nets such as colored nets, timed Petri Nets, and stochastic nets have been introduced to extend the modeling power of Petri Nets to deal with complex systems.

The primary strength of Petri Nets lies in the way they can deal with concurrency, non-determinism and casual connections between events. Furthermore, since Petri Nets use graphical notations, they are much easier to comprehend.

Graphically a "place" in a Petri net is analogous to a component, and the way that tokens travel through nets is similar to the passing of messages between components in component-based software. In this sense there exists a possibility of using concepts of Petri Nets to visually model component-based software. However, a component is much more complex than a "place". More features must be added to "places" if we are to use them to model components. The significance of Petri Nets, however, is that we could use them as a basis for visualizing the concepts of components. This is reflected in the visual syntax of our general model for component-based software presented in the next chapter.

4.2 Formalization of Component Model

Some component models have been formalized because of their importance. CORBA has been formalized by Bryant and Evans (Sullivan et al., 1997) using Z notation in order to provide a consistent and unambiguous specification so that member companies of OMG can follow it when they add more facilities to CORBA. Unfortunately the details for the model and its consequence are not described in the available publication.

There appears to be no the formalizations of the JavaBean component model, although Drossopoulou and Eisenbach (1998) have formalized the subset of the Java language on which JavaBean is based. From available publications it seems that there have been only two attempts to formalize the COM component model. Because of the popularity and some subtle features of COM, it is not surprising that people have paid more attention to the formalization of COM than other component models. In this section we will briefly present these two formalisms.

4.2.1 COMEL Language

As has been discussed in Chapter 2, COM provides a theoretical foundation for Microsoft's OLE and Active X controls. It specifies a binary standard for object invocations and a number of interfaces for foundation services, independent of any specific programming language.

Like many software systems, COM was originally not formally defined, but is defined by its implementation. If one wishes to write specifications for COM-based software, or derive properties of such software, a formal definition is necessary. To this end Ibrahim and Szyperski (1998) have proposed a formal model for COM presented in the form of a language, providing an example of how the concepts inherent in COM might be formally captured. This language, called COMEL (Component Object Model Exemplary Language), provides a formal syntax and semantics embodying COM's informal and complex rules.

The importance of this language is that it formalizes COM- specific properties in addition to the familiar object-orientation concepts. This may serve as a starting point for generating a general specification language for component-based software. Furthermore, this language has provided a formal foundation for simplifying COM implementation by removing the commonly used interfaces of COM like *IUnknown*.

4.2.2 Another Formal Model for COM

Another attempt to formalize COM has been carried out by Sullivan, Socha, and Marcjukov (1997). Their motivation for this work was purely practical, arising from a commercial software development effort in which they were using COM in the implementation of a multimedia authoring system. They discovered during this development that without precise, formal definitions for COM concepts, they could not be confident that their system would not fall prey to faulty decisions made in the early stages of design. In order to avoid such situations, it is necessary to reason about a proposed architectural style based on COM at an early stage. To this end, they proposed a formal model for COM based on basic first-order set theory concepts, expressed in Z notation. Its syntax was checked by the Z/Eves system (Sullivan et al., 1997).

Using Z to formalize COM has led the model to be unambiguous and consistent, allowing software developers to have more confidence in their COM-based projects. Based on this model, some expected properties of COM can be deduced. More importantly, this model has revealed subtle features of COM that developers may ignore or misinterpret during developing COM based software systems, thereby empowering the developers to easily find software design faults at early stage without sacrificing more time and cost.

4.3 Summary

In this chapter, we have introduced Petri Nets, and briefly described two formal models for COM. One is the model proposed by Ibrahim and Szyperski (1998) presented as a language COMEL described using appropriate programming language formalisms. Another is the model by Sullivan, Socha, and Marcjukov (1997) expressed in Z notation. Both of these formal models precisely specify COM, and could be used by developers to design and build robust COM-based software. Although both these formalisms are potentially quite useful for developing and reasoning about COM-based software, they apply only to COM. Even though it may be possible to extend them to other component models, they necessarily include details that are COM- specific, and therefore obscure the fundamental characteristics of components common to all models.

In summary, therefore, a general model is needed for component-based software. The goals for the general model are similar to those of the above two formal COM models. The emphasis of the general model, however, should be placed on capturing the fundamental common features of component-based software. Moreover, the model should be expressed in a succinct and clear style so that people can understand it easily. We will present our general model for component-based software in the next chapter.

CHAPTER FIVE

A General Model for Component-based Software

On the basis of our studies of three commonly used component models, JavaBeans, COM, and CORBA, we propose a general model to formalize component-based software systems. The model consists of definitions of the concept of “component” and its properties. The model offers the following features:

1 It captures the essence of component-based software at a high level;
2 It provides a simple but precise characterization of components which may clarify the confusion that results from the rapid evolution of many subtly different component technologies;
3 It describes an underlying structure on which the syntax of component-based languages can be based, as well as a semantics for such languages.
4 Components are normally delivered as black-boxes with an interface specification. The source code and other artifacts may be entirely unavailable. This may cause difficulties in evaluating and testing the component. We believe that our model could be used as a starting point to develop a formal testing and verification methodology.
5 Our model incorporates recursions, a control structure not incorporated in any existing component-based systems, to our knowledge.

5.1 Definitions

The following notional conventions will be used in our descriptions:

Notational conventions: When an entity X is defined as a tuple, we can refer to the constituents of X in various ways. For example, consider definition 5.1.5 below. If X is a simple component, we can refer to the first constituent of X as **inports**(X), or by the phrase "the inports of X". If X is understood in context, we can omit the X, and simply write **inports** or refer to "the inports". If an item is a set, we will usually give it a plural name so that we can refer to its members in the singular. For example, if X is a simple component, we can use phrases such as "an inport of X".

If f is a function with domain S and range T, and S' is a subset of S, then f(S') denotes the subset $\{ f(s) \mid s \in S' \}$ of T. If S' is an n-tuple or sequence of elements of S for some n, then f(S') denotes the n-tuple or sequence of elements of T obtained by applying f to each element of S'. When it is convenient to do so, we will treat a tuple or sequence as if it were a set, in which case we mean the set consisting of all elements occurring in the tuple or sequence.

Definition 5.1.1: The *domain* D is a set which does not include the element *none*. A is an arbitrary but fixed infinite set called the set of *attributes*, each element x of which is associated with a subset of D denoted $\tau(x)$ called the *type* of x.

Three important concepts are introduced in this definition: *domain*, *attribute*, and *type*. The *domain* D might include values from any data type such as integers, strings, or instances of classes. Attributes can be thought of as variables or data fields. τ is a function that defines the set of all elements of D that can be values of an attribute. For example, suppose D includes the set **Z** of all integers, and for some attribute *x* of A, $\tau(x) = \mathbf{Z}$, then τ defines the type of *x* as integer.

Definition 5.1.2: A *component over* A is either a source over A, a sink over A, a simple component over A, a compound component over A, or a prototype over A.

Definition 5.1.3: A source *over* A is an attribute. If X is a source, **attributes**(X) and **outports**(X) are both defined to be the set consisting of the single attribute X.

In general, a component produces data at its outports in response to the arrival of data at its inports. A source component has no inports, so from the point of view of the application in which it is embedded, a source component spontaneously generates data. For example, a button component generates an output value when the button is clicked rather than in response to receiving an input.

Pictorially, a source can be represented as a box with the outport represented by an arrow on the perimeter pointing out from the box as shown in Figure 5.1.

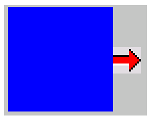


Figure 5.1 A Visual Representation of a Source Component

Definition 5.1.4: A sink *over* A is an attribute. If X is a sink, **attributes**(X) and **inports**(X) are both defined to be the set consisting of the single attribute X.

A sink is a component with only inports as shown in Figure 5.2. It receives its input data via its inports. If we can think of sources as active components, we can think of sinks as passive ones. The visual representation of a sink is similar to a source except that the single attribute is represented by an arrow pointing into the box. An example of a sink is a text field which displays the data it receives on its inport.

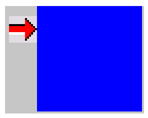


Figure 5.2 A Visual Representation of a Sink Component

Definition 5.1.5: A *simple component over* A is a 4-tuple X of the form (**inports**, **outports, function**, **triggers**) where:

- **inports** is an n-tuple ($a_1, \ldots, a_n$) of attributes for some integer $n \geq 0$;
- **outports** is an attribute distinct from $a_1, \ldots, a_n$;
- **function** is an n-ary function from $\tau(a_1) \times \ldots \times \tau(a_n)$ into τ (**outports**);
- **triggers** is a set of pairs of the form (**target**, **relation**), where **target** is an attribute distinct from the outport, and **relation** is a binary relation on τ(**target**).
- **attributes**(X) is defined to be the set of attributes consisting of the inports, the outport and the targets of X.

The role of simple components in a component-based system is analogous to that of primitive functions such as arithmetic operations or library routines in a programming language: that is, they provide services implemented using a different formalism or language. Ports (**inports** and **outports**) provide the interface through which the component interacts with other components. The functionality of a component is implemented through **function**. Triggers provide the mechanism for initiating execution of the component, the importance of which will be seen in the following definitions.

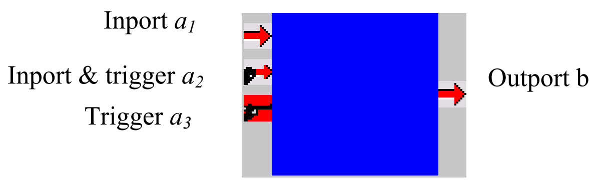


Figure 5.3 A Visual Representation of a Simple Component

Figure 5.3 shows the visual representation of a simple component. Inports and triggers are located on the left of the box and outports on the right side. Since a trigger may or may not be an inport, there will be three combinations on the left of box, only inport represented by an arrow, only trigger represented by a little gun-like icon, and both inport and trigger represented by icon with an arrow on one end and gun-like symbol on the another end.

An example of a simple component would be the computation for the sum of two integer values (Figure 5.3). **Inports** is a 2-tuple of attributes (a_1, a_2) and **outports** is an attribute *b*. **function** adds two integer values together and then assigns the result to τ (**outports**), an integer in *domain*. The output of the component will be provided through outports. There are two triggers for this simple component. The target for one trigger is a_2 and expressed as $(m, n)| \; m \neq n$. The target for another trigger is a_3 and expressed as $(m, n)| \; m=1$. The relation for target a_2 says that new value is not equal to old value. The relation for target a_3 says that the value of target is equal to 1. We can see that in the above example, a_1 is only inport, a_2 acts as both inport and trigger, and a_3 is only trigger.

The execution of the simple component will be defined in the following definition. But for now, we can simply think that when the relation for trigger is true, the simple component will perform its function evaluation. For example, when the relation for the trigger a_3 is true, the sum of two values for a_1 and a_2 will be calculated.

Definition 5.1.6: A *compound component over* A is a 4-tuple X of the form (**components**, **inports**, **outports**, **connections**) such that:

- **inports** is a sequence of distinct attributes.
- **outports** is a sequence of distinct attributes.
- **components** is a set of components, not including X.
- **attributes**(X) is defined as the set **inports**(X) $\cup$ **outports**(X) $\cup$ $\{ x \mid \exists Y \in$ **components**(X) such that $x \in$ **attributes**(Y) $\}$.

- The sets **inports**(X), **outports**(X), **attributes**(Y) and **attributes** (Z) are pairwise disjoint for any Y, Z $\in$ **components**(X).
- **connections** is a set of pairs of the form (**origin**, **destinations**) such that if K is a connection then
    - **origin**(K) is either an inport of X or an outport of a component of X
    - **destinations**(K) is a set of attributes of X not containing **origin**(K)
    - For each destination d of K, τ(**origin**(K)) $\subseteq$ τ(d).

A compound component is a network of connected components. This is a very important concept as it allows thinking about problems at the appropriate level of abstraction. A connection associates an outport of one component, the origin of the connection, with the inports of one or more other components, the destinations of the connections. A connection indicates the passage of data from origin to destinations. Each destination must be able to accept any value it receives from the origin, so its type must be a subset of the type of the origin.

Figure 5.4 (a) shows the visual representation of a compound component. It looks like a simple component except for the color of box. It is important to note that its visual representation has only inports, and does not have trigger or both inport and trigger as a simple component does. Furthermore, the compound component may have many outports.

The details for the compound component are hidden from its visual representation. Figure 5.4 (b) shows a detail for the compound component as shown in Figure 5.4 (a). The compound component consists of two simple components. The connections are expressed as lines, linking together all components by which the functionality of the compound component is accomplished. Consider connections K_1 and K_2 in Figure 5.4(b). They can be expressed as K_1= (a_4, {a_5, a_6}) and K_2= (b_3, { b_2}). Attributes for the compound component includes inports, outports, and all the attributes of the components inside the compound component.

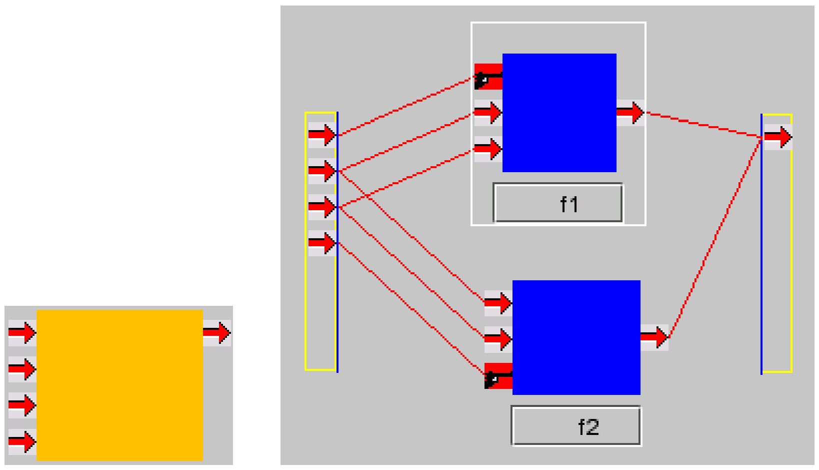


Figure 5.4 A Visual Representation of a Compound Component

Definition 5.1.7: Two components X and Y are *equivalent* iff there exists a bijection Φ: **attributes**(X) → **attributes**(Y) such that

- for all x ∈ **attributes**(X), τ(x) = τ(Φ(x))
- **inports**(Y) = Φ(**inports**(X))
- **outports**(Y) = Φ(**outports**(X))
- if X and Y are compound components, then
  - **connections**(Y) = Φ(**connections**(X))

  and there is a bijection Ψ: **components**(X) → **components** (Y) such that for each component Z of X
  - Ψ(Z) is equivalent to Z
  - Φ(**attributes**(Z)) = **attributes**(Ψ(Z))
- if X and Y are simple components, then
  - **function**(Y) = **function**(X)
  - **triggers**(Y) = { (Φ(**t**), **C**[x_1, x_2]) | (**t**, **C**[x_1, x_2]) ∈ **triggers**(X) }
- if X and Y are prototypes, then **class**(X) = **class**(Y).

According to this definition, components are equivalent if they are syntactically identical. It is easy to show that the relation "equivalence" is, in fact, an equivalence relation on components, which leads to the next definition.

