JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 15, 585-614 (1999)

Towards a Practical Visual Object-Oriented Programming Environment: Desirable Functionalities and Their Implementation*

CHUNG-HUA HU+ AND FENG-JIAN WANG
Institute of Computer Science and Information Engineering
National Chiao Tung University
Hsinchu, Taiwan 300, R.O.C.
E-mail: fjwang@csie.nctu.edu.tw
+*Information Technology Laboratory*
Chunghwa Telecommunication Laboratories
Taoyuan Hsien, Taiwan 326, R.O.C.
E-mail: chhu@ms.chttl.com.tw

The ultimate goal of a programming environment is to help simplify the software development process. For an object-oriented language, a visual (object-oriented) programming environment (VOOPE) must at least satisfy four essential requirements to meet this goal: *interactivity, integration, incrementality,* and *intelligence*. In this study, object-oriented techniques were systematically applied to construct such a VOOPE. On the other hand, some characteristics of object-oriented languages, such as inheritance and polymorphism, may themselves be barriers to understanding, maintaining, and even constructing object-oriented programs. To solve, or at least alleviate, this problem, a language-based editing process has been designed and incorporated into our VOOPE. This process contains two key elements: syntax-directed editing facilities and an in-place editing assistant, which facilitate object-oriented program development by providing useful programming guidance and by reducing the number of potential programming errors. We have so far developed a window-based environment prototype using Visual C++ and the Microsoft Foundation Classes library.

Keywords: visual programming, integrated programming environment, object-oriented techniques, incremental program development, C++

1. INTRODUCTION

Visual programming [1-3] has become popular in recent years. The use of concise and appropriate graphics facilitates easier understanding compared to the use of plain text alone. Visual object-oriented programming [4] has progressed in two directions. In one direction, the visual syntax of an object-oriented language is defined, and programs are edited using visual language constructs. The visual syntax usually embodies graphical representations for such object-oriented language features as class constructs, inheritance, object instantiation, message passing (between objects), and polymorphism. With an ob-

Received July 16, 1997; accepted April 10, 1998.
Communicated by Y. S. Kuo.
*This research was supported in part by the National Science Council, under contract number NSC 87-2213-E009-002.

ject-oriented programming language containing such visual syntax, called a VOOPL, a constructed program is inherently visual in nature. Prograph [5] and VIPR [6] are two examples of VOOPLs.

In the other direction lies a *visual object-oriented programming environment* (VOOPE) for object-oriented program construction (and execution). One important issue in designing a VOOPE is the construction of a graphical user interface associated with a friendly user-interaction model. For example, a VOOPE may allow the user to open multiple windows for constructing and visualizing object-oriented programs, and may provide some interaction facilities (e.g., *point-and-click* or *drag-and-drop*) which enable the user to interact with the environment via pointing devices. SPE [7], MCPE [8], Visual C++ [9], and Smalltalk [10] are examples of VOOPEs.

The ultimate goal of a programming environment is to help simplify the software development process [11]. For a VOOPE, the following four design requirements are essential: *interactivity, integration, incrementality,* and *intelligence*. The main characteristics of the 4**I**'s requirements are as follows.

- **Interactive user interface** [12-14]
  A programming environment usually comprises a number of "front-end" tools, such as a program editor and a browser, that the user can interact directly with to construct and/or examine various kinds of program information. The window-based interface associated with the *direct-manipulation* [15] interaction model is one of the keys to a practical visual programming environment.

- **Integration mechanism supporting environment evolution** [16, 17]
  A programming environment is said to be "integrated" [8, 18, 19] only if its constituent tools can share consistent programming information and communicate with each other through a common platform. An effective *integration mechanism* allows the addition of a new tool or removal of an old one without large-scale modification, thus making environment evolution possible.

- **Incremental program development** [14, 19, 20]
  During programming, a number of analyzers, such as semantic and data-flow analyzers, and a code generator can be invoked incrementally. Incremental program development, in general, has the following two advantages. First, analysis and execution of incomplete programs are possible. Second, programming errors, such as syntactic and semantic errors, can be detected earlier than in conventional "edit-compile-debug" environments.