Definition 5.1.8: If U is the set of all components, then U is partitioned into equivalence classes by the equivalence relation defined above. Each such class is called a *component class*.

Since the semantics of components, provided below, ascribes identical behavior to equivalent components, the partitioning provided by equivalence provides a basis for procedural abstraction, as follows.

Definition 5.1.9: A *prototype over* A is a triple X of the form (**class**, **inports**, **outports**) where

- **class** is a component class the elements of which are compound components.
- **inports** and **outports** are mutually exclusive sequences of distinct attributes of the same lengths respectively as **inports**(Y) and **outports**(Y) for any Y $\in$ class.
- **attributes**(X) is defined to be the set **inports**(X) $\cup$ **outports**(X).

Note that since prototypes can occur in compound components, the procedural abstraction mechanism provided by prototypes naturally includes recursion, which is not normally a feature of component software models.

Figure 5.5 shows the representation of a prototype based on the class of which the compound component in Figure 5.4 is a representative. Its appearance is the same as the compound component the class of which it is based, except that its box is green.

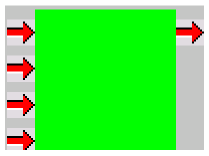


Figure 5.5 A Visual Representation of a Prototype

Definition 5.1.10:

- A component Y is said to *occur in* a component X iff either X = Y or X is a compound component and Y occurs in a component of X.
- A connection K is said to *occur in* a component X iff either X is a compound component and K is a connection of X, or K occurs in a component that occurs in X.

Having defined the five categories of components, we now define their semantics.

Definition 5.1.11: A *state* of the set A of attributes is a function σ: A $\rightarrow$ D such that $\sigma(x) \in \tau(x) \cup \{none\}$ for each $x \in$ A.

A state is an assignment of values to all attributes. As we shall see, an execution of a component assumes a certain starting state, which is transformed as the execution proceeds.

Definition 5.1.12: If X is a component and σ is a state, an *execution of* X *from* σ is a sequence of the form $(\sigma_0, X_0, S_0), (\sigma_1, X_1, S_1), (\sigma_2, X_2, S_2), \ldots$ where

- for each $i \geq 0$, σ_i is a state, X_i is a component, and S_i is a subset of the simple components occurring in X_i
- $\sigma_0 = \sigma$; $X_0 = X$; $S_0 = \varnothing$
- for each $i \geq 1$
  - (σ_i, X_i, S_i) is a propagation of $(\sigma_{i-1}, X_{i-1}, S_{i-1})$ if one exists;
  - otherwise, (σ_i, X_i, S_i) is an expansion of $(\sigma_{i-1}, X_{i-1}, S_{i-1})$ if one exists;
  - otherwise, (σ_i, X_i, S_i) is an evaluation of $(\sigma_{i-1}, X_{i-1}, S_{i-1})$

This definition divides execution of a component into three phases: propagation, expansion and evaluation defined below.

The propagation rule, as its name implies, moves values that have been generated by a component along connections from the component's outports to other components. During this process, some of these values may arrive at triggers of some simple components, which may become ready to execute as a result. The third item in each of the elements of an execution is the set of all simple components which are ready to execute.

The evaluation rule applies to the execution of the set of simple components which are ready to execute. During the evaluation process, the function of a simple component is evaluated and its resulting value is set to the outport of the simple component.

The expansion rule deals with a prototype. During the process, the prototype will be replaced by a compound component that is an instance of its class. The execution of the compound component is carried out in the same way as the above three phrases.

Definition 5.1.13: If σ and σ' are states, X is a component, and S and S' are subsets of the simple components occurring in X, then (σ', X, S') is a *propagation* of (σ, X, S) iff for all x ∈ A

- if x is a destination of a connection K occurring in X such that σ(**origin(**K)) ≠ *none*, then σ'(x) = σ(**origin(**N)), where N is a connection of which x is a destination and σ(**origin(**N)) ≠ *none*.
- if x is the origin of a connection occurring in X such that σ(x) ≠ *none* then σ'(x) = *none*
- otherwise σ'(x) = σ(x).
- S' = S ∪ { Y | Y is a simple component occurring in X

$\wedge$ Y has a trigger t

$\wedge$ t is a destination of a connection K occurring in X

$\wedge$ σ(**origin**(K)) ≠ *none*

$\wedge$ (σ'(**target**(t)), σ(**target**(t))) ∈ **relation**(t) }

Consider a compound component X shown in Figure 5.6. This compound component consists of two source components Sr1 and Sr2, a simple component Sc, and a sink component Si, connected as shown. All attributes are of integer type. The function f of Sc is defined by $f(x) = x^2$ for any integer x. The relation of the trigger of Sc is { (x,y) | x≠y}. Assume the triple characterizing the execution is (σ, X, ∅) where σ(a) = 2, and σ(b) = σ(c) = σ(d) = *none*. Propagation produces the triple (σ', X, ∅'), where σ'(b) = 2, and σ'(a) = σ'(c) = σ'(d) = *none,* and since σ'(b) ≠ σ(b), Sc is ready to execute, so that ∅' = { Sc }.

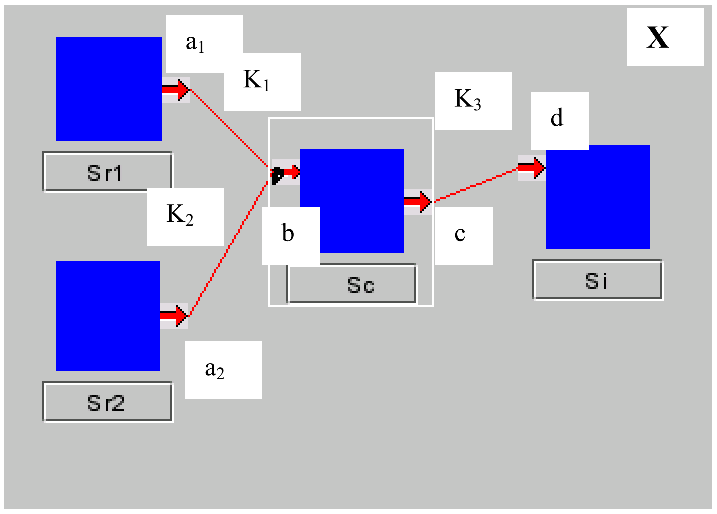


Figure 5.6 An Example

Definition 5.1.14: If σ and σ' are states, X and X' are components, and S and S' are subsets of the simple components occurring in X and X' respectively, then (σ', X', S') is an *expansion* of (σ, X, S) iff

- either S’ = S and X is a compound component and X’ is a compound component identical to X in every respect except that **components(X’) = components**(X) – {Y} ∪ Y’ where (σ’,Y’,∅) is an expansion of (σ,Y,∅).
- or S’ = S = ∅, X is a prototype and X’ is a compound component such that X’ ∈ **class**(X), **inports**(X’) = **inports**(X) and **outports**(X’) = **outports** (X) and σ’(x) = σ(x) if x ∈ **attributes**(X) and σ’(x) = *none* otherwise.

Expansion replaces a prototype with a compound component which is an instance of its class. As we have already mentioned, this process is the equivalent of procedural abstraction in programming languages, and therefore provides the basis for recursion. The example in Section 5.2 illustrates this process.

Definition 5.1.15: If σ and σ’ are states, X is a component, and S is a subset of the simple components occurring in X, then (σ’, X, Ø) is an *evaluation* of (σ, X, S) iff for all x ∈ A

- if x is an outport of Y for some Y ∈ S, then σ’(x) = **function**(Y) (σ(a_1), …, σ(a_n)) where **inports**(Y) = (a_1, …, a_n)
- if x is an outport of some source occurring in X, then σ’(x) ∈ τ(x) ∪ {*none*}
- otherwise σ’(x) = σ(x).

The evaluation rule describes the execution of a simple component. Consider the example following Definition 5.1.13, where applying propagation resulted in the triple (σ’, X, { Sc }) where σ’(b) = 2, and σ’(a) = σ’(c) = σ’(d) = *none*. Applying evaluation leads to the triple (σ”, X, Ø) where σ”(b) = 2, σ”(a) = σ’(d) = *none* and σ”(c) = $σ”(b)^2$ = 4.

5.2 Discussion with an Example

In this section, we go through an example to demonstrate how our model works. The example is to recursively calculate the factorial of a number.

We assume *domain* D includes integers and all attributes in this example have integer type. A compound component **fact** wraps all the calculation detail and is comprised of three simple components, one prototype of **fact** itself, one inport x, one outport y, and five connections. This can visually be represented as shown in Figure 5.7.

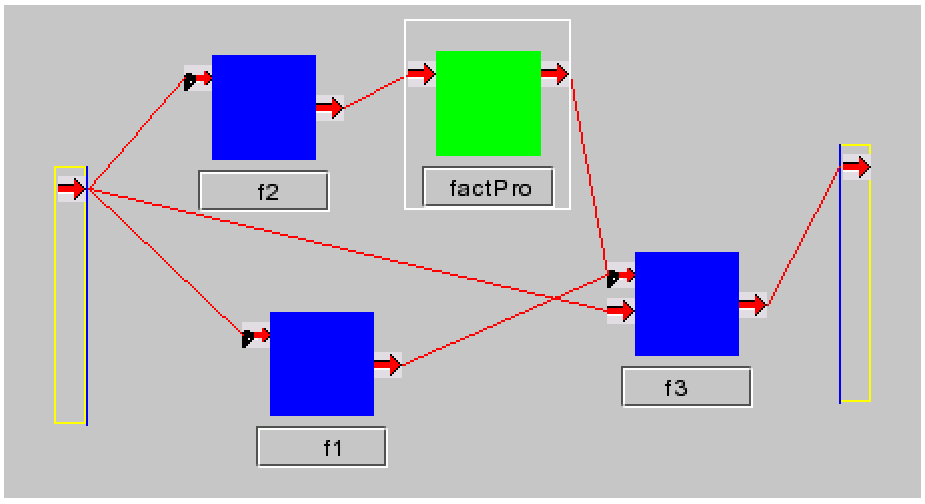


Figure 5.7 The Component **fact**

Formally, the component **fact** is defined as follows.

fact = ({ $\mathbf{f_1}$, $\mathbf{f_2}$, $\mathbf{f_3}$, **factPro**},(x), (y), {(x, {t_1, t_2, i_3}), (o_2, {t_f}), (o_f, {t_3}), (o_1, {t_3}), (o_3, {y})})

The three simple components $\mathbf{f_1}$, $\mathbf{f_2}$, $\mathbf{f_3}$, are as follows.

$\mathbf{f_1}$ = ((t_1), o_1, F_1, {(0,m), (1,m) }) where $F_1(i)$ = 1 for all integers i

$\mathbf{f_2}$ = ((t_2), o_2, F_1, {(n,m) | $n > 1$}) where $F_2(i)$ = i - 1 for all integers i

$\mathbf{f_3} = ((t_3, i_3), F_3, \{(n,m) \mid \text{either } n \neq m \text{ or } n = 1\})$

where $F_3(i,j) = 1$ for $j = 0$ and all integers i

$F_3(i,j) = i * j$ for $j > 0$ and all integers i

The prototype **factPro** is defined as follows.

factPro = (**factClass,** t_f, o_f)

Where class **factClass** is the component class that contains the compound component **fact.**

A simple component not only performs the basic computations, but also controls the execution flow of program. The function of $\mathbf{f_1}$ is to decide whether or not to invoke the base case, whereas the function of $\mathbf{f_2}$ is to decide whether or not to invoke the recursive case of factorial, passing a value to $\mathbf{f_3}$ only if the incoming integer is greater than 1. The function of $\mathbf{f_3}$ is to calculate the multiplication of two numbers.

To illustrate the above definitions, we will now describe the execution of **fact** given a starting value of 2. The paragraphs that follow describe successive triples in the execution, and show how each triple is derived from the previous one. In each paragraph, we will describe the new state by giving only the attribute values which have changed: attributes not mentioned have the same values as in the previous state.

(σ_0, **fact**, Ø)

$\sigma_0(x) = 2$, and $\sigma_0(a) = \textit{none}$ for all attributes $a \neq x$.

Propagation applies since $x \neq \textit{none}$ and is the origin of two connections.

(σ_1, **fact**, $\{\mathbf{f_2}\}$)

$\sigma_1(x) = \textit{none}$, and $\sigma_1(t_1) = \sigma_1(t_2) = 2$.

Since $\sigma_1(t_1)$ is not 1 or 0, $\mathbf{f_1}$ is not triggered. Since $\sigma_1(t_2) > 1$, $\mathbf{f_2}$ is triggered.

Propagation does not apply since no there is no origin with a value other than *none*.

Expansion does not apply since the third element of the triple is not Ø.

Evaluation applies, yielding the following triple.

(σ_2, **fact**, Ø)

Although the above example is quite simple, it does demonstrate most features of our model. It shows how a compound component is composed of other components, particularly including prototypes. From the example, we see that the notion of "recursive component" is a natural extension of the usual concept of component. The example also shows that the concept of compound component includes the notion of "program".

An execution of a program is thought of as a sequence of steps, each producing a new state by changing the values of one or more attributes. Each step is accomplished by propagating values then evaluating the functions of simple components that have been triggered by the arrival of new data.

Visually we can construct a component-based program by connecting components together according to our definitions. The components must satisfy the component definitions and connections cannot be made unless the types of connected attributes are compatible according to definition 5.1.6. Further features of our model will be demonstrated in a component-based visual programming environment, described in the next chapter.

5.3 Comparison with Other Component Models

Generally our model captures the essence of the JavaBean, COM, and CORBA component models at a high level. As has been stated, all component models emphasize that the functionality of a component is provided with an interface, and an implementation is encapsulated behind interfaces. In our model, the interface is expressed in term of ports (inports and outports). All the internal detail is hidden inside component. Through ports, a component can receive inputs or send outputs. Encapsulation is a characteristic that all components should possess.

Although the three component models, JavaBean, COM, and CORBA, have different mechanisms by which components interact with each other, they do use “connections” to facilitate communication between components via some specific connectors provided by the components. This is reflected in our model by the definition of **connections** with matching rules to enforce type correctness. Our model does not explicitly dictate the mechanisms used in connections. The connections could therefore be via event-listener as in JavaBeans, via pointers as in COM, or something else.

In our model, we classify five kinds of components, sources, sinks, simple components, compound components, and prototypes. This classification makes concepts much clearer and also allows us to achieve a certain level of programming abstraction.

We explicitly define a source component and a sink component. Although the three component models embed these two kinds of components implicitly, they do not explicitly define them. This lack may make people to confuse the concepts.

As has been stated, simple components provide primitive functionality in component-based software. Definition of simple component captures the core concept of component, that is, all services are provided via ports. This concept is reflected in all the three component models.

Our definition of compound component provides an abstraction mechanism analogous to begin-end blocks in a programming language. The component can contain components. For instance, a compound component may consist of several simple components, another compound component, and a prototype of the same compound component. But from outside, we still see only ports. This mechanism is the same as aggregation mechanism in the above three component models. In other words, the concept of compound component exists in the three component models implicitly.

One important concept in our model is “prototype”. As we discussed, the prototype concept provides an abstraction mechanism analogous to procedures in programming languages. One of its important consequences that our model provides for recursive components, thereby extending the usual notion of “component” in a natural and consistent way. There is no equivalent definition in the above three component models. In particular, the end-user JavaBean component for Java Studio does not have the prototype kinds of components. As a result it can not realize a recursive program as described in Chapter 3.

5.4 Relation with Petri Nets

Although the concepts of Petri Net have influenced the visualization of our general component model, we have found that it is hard to use Petri Nets as a basis for component-based software because such software requires concepts difficult to express in Petri Net graphical notation. For example a component possesses more properties than a simple “place” in Petri Net, whereas in Petri Net a “place” simply represents conditions for execution. In Petri Net tokens are used to control the execution of the Petri Net, conveying a signal to the next “place”. The signal can be thought as a special value. In our model, a message with data is travelling through the connection to control the execution of a program.

5.5 Summary

In this chapter, we defined our general model for component-based software. Discussions with an example have demonstrated most features of the model. One unique feature of the model is that recursion is naturally embedded, extending the usual definition of component in other models. We have compared our model with three leading component models, CORBA, COM and JavaBeans, showing how it captures their essence. Its relationship with Petri net has also been discussed.

CHAPTER SIX

A Prototype of a Component-Based Visual Programming Environment

As has been noted, our general model for component-based software captures the essence of various component models, providing a useful guideline for developing component-based software. The goals in designing and implementing a prototype of component-based software development system, Component Software Construction Kit (CSCK), are similar: to build a platform for verifying the applicability of our model to the software development task. With CSCK, we should be able to check how easy or difficult it is to write certain kinds of programs using our model, to discover the kinds of programming tasks it is good for and those it is not good for. It is also hoped that CSCK might be used to help people understand the concepts of component-based software.

6.1 Overview of CSCK

CSCK is implemented as a simple integrated development environment. The integration of visual programming principles into this environment is intended to simplify the development process, and benefit programmers in terms of development time and productivity by allowing the programmer to build execute, modify and debug components within a single environment. CSCK is implemented in the Java language. The reader is referred to Appendix A for the details of design and implementation.

CSCK consists of several editors and an interpreter. There are five different editors, the project editor, compound component editor, function editor, port editor, trigger and inport editor, and visual bean selection editor. Each editor appears in its own window which can be invoked by double-clicking an icon of an appropriate kind. These editors are used to create components, manipulate these components by adding ports and specifying component function, for example, and to construct a component-based software program. The interpreter traverses the data structure representing a component-based program built by the editors, and executes it according to the semantics of our model.

Unlike textual languages in which programs are encodings of the various multi-dimensional structures inherent in the syntax, the success of a visual language depends to a large extent on providing the programmer with consistent and meaningful concrete representations of syntactic structures. Designing a friendly GUI for such a language is therefore an important issue.

In designing CSCK we have tried to keep the GUI as simple and consistent as possible. An important aspect of consistency is adhering to the "look and feel" of the host operating system, since the time taken for a user to become familiar with new software can be shortened if that software employs familiar interface conventions. In order that the prototype system is consistent in this way, we have used the Java Swing classes to implement the GUI, since Swing provides a rich set of standard "look and feel" conventions (Sun Microsoft, 1999).

Color can be a powerful tool for communication if used correctly. We have carefully used color to convey meaningful semantic distinctions. For example, different colors are used to represent different components. It should be noted that in order enable color-blind users to use the system, we should use other means to remedy the possible consequence caused by the use of color.

It should be noted that Prograph has much influenced the design and implementation of CSCK. Some visual concepts and design styles for the prototype have been borrowed from Prograph. For example, an editor window can be opened by double-clicking its corresponding visual representing icon. Like Prograph CPX, our prototype implements an interaction style which encourages a top-down approach to software development.

6.2 Working with the System

In this section, an example is given to demonstrate how the system works and what features its environment has. The example is the same as that in Chapter 5, that is, to recursively calculate the factorial of a number.

When we start the system, the primary window appears, as shown in Figure 6.1. In this primary window, the user can construct a component-based program and execute it. Eight menus accompany the primary window, **File**, **Edit**, **Component**, **Prototype Classes**, **Project**, **Look & Feel**, **Window**, and **Help**. Table 6.1 lists the items on each menu.

Toolbars that float under the menu bar are provided for frequent actions across screens common to several different kinds of windows, for example, **new**, **open**, **save**, **simple component**, **compound component**, **prototype class**, **generate code** and **run**. Tooltips, the labels that appear as the mouse passes over toolbar graphics is also provided.

Menu	Menu Items	Description
File	New	Create a new project
	Load*	Open a project
	Save*	Save project
	Exit	Exit the system
Edit	Copy*	Copy the selected component
	Paste*	Paste component
	Delete	Delete the selected component
Component	Simple	Select simple component
	Compound	Select compound component

Prototype Classes		The menu items will dynamically be added as compound components are added to the project (need future improvement)
Project	Generate Code*	Generate corresponding JavaBean code
	Run	Execute the program
Look & Feel	Metal	Default Java Look & Feel
	CDE/Motif	X Windows Look & Feel
	Windows	MS Windows Look & Feel
Window	New Window*	
	Arrange All*	
Help	Help*	Provide help
	About	Displays information about the software.

Table 6.1 Menu Items for each Menu of Primary Window

NOTE: MENU ITEMS WITH * ARE NOT IMPLEMENTED IN THE PROTOTYPE FOR THE TIME BEING.

When the user selects the **new** item from the **File** menu, a project editor appears. The **Project editor** is the main graphical editor for manipulating all components and their connections. Figure 6.2 shows a snapshot of a project editor identified by a network-like icon in the upper left corner of the window.

The diagram in the project editor consists of a network of connected components. Although we have already seen examples of component icons in the last chapter, we will describe their representations here for completeness.

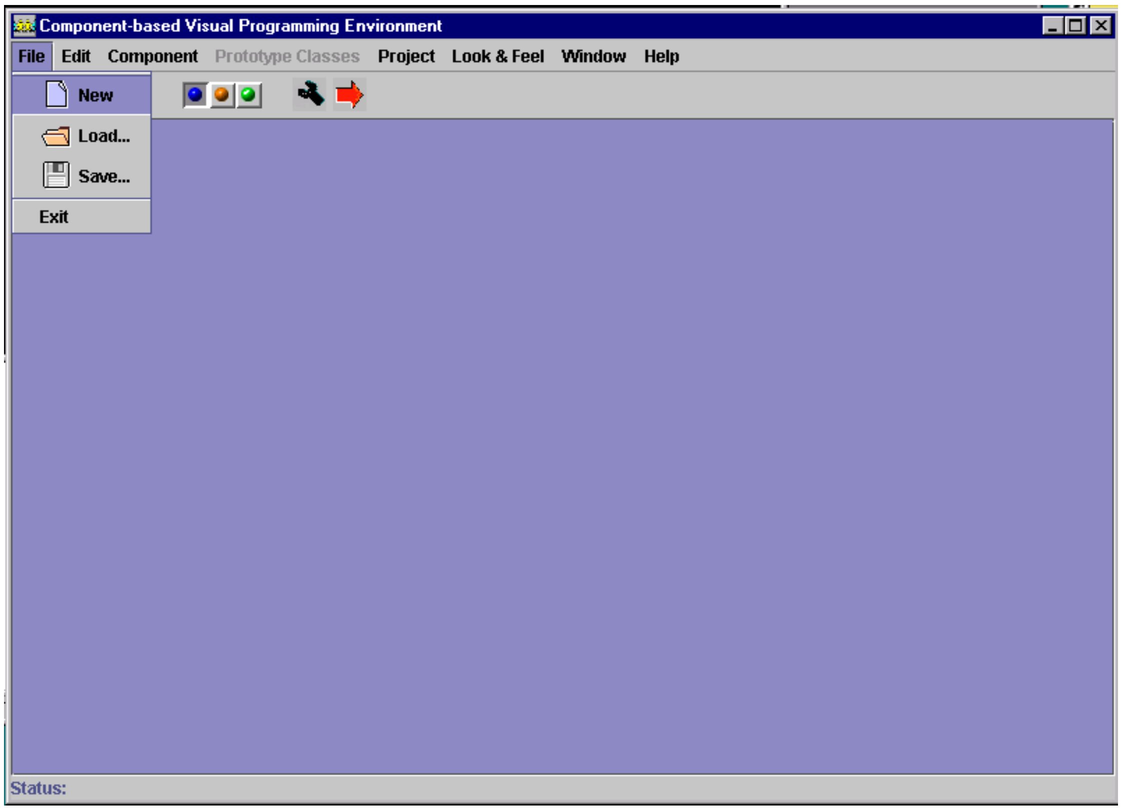


Figure 6.1 A Snapshot of CSCK after Starting-up

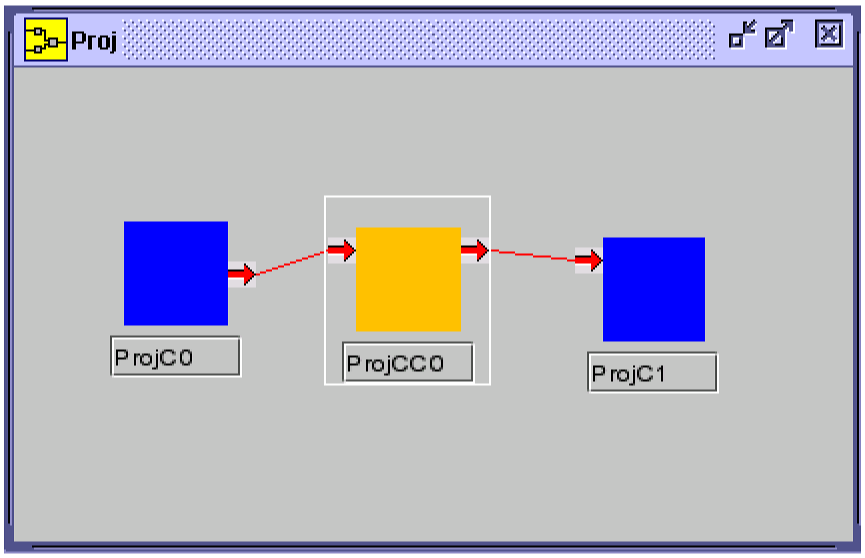


Figure 6.2 Project Editor

Component

A rectangular box with a title in a text field is used to represent a component. Different colors are used to represent different kinds of components. Blue, brown and green represent simple, compound, and prototype components respectively. Since source and sink components can be considered special kinds of simple components (a simple component with only one *inport* is a sink and a simple component with only one *outport* is a source) we still use blue to represent them. Figure 6.3 illustrates these five different kinds of components. It should be pointed out that in a full implementation of CSCK we would not rely solely on color but also use other devices such as different shapes to represent different kinds of components in order to allow color-blind people to distinguish them.

In the project editor, the user can select kind of component, using the **Component** or **Prototype Classes** menus, then add a component by double-clicking mouse button 1 at the point where the component is to be added. The user can move a component by dragging it with mouse button 1 depressed.

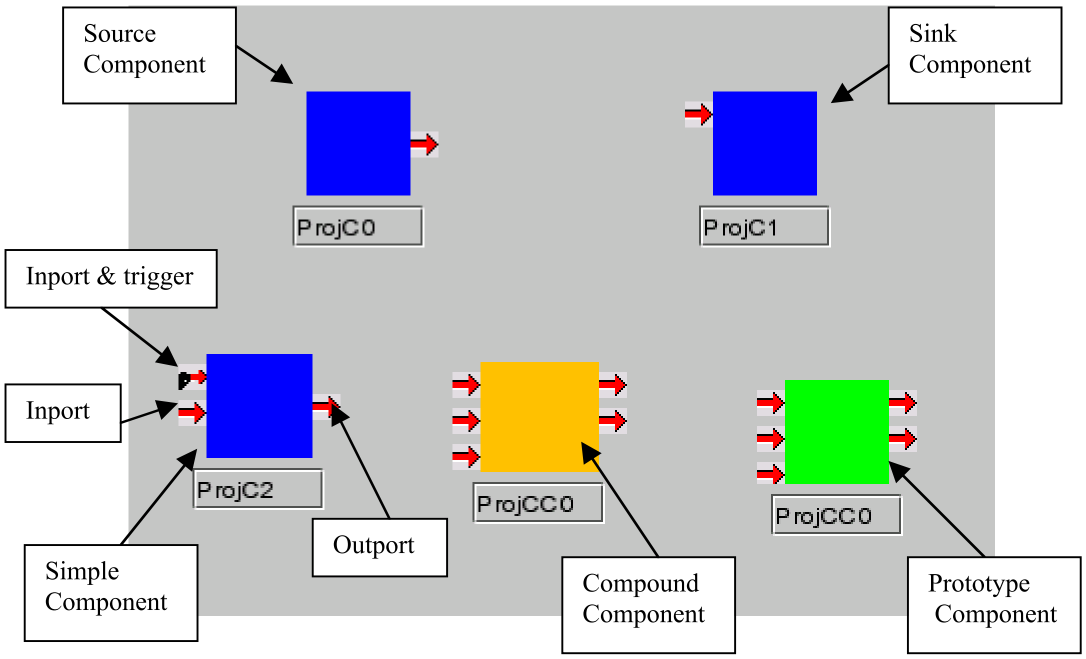


Figure 6.3 Components

Port and Trigger

In general, a port is represented by a red arrow, pointing inwards for an inport and outwards for an outport. Inports are located on the left of a component and outports on the right side. The user can add an inport or outport by double-clicking the left or right side of a component respectively. The environment allows only one outport to be added to a simple component.

Recall that in our model a simple component has triggers. A trigger may or may not be an inport. In the visual representation, there are three combinations for *inport* and *trigger*, represented by the three different icons shown in Table 6.2.

Icon	Description
➡	Inport only
🔫	Trigger only
🔫➡	Both inport and trigger

Table 6.2 Icons for Inports and Triggers

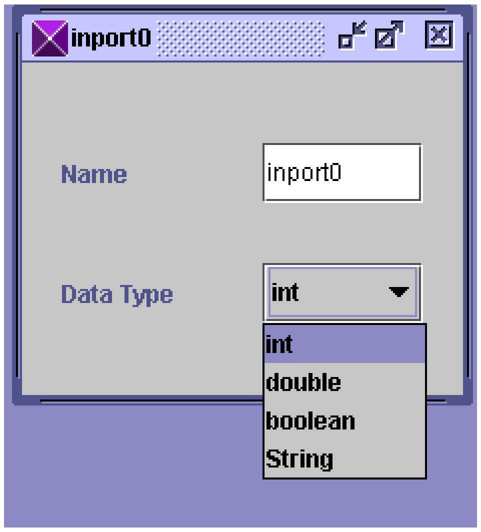
Connection

A connection defines a data transmission path from the outport of one component to the inports or triggers of others. A connection is represented by lines from the origin of the connection to its destinations. To add a destination to a connection, the user clicks mouse button 2 on the origin, drags a “wire” to the new destination on another component, and then double-clicks on the destination. Repeating this action removes an existing destination from a connection.

In the project editor as shown in Figure 6.2, we have created three components, a source component `ProjC0`, a sink component `ProjC1`, and a compound component `ProjCC0`. The source and sink components are used to provide the input and display the output respectively. The compound component will be used to calculate the factorial of number in this example.