- **Language-level assistance on program construction**
  An "intelligent" programming environment [21-23] may provide *automatic programming capabilities* or at least *language-level assistance* to facilitate program editing and analysis. The language-level assistance employs programming information, such as syntactic and semantic information, to aid program construction by providing a certain amount of editing assistance and, at the same time, detecting potential programming errors or anomalies.

The contributions of our work, as shown in Fig. 1, are twofold. First, this paper presents an object-oriented approach that aids construction of a VOOPE based on the proposed *model-view-shape* (MVS) architecture [24, 25]. This architecture enforces a layered

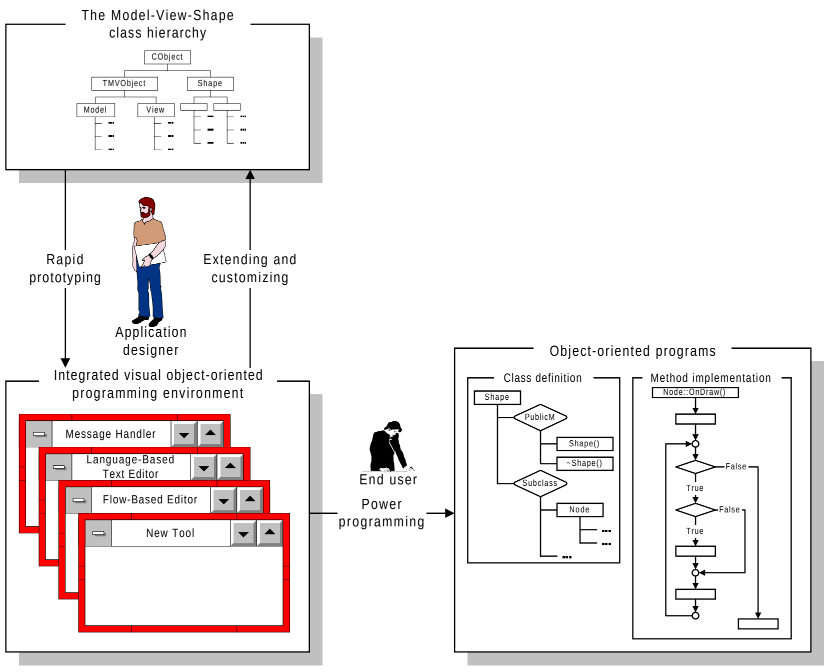


Fig. 1. Goals of our work.

and loosely-coupled structure, so that the user-interface part of a component can be more independent, maintainable, and reusable than those proposed in the original model-view-controller architecture. During the construction of our VOOPE, object-oriented techniques have been systematically applied to construct a C++ class hierarchy. Application designers can reuse and extend the class hierarchy so as to rapidly develop new tools for the VOOPE or visual programming environments that support different languages. To date, a window-based VOOPE prototype has been constructed using Visual C++ and the Microsoft Foundation Classes library [26].

Second, the VOOPE prototype provides well-designed editing, display, and analysis facilities that enable end users to effectively construct object-oriented programs. In other words, users are able to acquire some language-level assistance from the VOOPE during programming, so that they can reduce the number of programming errors (e.g., syntactic and semantic errors) or at least be informed of errors before compilation. This assistance is especially significant in object-oriented programming because some object-oriented language features, such as inheritance and polymorphism, are themselves barriers to understanding, maintaining, and even constructing programs [27, 28].

The rest of this paper is organized as follows. Section 2 gives the system architecture of a practical VOOPE. In section 3, a number of design and implementation aspects of our VOOPE are discussed. Section 4 describes how the user constructs object-oriented programs by interacting with the VOOPE. We discuss the main features supported by several VOOPEs in section 5 and draw the conclusions in section 6.