Inports, triggers and outports are editable. When the user double-clicks a port or trigger, either a **Port editor** or a **Trigger and Inport editor** appears**.** Using the **Port editor**, the programmer names a port, and specifies its type by selecting from a default list provided in a combo box, as illustrated in Figure 6.4. The four data types, integer, double, boolean, and String are provided. The default data type is integer. In our example for calculating factorial, we choose the default port name and the default data type integer for all the ports.

A **Trigger and Inports editor** invoked by a double-click on an inport of a simple component, is used to name the port and specifying its data type. With this editor the user can also change the port into a trigger, or specify that it is a trigger as well as a port, and can enter the trigger condition. This will be illustrated as we proceed with our example.


inport0

Name: inport0

Data Type: int

Figure 6.4 Port Editor

In our example, when we double-click the compound component in the project editor, a compound component editor appears, indicated by the brown cube in the upper left corner of its window as shown in Figure 6.5. The **Compound component editor** is very similar to the project editor differing only in that its window contains two vertical bars that represent the inports and outports of the corresponding compound component icon. This is analogous to the input and output bars in the case window of Prograph (Prograph International, 1993).

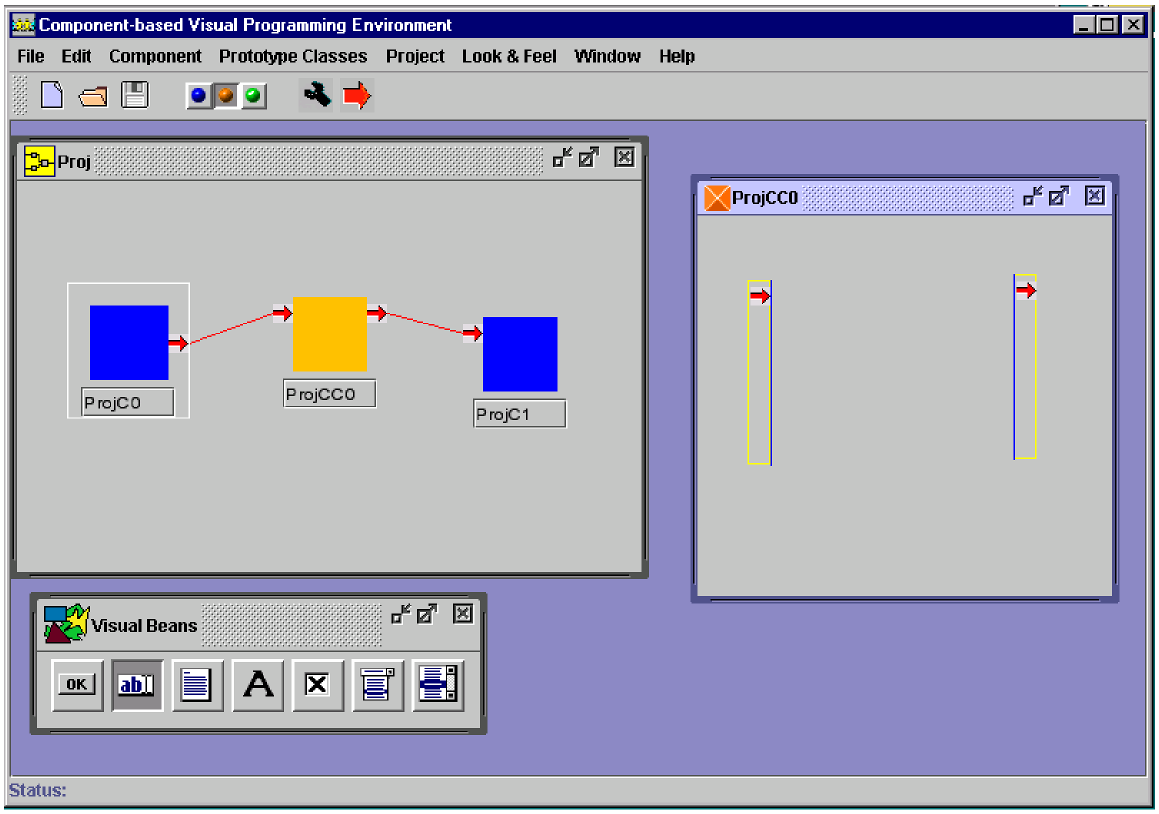

Component-based Visual Programming Environment
File Edit Component Prototype Classes Project Look & Feel Window Help
Proj
ProjC0
ProjCC0
ProjC1
ProjCC0
Visual Beans
OK
ab|
A
Status:

Figure 6.5 Invocation of a Compound Component Editor

In the compound component editor, we construct a component that calculates the factorial of a number as illustrated in Figure 6.6. We have created three simple components, namely, `ProjCC0C0`, `ProjCC0C1`, and `ProjCC0C2`. For each of these, two important tasks have to be performed, specifying triggers and trigger relations, and specifying the function.

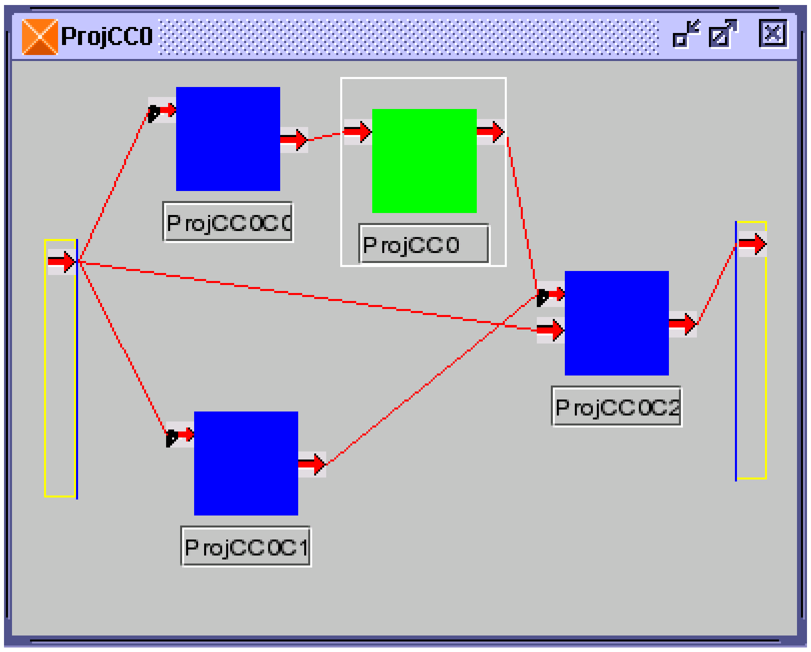
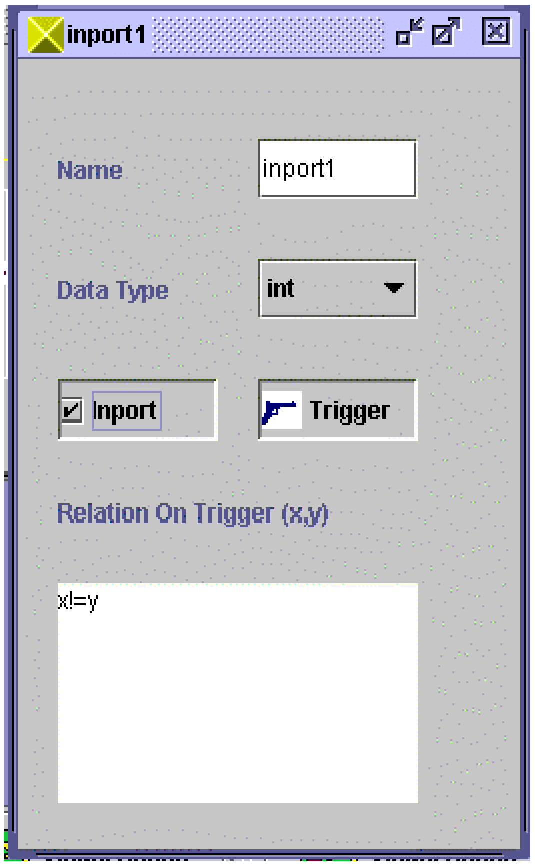


Figure 6.6 Compound Component Editor

Consider the simple component `ProjCC0C2`. We have decided to use its port *inport1* as both inport and trigger. When we double-click the port, a **Trigger and Inports editor** appears as shown in Figure 6.7. Here we can select a data type using a combo box as in the port editor. There are two check boxes, one indicating whether or not the attribute is an inport (by default selected), another for indicating whether or not the attribute is a trigger (by default unselected). A text area is shown if and only if trigger check box is selected, in which the user can type Java code for the trigger relation. In the relation, x and y represent the old and new values of the attribute respectively. Note that this editor only valid combinations of the inport and trigger checkboxes. In our example, we check both check boxes, and specify the trigger relation as x!=y indicating that the component should be executed if the value of the attribute is changed.



Name: inport1

Data Type: int

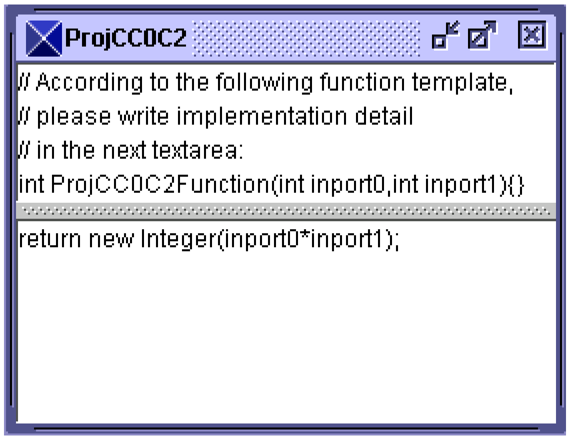
Relation On Trigger (x,y)

x!=y

Figure 6.7 Trigger and Inport Editor

To specify the function of the simple component `ProjCC0C2`, we double-click the component to open the **function editor** shown in Figure 6.8. The **Function editor** has two panes, providing the function template that the programmer is not allowed to change and a text area where the user can type in the function. The function template is created according to the names and data types of the inports and the outport. The function itself is coded in Java in the provided text area, using the information form the template. In our example, we write the function as: "*return new Integer(inport0*inport1);*" since the function template specifies two inports inport0 and inport1 with data type integer, and the returned data type is also integer.

The same procedure is followed to specify the details of the simple components `ProjCC0C0` and `ProjCC0C1`.



```
// According to the following function template,
// please write implementation detail
// in the next textarea:
int ProjCC0C2Function(int inport0,int inport1){}
```

```
return new Integer(inport0*inport1);
```

Figure 6.8 Function Editor

In the compound component editor shown in Figure 6.6, we have added a prototype component whose class is a component class the element of which is the compound component `ProjCC0`. A menu item corresponding to a compound component will dynamically be added to the menu **Prototype Classes** when the compound component is created. Note that since many or most of the compound components created in a project are likely to be used for one time, we will improve this implementation in the future. We could implement a popup dialog to allow the user to select a component from a list of compound components and then ask for a prototype to create from it.

The user adds a prototype component by first selecting the prototype name in the menu **Prototype Classes,** which places a checkmark by that item in the menu then adding a component to the project editor or compound component editor. The system guarantees that only one kind of component can be selected at any time. For instance, if a user selects a prototype component from the check mark in the menu, then menu item for simple component and compound component will be automatically deselected.

To complete our program components are "wired" together by connections, as explained above. Before we execute it, we have to create an interface by specifying visual beans to correspond to the source and sink components. A **Visual Beans Selection editor** is opened by double-clicking the source or sink as illustrated in Figure 6.9 allowing the user to select the required bean. Icons for these visual beans are organized by toggle buttons in Java Swing, allowing only one visual bean to be selected at one time. In this editor, tooltips are provided to explain the functionality of the available beans. In our example, we have chosen text fields for the source component `ProjC0` and the sink component `ProjC1`.

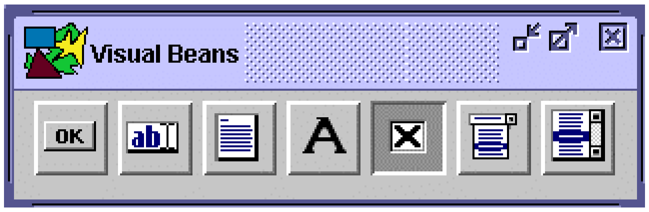


Figure 6.9 Visual Beans Selection Editor

Now we are ready to execute the program. Selecting the **run** item in the **Project** menu or clicking **run** in toolbar, opens an interpreter window as shown in Figure 6.10. In our example, two textfields are displayed in the interpreter window corresponding to the project editor. The left one is for input and the right one for output. When 2 is typed in the left textfield, its factorial 2 is computed as described in Chapter 5, and displayed in the right textfield.

The data structures and algorithms used in the implementation of CSCK are presented in Appendix A.

6.3 Summary

In this chapter, we have described, CSCK, the prototype implementation of our general model for component-based software. The functional operations of the prototype have been introduced with an example. Through the example, the features of the prototype have been demonstrated. We will evaluate this prototype and discuss related issues in the next chapter.

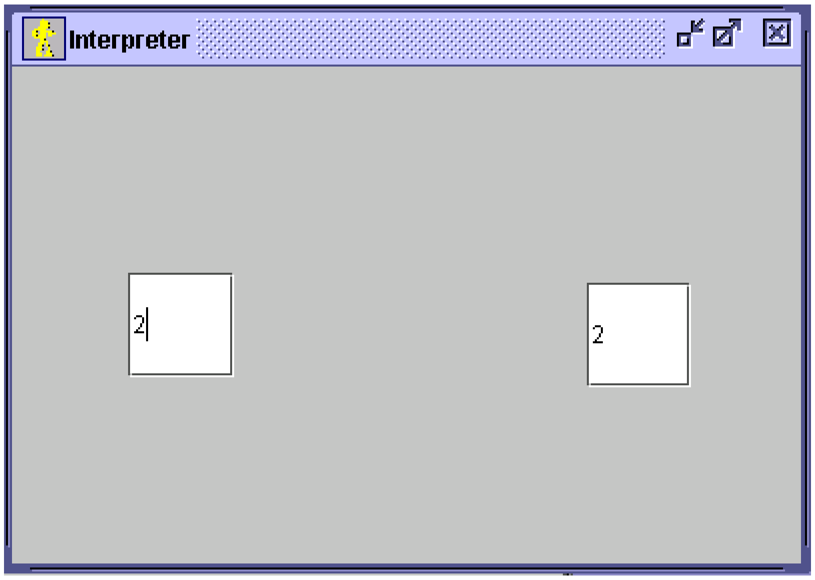


Figure 6.10 Interpreter

CHAPTER SEVEN

Evaluation of the Prototype CSCK

After developing the prototype of component-based visual programming environment, CSCK, we must perform the evaluation and testing. This is the most important stage of software development, particularly for a prototype (Smmerville, 1992). In this chapter, we will evaluate the prototype with respect to the Green and Petre's cognitive dimensions framework (Green and Petre, 1996) against other similar environments, Java Studio, Prograph, and VisualAge for Java. The necessity of testing for CSCK will be discussed.

7.1 Evaluation of CSCK According to Cognitive Framework

The cognitive dimensions framework of Green and Petre (1996) is a set of commonly accepted, qualitative, visual programming language evaluation criteria. We use this framework to evaluate the prototype against three visual programming environments, Prograph, VisualAge for Java, and Java Studio. Although it may be not appropriate to evaluate CSCK at this stage, it is hoped the evaluation will reveal something valuable that can be used for further improvement and development. Readers are assumed to be familiar with the cognitive dimensions framework.

Abstraction Gradient

Both VisualAge for Java and Java Studio are based on the JavaBean component model implemented by Java; so they are object-oriented systems. Prograph can be used a full object-oriented programming environment. Therefore, these environments have all abstraction mechanisms that an object-oriented language offers.

In our general model we have provided compound component and prototype component concepts which allow programmers to construct a component-based program incrementally. In CSCK, the coding detail for a compound component is hidden from its visual representation. The abstraction gradient in CSCK can be considered gentle.

Closeness of mapping

The closeness of mapping between a problem world and a program world is very important in the programming. The closer the mapping is, the easier the problem should be solved. Java Studio provides a good mapping in this respect. Each visual bean affords close mapping to its functional metaphor. For example, a GUI bean component resembles their actual appearances in the design, a timer bean is like a clock, an expression evaluator is like a calculator, etc. Each bean has explicitly expressed ports. A user can write applets, applications or beans by easily “wiring” these metaphors together.

VisualAge for Java only has a good mapping for GUI components. For the user defined JavaBean component, a puzzle-like icon is generally used. Also the interfaces or ports for Javabeans is not explicitly visualized. This makes connections between components not so easy as Java Studio does. Therefore VisualAge for Java is moderate in this dimension.

Prograph is for general programming purpose. Although it did not provide a good mapping to a particular problem, it employs dataflow diagrams which offer a clear logic of the program.

Like Prograph, CSCK is implemented for general component-based software programming purpose. The component is visualized to reassemble the black box in the real world. Its ports are explicitly expressed. The relationships between components are represented by connecting lines, which mimics the networks in the real world. Although there is no specifically functional metaphor to map for each component, we believe that the closeness of mapping in CSCK is still good.

Consistency

Consistency refers *to "when some of the language has been learnt, how much of the rest can be inferred?*"(Green and Petre, 1996). Prograph, Java Studio, and ViualAge for Java are excellent in this dimension. CSCK must be considered strong in the dimension. The programming procedure in CSCK is pretty much consistent. The visual syntax and semantics are logic and consistent. For example, the coding procedure in the compound component editor is the same as the project editor. All editors in CSCK take advantage of the user's familiarity with common windowing and mouse functionality.

Diffuseness/Terseness

Diffuseness defined by Green and Petre (1996) means that "*the more material to be scanned, the smaller the proportion of it that can be held in the working memory, and the greater the disruption caused by frequent searches through tex*t." Green and Petre found that Prograph is much more diffuse in terms of entities that it uses. The entity here refers to as words, icons, connectors, and windows.

Java Studio and VisualAge for Java fare no better. Java Studio requires many beans and connections even for a simply application and the design window becomes very crowded with too many beans and lines. In VisualAge for Java, many connection lines are needed for building an application and it makes the Visual Composition Editor look like a "spider-net" window. CSCK suffers from the same problem. For a simple application, we may need several editor windows. However, it should be note that terseness is not always good either (Green and Petre, 1996).

Error-proneness

Generally speaking, visual programming environments offer less syntax errors than does a textual programming environment (Green and Petre, 1996). In Prograph, Java Studio, and VisualAge for Java, there is little that can go wrong to induce errors.

Like the three environments, CSCK can minimize syntax errors. Editors in the prototype enforce the syntax of the model. Moreover, the concrete visual syntax makes the users to find errors easily. For example, when connecting two components, the data type for ports are automatically checked. Thus the users can not connect two incompatible ports.

Hard mental operations

Hard mental operations concern two questions. *Are there places where the user needs to resort to fingers or pencilled annotation to keep track of what's happening? Does the notation induce serious logic flaws?* (Green and Petre, 1996)

Green and Petre (1996) pointed out that Prograph would result in hard mental operations because of control constructs. Java Studio has not been found any hard mental operations from the author's experience. VisualAge for Java would cause hard mental operations for a large program because the user needs to keep track of each step.

Due to the simplicity of CSCK, it is not possible to have "hard mental operations". There is no need for the user to resort to fingers or pencilled annotation to keep track of what is going on.

Hidden dependencies

Java Studio and VisualAge for Java do well in avoiding the hidden dependency problem. For both products, the relationship (dependency) between two components are clearly visible since connections explicitly indicate which one is dependent on which one. CSCK has the same problem as Prograph in this dimension. Like Prograph, there is no any hidden dependency problem at a local level. The connecting lines make the local component dependencies clearly visible. But there would be a problem with proceeding up the call graph in the same way as the programmer can quickly navigate down the call graph by clicking on icons to open their edit windows (Green and Petre, 1996). To alleviate the difficulty, a searching tool is needed, allowing the programmer to look back the icon from the edit windows.

Premature commitment

A thumbnail description of premature commitment is that *do programmers have to make decisions before they have the information they need?* (Green and Petre, 1996)

Prograph, VisualAge for Java, and Java Studio use the box-and-line to represent a program. There is less commitment to the order of constructing a program than text languages. In Java Studio the user can easily change the decision at any stages. There is no separate build or run step to worry about. However, VisualAge for Java does require the order of connecting components for creating code.

For CSCK, there is no observation of premature commitment. But when a program becomes larger, “visual spaghetti” will occur. To avoid this “visual spaghetti”, the programmer has to look ahead. This problem is one of the common problems in box-and-line visual programming languages or environments

Progressive evaluation

Progressive evaluation is interpreted as *can a partially-complete program be executed to obtain feedback on “How am I doing*”? (Green and Petre, 1996)

Prograph is excellent for progressive evaluation (Green and Petre, 1996). It provides facilities to allow the user to evaluate any method separately. Java Studio is also good for progressive evaluation. It supports immediate feedback through parallel views and provides the interactive environment that lets the user create “live” applications easily. VisualAge for Java is moderate. The user can test out a small piece of code in the Scrapbook window even though VisualAge for Java does not support immediate feedback.

For the time being, the progressive evaluation functionality has not yet been implemented in CSCK. The design concepts for progressive evaluation in Prograph and Java Studio should be applied in the future implementation.

Role-expressiveness

Role-expressiveness concerns whether the reader can see how each component of a program relates to the whole. It can be improved by using meaningful identifiers, well-structured modularity, secondary notation, and so on (Green and Petre, 1996). In Java Studio, each bean can be given a name and users can define identifier for the packaged design. VisualAge for Java integrates both techniques of visual and textual programming. Users can either read the visual programming design or read the generated source codes. This would increase the program readability.

CSCK has provided a compound component concept. The complex programs can be constructed as a compound component. This definitely improves role-expressiveness. But much more work is needed to further enhance this criterion.

Secondary notation

Secondary notations such as layout, color, or other cues can be used to convey extra meaning besides the semantics of the language.

The secondary notation is weak in box-and-line visual programming environments like Prograph, VisualAge for Java and Java Studio. There is no exception for CSCK.
In CSCK, we only use color to distinguish different components. There is almost no secondary notation supported in the prototype. We need to improve this in the future.

Viscosity

Viscosity refers to how difficult the user changes a program. Prograph allows users to add the extra code in the existing code with little difficulty. In Java Studio, when users want to insert some new functions (or bean) into design, they may have to rebuilt many of the wires. Since the program in Java Studio is usually not very large, this problem seems not serious. VisualAge for Java allow users to change program quite easily. It has "undo" and "redo" facilities while building an application. Besides, since VisualAge for Java supports textual programming, the user can change the program by just typing a few lines as long as the added codes do not effect the connections.

CSCK is quite viscose at present. It is very difficult to make change after finishing the program even though users can remove or add connection easily. More features will be added and allow users to easily remove or add components and connect them together.

Visibility

Visibility in Prograph is weak because of its deep subroutine. Both Java Studio and VisualAge for Java are moderate. The user can use scroll bar to check the program when building an application. Each bean in the design is at the same level.

For CSCK, there is no visible problem for a small program. Each compound component is defined in its own window. They can be compared side by side up to the limits of screen space. However, as a program becomes large, it may be very difficult to see every part of the code simultaneously.

As many components add or create in the environment, how to manage these components becomes a big issue. A component browser is needed for organizing and navigating components easily.

7.2 Testing

In addition to the cognitive evaluation of CSCK, a thorough testing for CSCK is required. Testing is the software process of critical importance, assuring that the software meets its requirements (verification process) and the requirements meet the needs of software users (validation process). For CSCK implementation, due to the lack of time and resource, we have not performed testing yet. In order to test CSCK in a meaningful way, CSCK requires further improvement and development. In this section, we discuss the testing process of CSCK.

The testing of CSCK should follow the normal software testing processes. It should involve two processes: verification and validation. The verification of CSCK is to verify the applicability of our general model for component-based software. We have used some practical examples to go through this process. We believe CSCK represents a sound proof of our general component model even though it needs further improvement.

The validation of CSCK involves intensive evaluation processes. Its purpose is to test whether this innovation meets the needs of target users and whether it increases software productivity. An independent test group is recommended to perform this process. At least two kinds of tests are required, usability testing and functionality testing.

The usability testing involves having the users work with the innovation and observing their response to it. It often focuses on the product's presentation rather than its functionality and in the end discovers how easy it is for users to bring up what they want.

The target users for CSCK are professional developers or novice programmers. This means these users are assumed to have basic background for programming. The intended use for the innovation will be for both developing "off-the-shelf" software components and developing application using these components. We shall invite these target users to go through the usability testing. The following usability characteristics should be tested (Kit, 1995):

- *Accessibility: can users enter, navigate, and exit with relative ease?*
- *Responsiveness: can users do what they want, when they want in a clear way?*
- *Efficiency: can user do what they want in a minimum amount of steps and time?*
- *Comprehensibility: do users understand the prototype structure easily?*

Functionality testing is a process of attempting to detect discrepancy between a program's functional specification and its actual behavior. We recommend use the black box testing method to test the functionality of the environment. The functional specifications for the environment should include a number of test cases which cover all aspects of the functionality of the prototype.

In addition to the usability testing and functionality testing, we also recommend to conduct other testing processes including performance testing, resource usage testing.

It is very important to note that we should test CSCK against the existing similar visual programming environments like VisualAge for Java, PART for Java, Prograph, and Java Studio. The same evaluation criteria and testing procedures should be applied to these environments to make comparison more sense. Based on the testing results, we can draw conclusions and find out the shortcomings for further improvement of CSCK.

7.3 Extending the Prototype

Besides the features mentioned in Section 7.1, other features that are needed to add or improve are:

1. The unimplemented parts shown in Table 6.1 should be completed.
2. It should allow users to generate JavaBean source codes or COM binary component based on the network diagram in the project editor.
3. It should allow users to import any component based on the component model like COM or JavaBeans. These components can be used in the construction of component-based programs.
4. An online help is needed which allows users to get help when necessary.
5. A debug facility should be integrated with the environment, allowing for detecting and correcting syntax and logical errors.

7.4 Comparisons with Other Aspects

In section 7.1, we have compared CSCK with visual programming environments in terms of cognitive dimensions framework. Now we compare them in other aspects.

Both Java Studio and Visual Age for Java are based on the JavaBean component model. The communication between components is through event delegation model. The internal works of a Javabean component consist of objects since JavaBeans is implemented by Java -- an object-oriented programming language. Prograph is based on a dataflow computational model. In the model, an operation executes when its data become available. Therefore the communication between operations is through data.

In CSCK, the communication between components can be considered a message with data via component ports. Whenever the trigger condition is met, the component executes. In the current implementation of CSCK, the internal works of a component is realized by using a Java method. However, it is very important to note that we can use any implementation to realize the internal works of a component like objects, procedure abstractions. To make whole implementation process of component-based software fully visual, we may use the visual programming language like Prograph to implement the internal works of a component. This is a very interest area to explore.

As we have presented in Section 7.1, CSCK is quite similar to Java Studio in terms of networking between components. However, CSCK is more powerful than Java Studio. A component can be initially created by CSCK. More importantly, CSCK offers an abstract mechanism to realize the program recursion. In the other hand, CSCK avoids the complex GUI as VisualAge for Java does while its functionality is still rich, provided that more features are added.

7.5 Summary

In this chapter, we have evaluated CSCK according to the Green and Petre’s framework, indicating that CSCK is good in many dimensions. During the discussion, we have also identified many features that need to add or improve for CSCK, as compared to Prograph, Java Studio, and VisualAge for Java. The testing process for CSCK has been discussed. At this stage, we can draw conclusions as follows.

- CSCK has proved the our general model for component-based software to be applicable;

- CSCK is able to help users familiarize themselves with our general model and component-based software development;
- CSCK exhibits good standings in term of the cognitive framework of Green and Petre although many features are needed to add.
- CSCK is implemented using Java Swing and Java in Window NT. As Java is a platform-independent language, it is easy to deploy the prototype to other platforms such as UNIX, Mac-OS.

CHAPTER EIGHT

Summary and Conclusions

This thesis presented a general model for component-based software and a prototype for the general model.

We have reviewed currently most successful component technologies, particularly three leading component models, CORBA, COM, and JavaBean. Their similarities and differences have been summarized and discussed. In order that developers are able to use component technologies properly and reduce the misinterpretations and implementation errors, a formal approach is necessary to precisely specify component models. The formal approaches for component-based software has therefore been investigated, which revealed that only COM component model has been formalized and there is a lack of a general model for component-based software. Based on the investigation of three component models, CORBA, COM, and JavaBean, we have proposed a general model for component-based software that consists of a series of definitions with a sound basis in mathematics. We believe that the general model captures the essences of most component models. It can help us familiarize ourselves with a component model, and learn the basic concepts of component and component-based software development.

A prototype for the general model, CSCK, has been designed and implemented in the thesis. CSCK has integrated the principle of visual programming, backed up by our overview of component-based software development with visual programming. It has well demonstrated the general model and proved that the general model is applicable. We have further evaluated CSCK according to the cognitive dimensions frameworks of Green and Petre (1996) while compared to the existing component-based visual programming environments like Java Studio, VisualAge for Java and Prograph. The testing procedure for CSCK has been discussed.

Further work is needed to extend the efforts we have made.

- Although our general model contains syntax and semantics, rules for proving theorems are required to enhance the general model;
- More features are required to add to the prototype. These features are described in Chapter 7.

References

Allen R. and Garlan D. (1997). A Formal Basis for Architectural Connection, ACM Transaction on Software Engineering and Methodology, Vol. 6, No. 3, pp. 213-249.

Andersen Consulting (1998). Understanding Component, http://www.ac.com/services/eagle/eagl_thought1.html.

Apple (1993). OpenDoc – Shaping Tomorrow's Software (white paper).

Barn B., Brown A.W. & Cheesman J. (1998). Methods and Tools for Component-Based Development, In: *Proceedings of Technology of Object-Oriented Languages*, pp. 385-395.

Bass L., Clements P., Kazman R.(1998). *Software Architecture in Practice*, Addison-Wesley.

Booch G. (1994). *Object-Oriented Analysis and Design, with Applications*, 2nd edn, The Benjamin/Cummins Publishing Company, Redwood City, CA.

Booch G. (1998). The Future of Software, Developing Component-based Systems, Presented at Component Directions '98 in Chicago.