2. SYSTEM ARCHITECTURE

A practical VOOPE, whose system architecture is shown in Fig. 2, is proposed here to meet the above requirements. Tools widely used in a VOOPE are categorized into four toolsets according to the functions they perform: *programming*, *software-maintenance*, *software-reuse*, and *program-analysis*. The first three toolsets, which interact with users during various phases of software development, are usually equipped with (graphical) user-interface packages to display different kinds of program information. For example, the flow-based editor is used to display program flow information, such as control flow, data flow, or certain kinds of program-dependency relationships, in the graphical layout. By interacting with the flow-based editor, the user is able to specify and control the program structure at will. Another example, the message handler, is responsible for reporting programming errors and helping users locate erroneous program fragments. Tools in the first three toolsets are capable of receiving user-input events, interpreting and handling these events, and responding to users in some way.

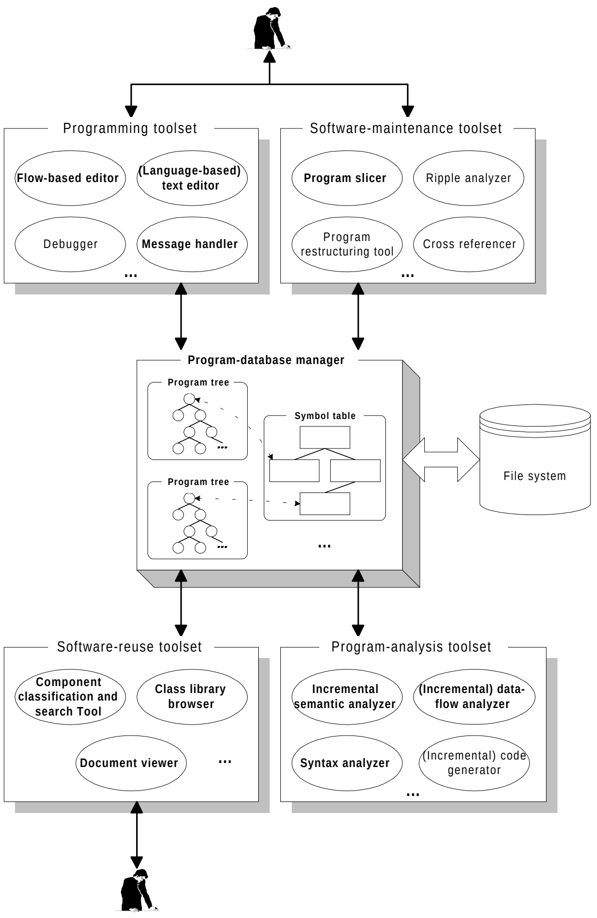


Fig. 2. The system architecture of a practical VOOPE.

The program-analysis toolset consists of tools for handling and analyzing program text during program development. These tools can be viewed as back-end tools and do not interact with users directly. They are activated only during changes in internal program representations, such as program trees and symbol tables. These tools can be further classified into two categories: *incremental* and *non-incremental* tools. Typical incremental tools include the semantic analyzer, the data-flow analyzer, and even the code generator. To ensure consistent and concurrent access to internal program representations, all the tools in the VOOPE are prevented from operating directly on the representations. The only way to access shared representations is through the program-database manager, which is used to coordinate tool communication and maintain data consistency by interacting with the underlying file system. A number of prototypical tools, whose names are shown in boldface in Fig. 2, have been constructed in our VOOPE.

3. CONSTRUCTING A VOOPE USING OBJECT-ORIENTED TECHNIQUES

3.1 A Class Hierarchy for Constructing the VOOPE Kernel

Fig. 3 shows a class hierarchy, based on the MVS architecture, for constructing the kernel of our VOOPE. The **Model, View**, and **Shape** class hierarchies correspond to model, view, and shape classes, respectively.