Budd T. (1997). *An Introduction to Object-Oriented Programming*, 2nd edn, Addison-Wesley.

Burnett M., Baker M., Bohus C., Carlson P., Yang S., van Zee P. (1995). Scaling Up Visual Programming Languages, Computer, http://www.cs.orst.edu/~burnett/Scaling/ScalingUp.html.

Carrel-Billiard M & Akerley J. (1998). *Programming with VisualAge for Java,* IBM redbook.

Chappell D. (1997). The Next Wave: Component Software Enters the Mainstream, Rational Rose white paper, http://www.rational.com/uml/resources/whitepaers/dynamic.jtmpl?doc_key=354.

Coad P. (1999). *Java Modeling in Color with UML*, http://www.oi.com.

Coad P. and Yourdon E. (1991). *Object-Oriented Design*, Yourdon Press.

Cox P.T., Giles F.R., and Pietrzykowski T. (1989). Prograph: A Step toward Liberating the Programmer from Textual Conditioning, In: *Proceedings of the 1989 IEEE Workshop in Visual Languages*, pp. 150-156.

Cox P. T. and Pietryzkowsky T. (1990). Using a Pictorial Representation to Combine Dataflow and Object-Orientation in a Language-Independent Programming Mechanism, In Glinert, E. P., editor, *Visual Programming Environments: Paradigms and Systems*. IEEE Computer Society Press, Los Alamitos, CA.

D'Souza D. F. and Wills A.C. (1997). *Objects, Components, And Frameworks with UML – the Catalysis Approach*, Addison-Wesley, Reading, Mass. http://www.iconcomp.com/catalysis/index.html.

Drossopoulou S and Eisenbach S (1998). Towards an Operational Semantics and Proof of Type Soundness in Java, http://www-dse.doc.ic.ac.uk/projects/slurp/pubs.html#towards.

Englander R. (1997). *Developing Java Beans*, O'Reilly.

Flanagan D. (1997). *Java in a Nutshell*, 2nd edition, O'Reilly.

Gamma E., Helm R., Johnson R. and Vlissides J. (1994). *Design Patterns – Elements of Resuable Object-Oriented Software*, Addison-Wesley, Reading, MA.

Geary D. (1999). *Graphic Java 2 Mastering the JFC: Swing* (Sun Microsystems Press Java Series), Prentice Hall.

Geary D. and McClellan A. L. (1996). *Graphic Java: Mastering the AWT* (Sunsoft Press Java Series), Prentice Hall.

Ghezzi C., Jazayeri M. and Mandrioli D. (1991). *Fundamentals of Software Engineering*, Prentice Hall.

Goldberg A., Burrentt M.M., and Lewis T. (1995). What is Visual Object-Oriented Programming? M. Burnett, A. Goldberg & T. Lewis (Eds) *Visual Object-Oriented Programming: Concepts and Environments,* Manning Publications Co., Grrenwich, pp. 21-42

Green T.R.G., Petre M. (1996). Usability Analysis of Visual Programming Environment: A "Cognitive Dimensions" Framework, Journal of Visual Languages and Computing, v.7, no.2, pp.131-174.

Gregory D.A., Allen R. and Garlan D. (1995). Formalizing Style to Understand Descriptions of Software Architecture, ACM Transaction on Software Engineering and Methodology. Vol4, No 4, pp. 319-364

Harmon P. (1998). Component Development Strategies, Vol. VIII, No 7, http://www.cutter.com/itgroup.

IBM (1997). *VisualAge for Java – Getting Started for OS/2 and for Windows,* version 1.0.

Ibrahim R. and Szyperski C. (1998). Formalization of Component Object Model (COM) – The COMEL Language, In 8th PhD Wrokshop, ECOOP'98. http://www.fit.qut.edu.au/~ibrahim.

Jacobson I., Christerson M., Jonsson P. and Overgaard G. (1992). *Object-Oriented Software Engineering, A Use Case Driven Approach*, ACM Press, Addison-Wesley.

Jacobson I., Griss M., and Jonsson P. (1997). *Software Reuse, Architecture Process and Organization for Business Success*, ACM Press, Addison-Wesley Longman.

Kirtland M. (1997). The COM+ Programming Model Makes it Easy to Write Components in Any Language, Microsoft Systems Journal.

Kit, E. (1995). *Software Testing in the Real World: improving the process*, ACM press, Addison-Wesley.

Leach R. J. (1997). *Software Reuse - method, models, and costs*, McGraw-Hill.

Lewis T., Rosenstein L., Pree W., Weinand A., Gamma E., Caulder P., Andert G., Vlissides J. and Schmucker K. (1995). *Object Oriented Application Frameworks,* Manning Publications Co.

Liu X., Chen Z, Yang H., Zedan H., and Chu W.C. (1997). A Design Framework for System Re-engineering, In: *Proceedings of Asia Pacific and International Computer Science Conference*, pp. 324-352.

Logica UK Ltd. (1995). *Z Specific Formaliser User Guide* v7.3.

Meyer B. (1990). *Introduction to the Theory of Programming Language*, Prentice Hall.

Munch M. and Schurr A (1999), Leaving the Visual Language Ghetto, *Proceedings of IEEE Symposium on Visual Languages*.

Orfali R., Harkey D, and Edwards J. (1996). *The Essential Distributed Objects Survival Guide*. Wiley.

PARTS for Java, http://www.objectshare.com.

Pelegri-Llopart E. and Cable L.P.G. (1997). How to be a Good Bean, Sun white paper.

Prograph Internationals (1993). *Prograph CPX User Manuals*.

Rational Software Corporation (1999). Unified Modeling Language (UML), http://www.rational.com

Robinson M. and Vorobiev P. (1999). *Swing*, Manning Publications Co. (to be published) http://manning.spindoczine.com/sbe/

Rumbaugh J., Blaha M., Remerlani W., Eddy F. and Lorensen W. (1991). *Object-Oriented Modeling and Design,* Prentice Hall.

Schach S.R. (1997). *Software Enginnering with Java*, WCB/McGraw-Hill.

Schmidt D. Overview of CORBA, http://www.cs.wustl.edu/~schmidt/corba-overview.html.

Shaw M. and Garlan D. (1996). *Software Architecture - Perspectives on an Emerging Discipline*, Prentice Hall.

Sommerville I. (1992). *Software Engineering,* 4th edition, Addison-Wesley Publishing Company.

Sterling (1998). Modeling with Interface, http://www.sterling.com/content/white_papers_article.asp?id=37&pid=57&sid=1.

Sullivan K. J., Socha J., and Marchukov M. (1997). Using Formal Methods to Reason about Architectural Standards, In: *Proceedings of 19th International Conference on Software Engineering*, Boston, USA

Sun Microsystems (1997). JavaBeans for Java Studio: Architecture and API, White paper.

Sun Microsystems (1997). JavaBeans Specification, http://www.javasoft.com/beans/spec.html.

Sun Microsystems (1999a). Enterprise JavaBeans Specification, v1.1, http://java.sun.com/products/ejb/docs.html.

Sun Microsystems (1999b). *Java Look and Feel Design Guidelines*, http://java.sun.com/products/jlf/.

Szyperski C. (1996). Independently Extensible Systems – Software Engineering Potential and Challenges, In Proceedings of the 19th Australasian Computer Science Conference, Springer, Melbourne, Australia.

Szyperski C. (1998). *Component Software, Beyond Object-Oriented Programming,* ACM Press, Addison-Wesley.

Thomas A. (1997). Enterprise JavaBeans™ – Server Component Model for Java™, http://java.sun.com/products/ejb/white_paper.html.

Tran V., Liu D., and Hummel B. (1997). Component-based Systems Development: Challenges and Lessons Learned, In: *Proceedings of eighth IEEE International Workshop on incorporating Computer Aided Software Engineering*, pp. 452 - 462.

Weinschenk S., Jamar P., and Yeo S.C. (1997). *GUI Design Essentials*, John Wiley & Sons.

Wordsworth J.B. (1992). *Software Development with Z – A Practical Approach to Formal Methods in Software Engineering*, Addison-Wesley.

Yang Z. and Duddy K. (1996). CORBA: A Platform for Distributed Object Computing, ACM Operating Systems Review, 30 (2), pp. 4-31, http://www.omg.com.

APPENDIX A

Design and Implementation of CSCK

This appendix illustrates the design and implementation of the prototype of component-based visual programming environment, CSCK.

A.1 Class Diagram of the implementation

UML (unified modeling language) notation has been used for the documentation of class diagram for the implementation. UML is the standard modeling language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. UML modeling tool, Rational Rose by Rational Software Corporation (http://www.rational.com), has been used to draw all design diagrams.

A class diagram shows how the classes that implement a system are related. It provides a blueprint for implementation of the system. A package is a grouping of model elements, usually classes. Packages "own" the model elements they contain and may be nested with one another. In designing CSCK, we have used several packages to group classes with similar functionality together. As shown in Figure A.1, there are seven packages, namely, *views, editors, processing, system, interpreter, customizedComp, and Java Swing*. *MainProg and temp* are also included in the diagram for the purpose of showing the whole picture of the system even though they are not packages. *MainProg* is the main program for the system that invokes the primary window. *temp* is a directory used to temporarily store the Java bytecode generated during builder processing. Now we discuss each package in detail.

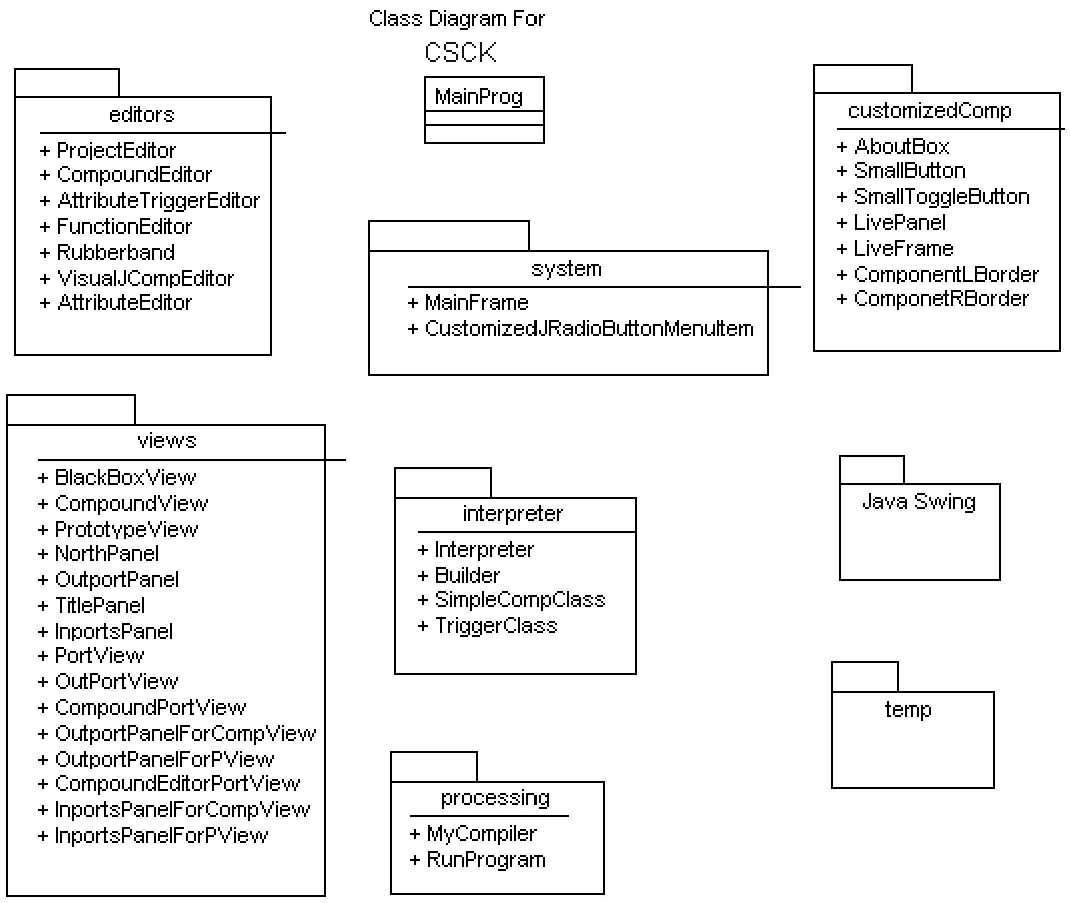


Figure A.1 Packages in Implementation of CSCK

The Package *editor* contains all classes that implement various editors as we discussed above. Class diagram for the package is shown in Figure A.2. All of these editor classes inherit from the Java Swing class *JinternalFrame*. *CompoundEditor* is a subclass of *ProjectEditor* which has *BlackBoxView* (simple component visual representation), *CompoundView*, and *PrototypeView*. *Rubberband* class is used to draw a rubber band when connecting components. Most of these classes consist of Java Swing classes like *JTextArea*, *JLable*, and *JTextField*.

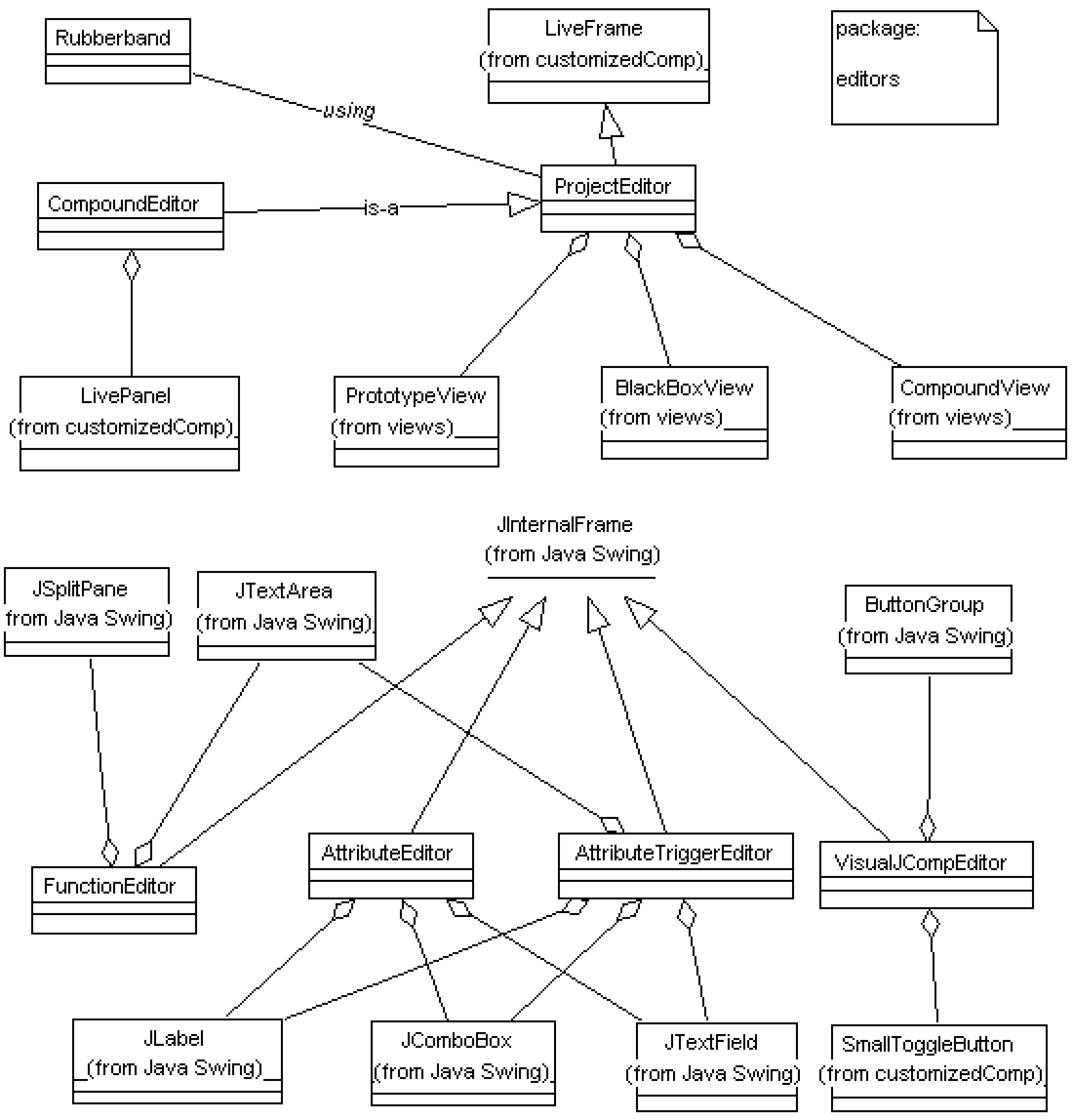


Figure A.2 Class Diagram in Package *editors*

Classes for the visual representations for all elements are grouped into package *views* shown in Figure A.3. Class *BlackBoxView* is for the visual representation of a simple component and also for a source or a sink component. Class *CompoundView* is for compound components, and Class *PrototypeView* is for a prototype component. Java Swing class *JPanel* is the superclass for all the view classes.