The **Shape** class hierarchy consists of two subclass hierarchies; one for node classes and the other for link classes. Common attributes and generic graphics-handling methods for these nodes and links are defined in classes **Node** and **Link**. Attributes defined in shape classes are used to store the graphical layouts and related information, such as the dimensions and coordinates of graphical components, while methods (defined in a shape class) can be divided into the following two sets:

- Graphics-handling methods. Examples include drawing graphical layouts at specific locations.
- Event-handling methods. Examples include detecting and interpreting user-input events.

The model, a representation of the application domain, contains attributes and operations for maintaining the application's state and behavior. Attributes defined in model classes are generally divided into two sets: one for the maintenance of internal program representations and the other for the storage of language-dependent information, such as source code, comments, and static semantics. Methods defined in model classes are generally used to perform syntactic and semantic analyses as well as language-dependent functions. The view, which is used to handle the user interface, contains attributes and operations for managing the application's display and input-event interactions. Attributes defined in view classes are used to store high-level presentation information, such as view dimensions, while methods (defined in view classes) can be divided into the following two sets:

- View-management methods. Examples include calculating and retrieving view dimensions.
- View-presentation methods. An example is presenting view layouts.

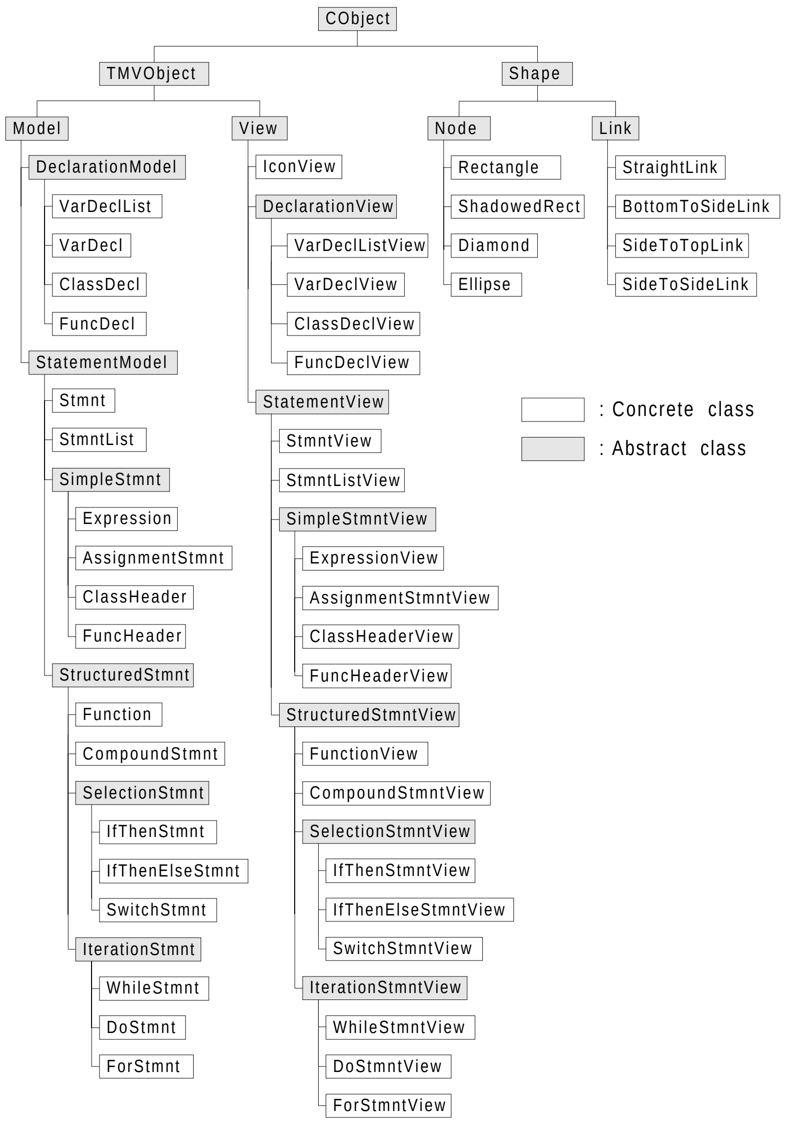


Fig. 3. The MVS class hierarchy.

3.2 Graphical Representations for Object-Oriented Language Constructs

To help users depict flow information about object-oriented programs visually and rapidly, it is desirable for the VOOPE to provide graphical representations, i.e., *graphical templates*, for high-level language constructs. A set of graphical templates, as shown in Fig. 4, has been designed to represent well-known language constructs using the syntax employed in the C++ language subset. These graphical templates are used to represent the static language features of object-oriented programs, such as class constructs, inheritance, and structured statements. Object-oriented dynamic structures, such as object instantiation and polymorphism, are not currently supported because these structures require that information be provided by a runtime visualization tool [29, 30].

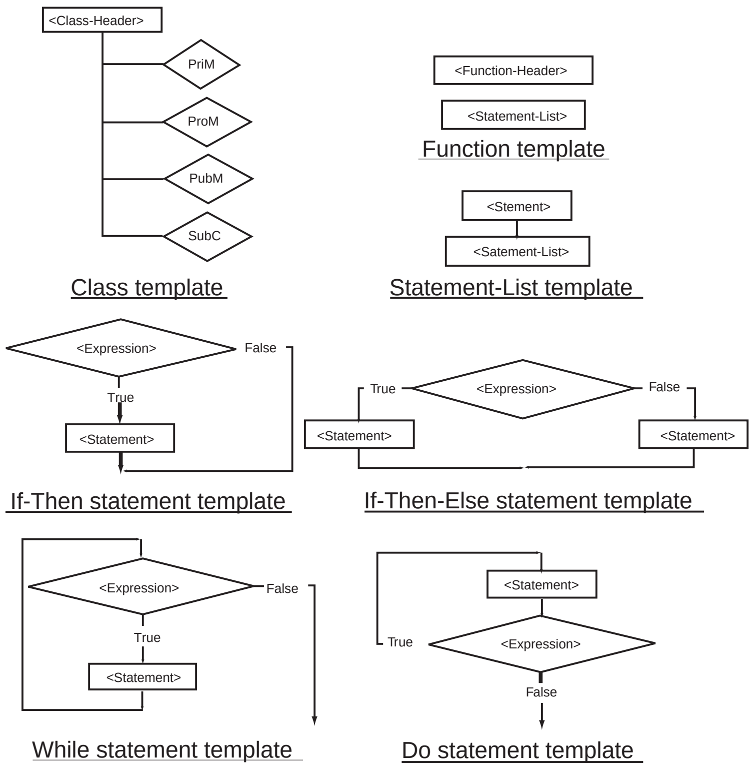


Fig. 4. Sample graphical templates for language constructs.

Three levels of access mechanisms, *private, protected*, and *public*, classify attributes and methods defined in (C++) classes into three corresponding groups. Such a class construct can be supported by a class template, as shown in Fig. 4, containing four diamond nodes, where the first three diamond nodes, called "**PriM**", "**ProM**", and "**PubM**", are used to "hook" three groups (i.e., private, protected, and public) of methods, and the last diamond node, called "**SubC**", hooks a list of class(es) derived from the class. Note that this class template is used to represent tree-based class structures, i.e., single-inheritance structures. Multiple-inheritance structures entail graph-based class structures. Fig. 5 shows a graphical representation of multiple inheritance using four class templates.

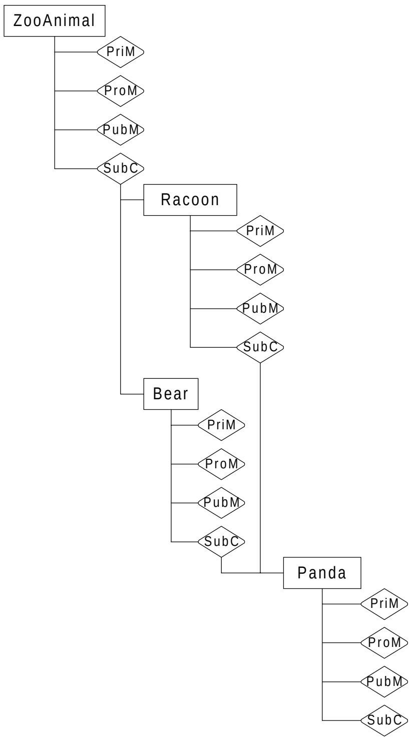


Fig. 5. A graphical representation of multiple inheritance.