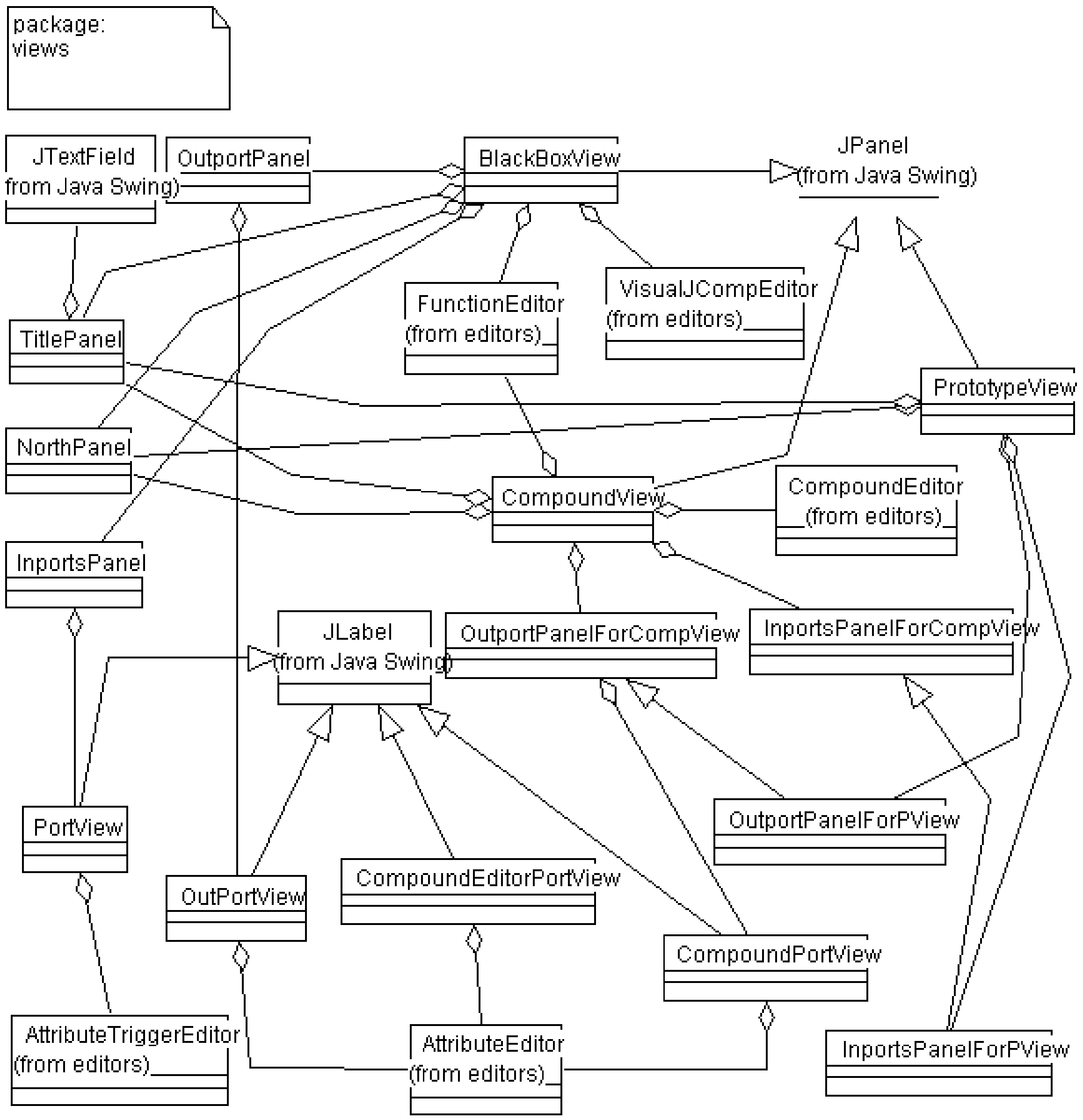


Figure A.3 Class Diagram in Package *views*

Package *system* mainly contains *MainFrame* class (Figure A.4). Class *MainFrame* is for designing the primary window. It inherits from *JFrame* and consists of *JMenuBar*, *JMenuItem*, *JDesktopPane*, *JPanel*, *ButtonGroup*, *CustomizedJRadioButtonMenuItem* (a subclass of *JRadioButtonMenuItem*), as well as *ProjectEditor*, *Interpreter*, and *AboutBox*.

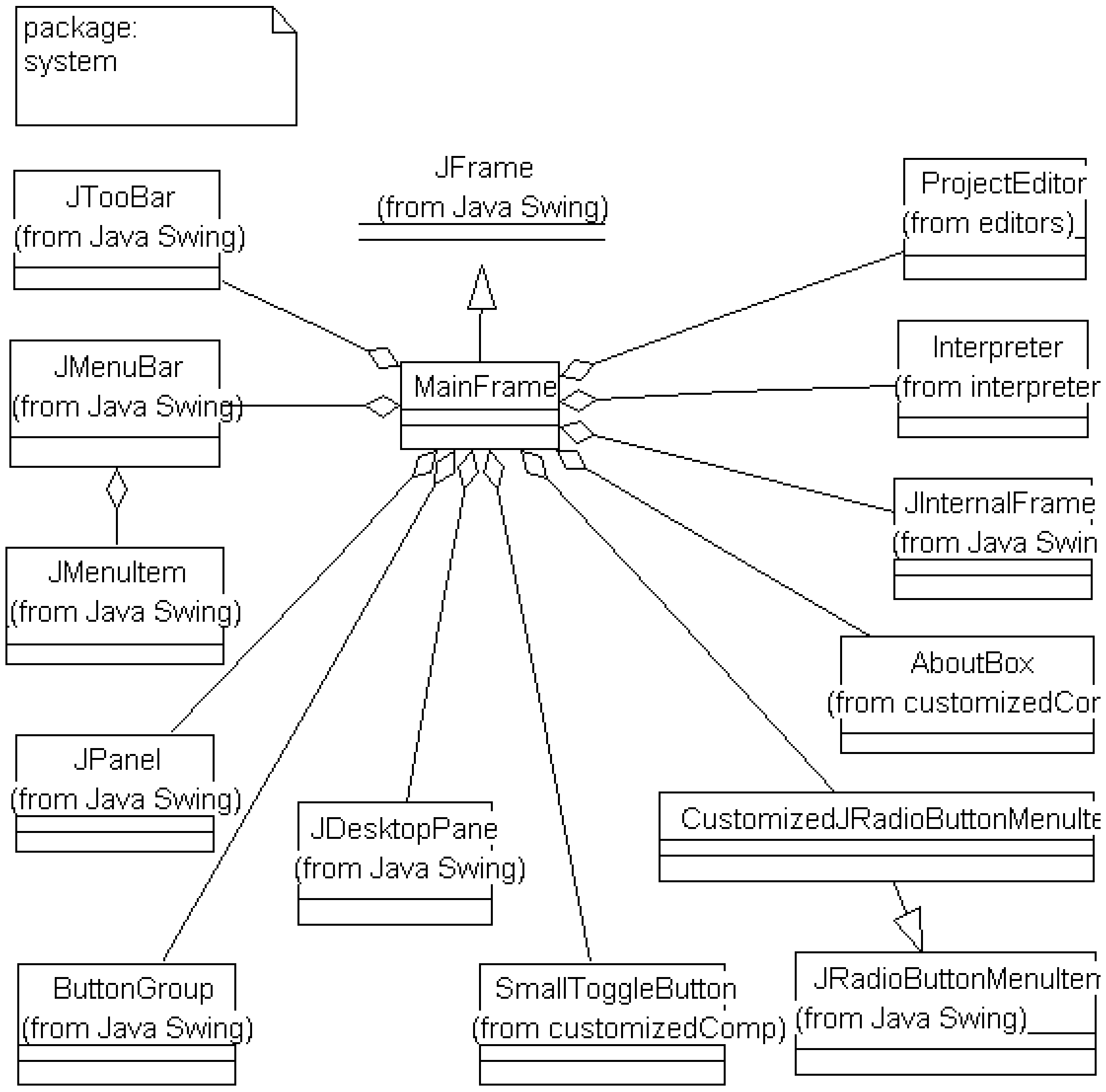

package: system
JFrame (from Java Swing)
JTooBar (from Java Swing)
ProjectEditor (from editors)
JMenuBar (from Java Swing)
MainFrame
Interpreter (from interpreter
JMenuItem (from Java Swing)
JInternalFrame (from Java Swin
AboutBox (from customizedCor
JPanel (from Java Swing)
JDesktopPane (from Java Swing)
CustomizedJRadioButtonMenuIte
ButtonGroup (from Java Swing)
SmallToggleButton (from customizedComp)
JRadioButtonMenuItem (from Java Swing)

Figure A.4 Class Diagram in Package *system*

The Package *interpreter* is shown in Figure A.5. Classes in this package are used to build and interpret the component-based program. Class *Builder* is used to generate Java source code using the function and trigger condition of a component and then compile it into Java byte code and store it in directory *temp*. The Class *Interpreter* will execute the program by calling the Java byte codes according to the semantics of the model. It is a subclass of *JinternalFrame* implemented with Java Interface *Runnable* in order to run this class as a separate thread. Classes *SimpleCompClass* and *TriggerClass* are abstract classes used only as a base from which other classes inherit. Java code generated by Class *Builder* for component function is a class which inherits from Class *SimpleCompClass*, whereas Java code generated by Class Builder for trigger condition is a class which inherits from Class *TriggerClass*. Doing so will provide a general way for Class *Interprete*r to execute Java byte code for function and trigger condition.

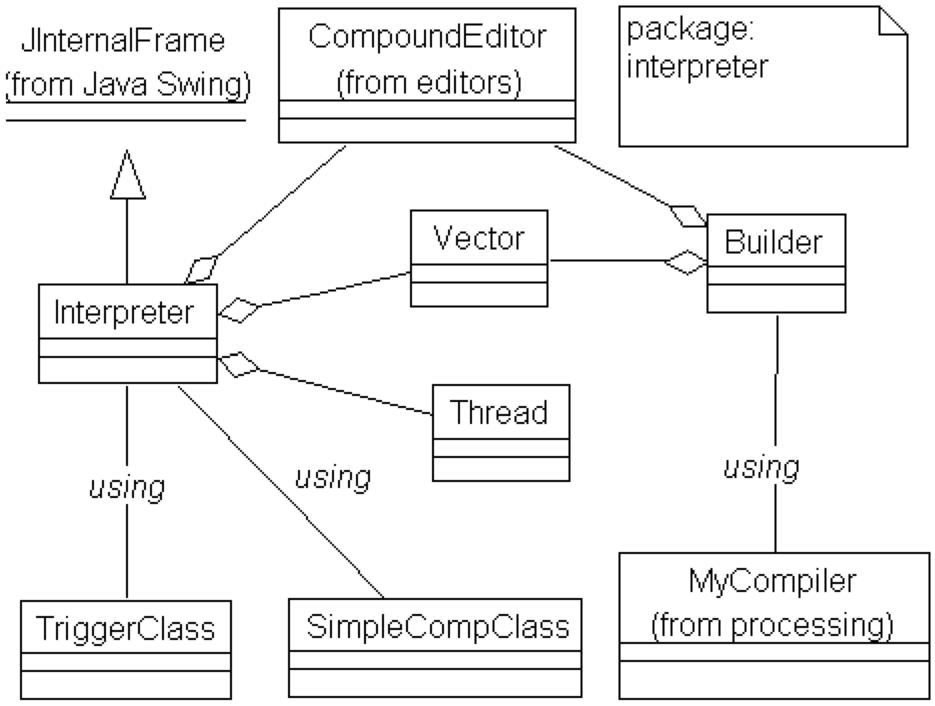


Figure A.5 Class Diagram in Package *interpreter*

Packages *processing* (Figure A.6) and *customizedComp* (Figure A.7) are two rather simple packages. Package *processing* provides class *MyCompile*r which is used to compile a Java source code into Java byte codes, and class *RunProgram* can run outside

Java class from a running Java program. Package *customizedComp* contains classes which customize some Java Swing class in order to conveniently use these Java Swing classes.

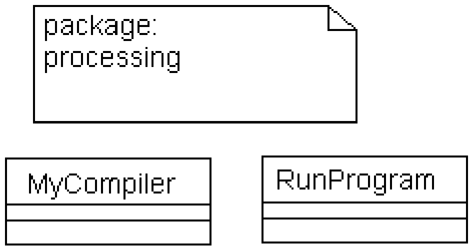


Figure A.6 Class Diagram in Package *processing*

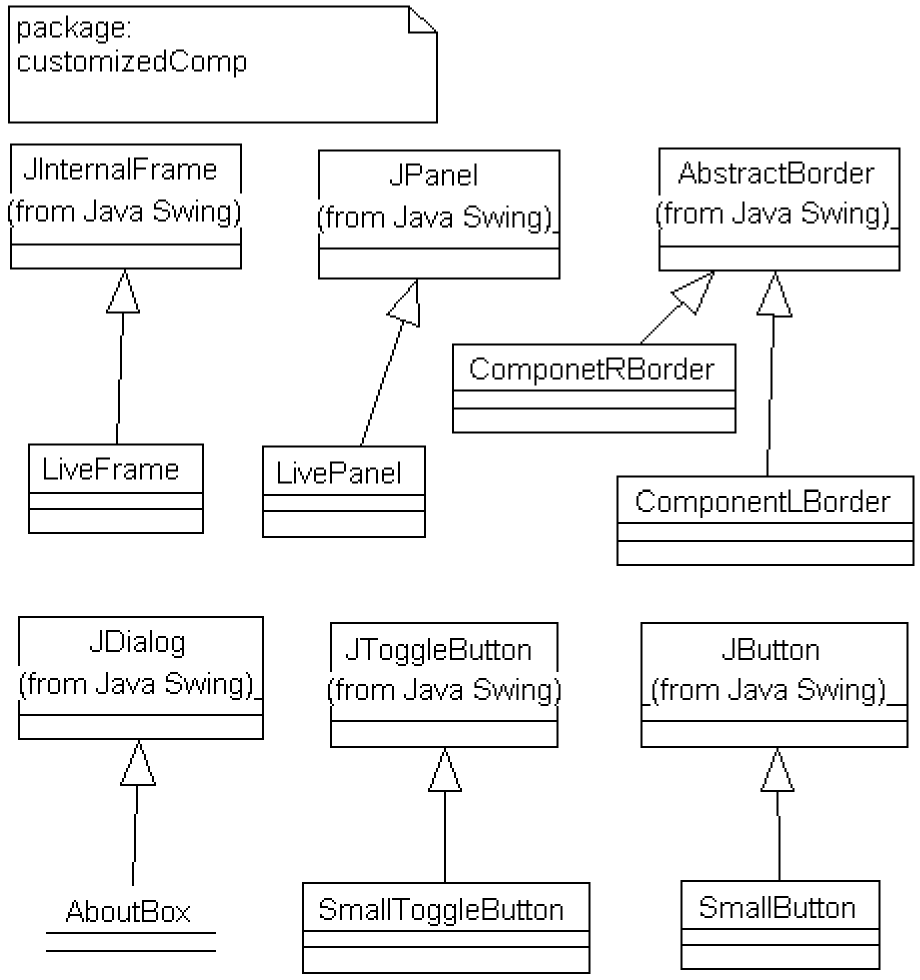


Figure A.7 Class Diagram in Package *customizedComp*

Since we use mouse to do most of the operations, most of the above classes have implemented Java Interface *MouseListener* and *MouseMotionListener* which handle actions such as mouse click, mouse drag, mouse press, mouse release. Interface *FocusListener* is also used in some classes to manipulate focus actions.

A.2 Implementation with Java

A.2.1 Java Look & Feel and Swing

Java look and feel is the default GUI for applications created by Java Foundation Classes (JFC). JFC includes Swing. Swing is a large set of components ranging from the very simple, such as labels, to the very complex, such as tables, trees, and styled text documents. These Swing classes provide good, high-level support for GUI development. Their appearance conforms to the standards of different platforms like Windows, Unix and Mac. Besides, all these Swing components have look and feel designs to specify.

Swing adopts the well–understood and highly advanced concepts from Java's predecessors in the object-oriented world. It can easily be integrated with Java implementation. The Swing basic concepts are the same as a regular object-oriented user interface toolkit. But Swing uses some advanced concepts like Model-view design pattern that separates the application's data from the data's display, to provide a flexible and solid foundation for creating novel GUI design (Robinson and Vorobiev, 1999).

Swing supports component-based software development. Swing classes are compatible with JavaBeans component model. All of the associated benefits of JavaBeans, for example, ease of use in IDEs, serialization support, and full support for event delegation model, can be gained from Swing components.

Given the above features of Java Swing, it is not surprising that Java Swing has been used to design and implement the GUI of the prototype.

A.2.2 Why Java?

We choose Java programming language to implement CSCK not only because Java Swing is so attractive for designing GUI as we discussed above, but because Java has offered important features (Flanagan, 1997) for the implementation of CSCK. First of all, Java provides a rich set of classes such as Java Swing, classes for data structures like Vector, Hashtable. Secondly, Java supports runtime compilation of code, making it easy to implement the semantics of our general model. Thirdly, Java is a simple language, as compared to other object-oriented programming languages such as C++. The most important is that Java uses automatic garbage collection to deal with memory management, and totally eliminates pointers. This would ease the development. Fourthly, Java is platform-independent. Because Java programs are compiled to a platform-independent byte code formats, a Java application can run on any system, as long as the system implements the Java Virtual Machine.

A.2.3 Major Data Structure

In the implementation, class *Vector* in Java is frequently used for keeping track of a number of objects. *Vector* implements an array of objects that grows in size as necessary. It is very convenient for manipulating a number of objects without knowing in advance how many there will be. Figure A.8 shows major data structure in the implementation. As seen, class *ProjectEditor* maintains a *Vector* which manipulates a number of objects. These objects have three kinds, *BlackBoxView*, *CompoundView*, and *PrototypeView*. The number of objects depends on how many components are added into the project editor window.

Each *BlackBoxView* maintains a *Vector* which keeps track of how many inports (represented in *InportView* object) are added into the simple component. Furthermore, each I*nportView* maintains a *Vector* which keeps track of a number of objects including *OutportView* and *CompoundPortView* that are connected to this inport, and at the same time, an *OutportView* for *BlackBoxView* also maintains a *Vector* to keep track of *InportView* or *CompoundPortView* that are connected to this outport.

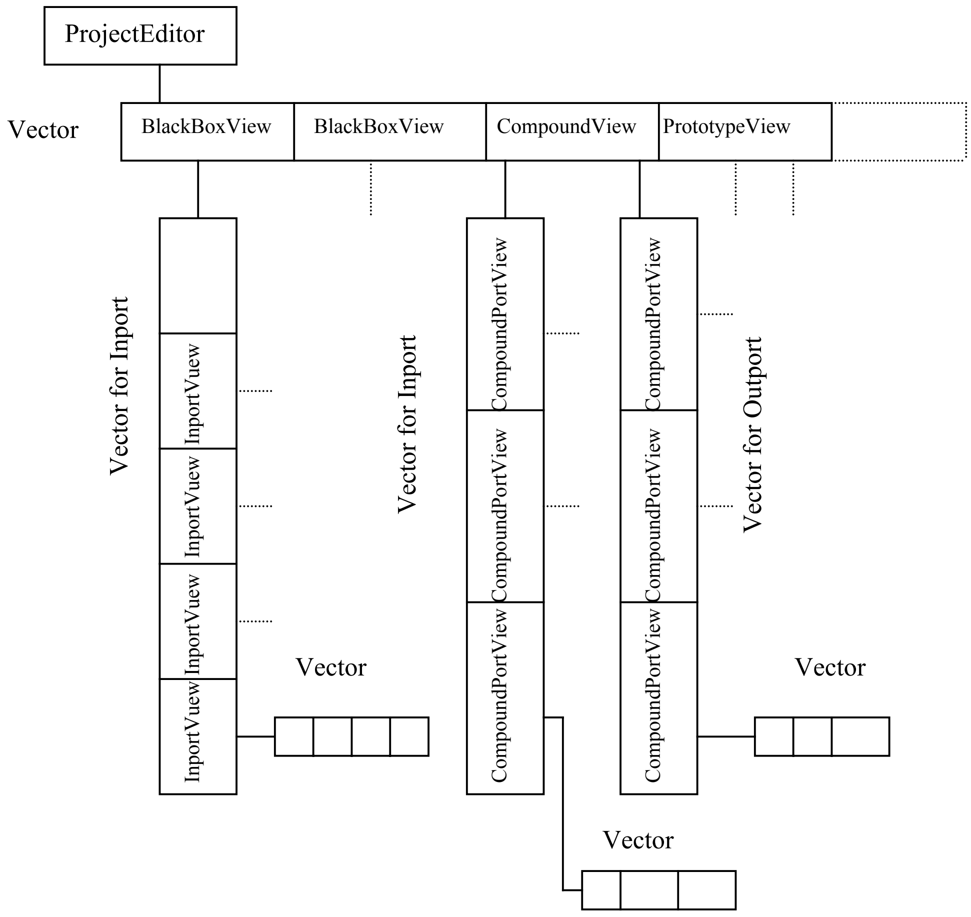


Figure A.8 Major Data Structure in the Implementation

Each *CompoundView* maintains two *Vector*s, one is used to keep track of inports, another is used to keep track of outports. Each inport or outport (implemented by *CompoundPortView*) also has a *Vector* which is used to record outports or inports that are connected to the inport or outport. The data structure for *PrototypeView* is the same as *CompoundView*.

A.2.3 Data Structure and Algorithms for Builder and Interpreter

Both function and trigger are of critical importance for execution of a component-based program. In our implementation, both are coded in Java and must therefore be compiled before execution. This requires two steps, First, Java source code is generated incorporating the text input by the programmer, then this code is compiled into Java byte code. This task is accomplished by Class *Builder* as discussed in the last section.

In order to provide a general way for the Class *interpreter* to execute the byte code generated by Class *Builder*, two abstract classes *SimpleCompClass* and *TriggerClass* are used. *SimpleCompClass* serves as superclass of a class (Java source code generated by the Class *Builder*) for each function of simple component and *TriggerClass* serves as a supuerclass of a class (Java source code generated by the Class *Builder*) for all trigger condition.

```
/**
 *  SimpleCompClass.java 1.0 6/26/99
 *  Copyright 1999 by Baoming Song
 *  This class serves as superclass for each component function
 */

package interpreter;

import views.*;
import editors.*;

public abstract class SimpleCompClass{

            public abstract void copyInfo(BlackBoxView bbv);
            public abstract Object function();

}
```

Figure A.9 Implementation of Class *SimpleCompClass*

In *SimpleCompClass*, there are two abstract methods, *copyInfo*, and *function*, as shown in Figure A.9. Since we do not know in advance how many inports a simple component has, the arguments for the function method have to be determined dynamically. To simplify the implementation of interpreter, in Java source code generated by the Class *Builder* (Figure A.10) we use method *copyInfo* to copy all the information about inports in the simple component visual representation into corresponding data fields in the class (Java source code generated by the class *Builder*). Then we can use method *function* (without arguments) to evaluate component function. The method *function* returns a Java Object. Sample source code generated by class *Builder* for a component function is shown as Figure A.10.

```
package temp;
import java.util.*;
import interpreter.*;
import views.*;

public class TempProjCC0C2 extends SimpleCompClass {

        protected int inport0;
        protected int inport1;

        public void copyInfo(BlackBoxView bbv){
                Vector inportsVector = bbv.getInports();
                Enumeration enum=inportsVector.elements();
                while(enum.hasMoreElements()){
                        PortView temp = (PortView)enum.nextElement();
                        String portName=temp.getPortName();
                        if(portName.equals("inport0"))
                                inport0=((Integer)(temp.getValue())).intValue();
                        if(portName.equals("inport1"))
                                inport1=((Integer)(temp.getValue())).intValue();
                }
        }

        public Object function(){
                return new Integer(inport0*inport1);
        }
}
```

Figure A.10 A Sample Java Source Codes for Component Function Generated by *Builder*

```
/**
 *  TriggerClass.java 1.0 6/30/99
 *  Copyright 1999 by Baoming Song
 *  This class will serve as a superclass for all trigger condition
 */

package interpreter;

import views.*;
import editors.*;

public abstract class TriggerClass{

        public abstract boolean condition(Object oldValue, Object newValue);

}
```

Figure A.11 Implementation of Class TriggerClass

Each Trigger condition is a binary relation. This means we can use two arguments to express trigger condition, one is the new value for the trigger, and the other is its old value. Since the number of arguments is fixed, triggers are easy to implement. In the Class *TriggerClass*, there is only one method, called *condition*. This method returns a boolean value (Figure A.11). A sample of the Java source code for a trigger generated by class *Builder* is shown in Figure A.12.

```
package temp;
import java.util.*;
import interpreter.*;
import views.*;

public class TempProjCC0C0inport0 extends TriggerClass {

        public boolean condition(Object oldV, Object newV){
                boolean flag = false;
                int x, y;
                if(oldV==null) x = 0;
                else
                        x =((Integer)oldV).intValue();
                        y =((Integer)newV).intValue();
                flag = (x!=y);
                return flag;
        }
}
```

Figure A.12 A Sample of Java Source Codes for Trigger Condition Generated by *Builder*

It should be noted that source code is generated only for simple components. This is because in our model compound component or prototype component can essentially be decomposed to simple components.

Now we discuss the algorithms used in the class *Interpreter*. The main algorithm used to interpret a program according to the semantics of the model is described in the form of pseudo code in Figure A.13. Recall that our general model has three processes to execute a program, propagation, evaluation, and expansion. In the implementation, we use three *Vectors* to manipulate and control these three processes. A while-loop is kept running. A *Thread* is forced to sleep for a while in order to allow the programmer to input data if necessary.

```
// Keep running a the following
 While (true) {
        // allows user to type input
        Try {
            Thread.sleep (100)
        } catch (InterruptedException e){}
        if (Vector 1 for components for evaluation is not empty)
                do evaluation (pseudo code in Figure A.14)
        if (Vector 2 for inports for propagation is not empty)
                do propagation (pseudo code in Figure A.15)
        if (Vector 3 for prototype for expansion is not empty)
                do expansion (pseudo code in Figure A.16)
}
```

Figure A.13 Pseudo Code for Algorithm used in *Interpreter*

Vector 1 is used to store a list of simple components ready for the evaluation process. Vector 2 is used to store a list of inports ready for the propagation process. Vector 3 is used to store a list of prototype components for the expansion process.

Figure A.14 shows a pseudo code for the evaluation process. For each simple component in Vector 1, we perform the evaluation process according to Definition 5.1.15. Note that the contents of Vector 2 or Vector 3 will be added before all elements in Vector 1 are removed. Figure A.15 shows a pseudo code for the propagation process according to Definition 5.1.13. During this process, simple components are added to Vector 1 if certain conditions are met. Figure A.16 shows a pseudo code for the expansion process according to Definition 5.1.14. Since the expansion rule of execution changes program structure by expanding prototype component, we need to make a copy of the structure. In order to avoid duplication during copying, Java Class *Hashtable* is used.

In order to start the execution, when the user inputs data into a source component, the inports that connect to this source component will add into Vector 2.

```
For each simple component in Vector 1 for components for evaluation
    Get its class name
    Call corresponding function Java byte code
    Set the result into outport for the component
    Get a list of the inports to which this outport connects
    If the component to which the inports belong is not prototype component
        Add the list of inports into Vector 2 for inports for propagation
    Else add the component into Vector 3 for prototype for expansion
End for loop
Remove all elements in the vector for components for evaluation
```

Figure A.14 Pseudo Code for Evaluation

```
For each inport in Vector 2 for inports for propagation
    If the inport is a trigger
        Call trigger Java byte code
        If the result is true
            Add the component to which this inport belongs into Vector 1
            for simple components for evaluation
    Set the inport a new value from outport it connects, and then set the outport null
    If the component to which the inport belongs is a visual bean (a sink component
    with no outport)  such as a text field
        show the value of inport in the visual bean
End for loop
Remove all elements in the vector for inports for propagation
```

Figure A.15 Pseudo Code for Propagation

```
For each prototype in Vector 3 for prototype for expansion
    Using a universal hashtable as reference, do copy structure
    Expansion the corresponding compound component
    Add the inport into Vector 1 for inports for propagation
End for loop
Remove all elements in the vector for prototype for expansion
```

Figure A.16 Pseudo Code for Expansion