3.3 Manipulating Internal Object-Oriented Program Representations

As users construct object-oriented programs, our VOOPE employs two data structures, *symbol tables* and *program trees*, to store these object-oriented programs internally and incrementally. The symbol table, a hierarchical tree structure corresponding to the scope rules of object-oriented languages, is used to store the scope and binding information of identifiers. The program tree, an internal representation of the method, is used to store language-dependent information, such as source code, comments, and other semantic attributes in the corresponding tree nodes. These two data structures constitute the shared programming information for all the tools in the VOOPE.

Maintaining internal program representations consistently is one important issue in an integrated programming environment [16]. To allow different tools to have consistent views of shared data structures, the program-database manager in our VOOPE performs three functions. First, it manages internal program representations; that is, it is responsible for constructing and maintaining symbol tables and program trees. Second, it can be viewed as a *message server* for tool communication and coordination. Through the *service routines* provided by the program-database manager, the tools in the VOOPE are able to communicate with each other and access the internal program representations concurrently and consistently. Third, it serves as an intermediator for storing and retrieving object-oriented programs by interacting with the underlying file system. In the following, we describe the three tasks supported by the program-database manager in more detail.

3.3.1 A manager of internal program representations

Analysis of program scope aids construction of feasible symbol tables. Here, at least three kinds of program scope must be provided in an object-oriented language: the *file scope, class scope*, and *local scope*. The file scope, within which global variables are declared, is the outermost scope of a program. Each class in an object-oriented program has a distinct class scope, in which method and attribute names are considered inside of the class scope. Within a method, each compound statement (i.e., block) has an associated local scope. The arguments of a method can be viewed as being within the local scope of the outermost block of the method. To simplify discussion, the file scope is omitted here because the use of global variables violates the principle of information-hiding of object-oriented programs. Moreover, nested blocks may introduce a nested local scope, which is not currently supported by our symbol table.

```
Class Node: public Shape {
Public:
  .....
  Node(CString contentText);
  virtual void OnDraw(CDC* pDC);
};
class Link: public Shape {
  .....
};
class Rectangle: public Node {
  ......
};
class Diamond: public Node {
  ......
};
```

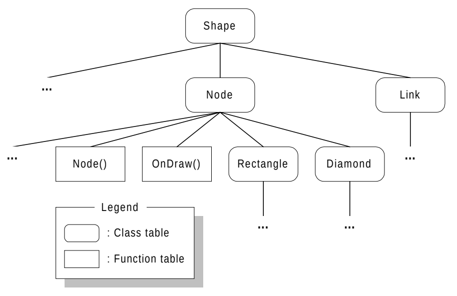


Fig. 6. Class interfaces and the associated symbol table.

The symbol table, as shown on the right side of Fig. 6, comprises two kinds of table objects, *class-table* and *function-table* objects, used to record the class and local scope information. When the user defines a class, the program-database manager creates a class table and attaches it to the symbol-table hierarchy. For a method defined in a class, the program-database manager creates a distinct function table and attaches it to the class's class table. In our VOOPE, class and function tables are instances of classes **ClassTable** and **FuncTable**, respectively. Table 1 lists some of the interfaces of classes **ClassTable** and **FuncTable**.

Each class table maintains two attributes, called **BaseClassTable** and **DerivedClassTableList**, to reference the class tables of base and derived classes, respectively. These two attributes help the symbol table record inheritance relationships among classes. Another attribute, called **FuncTableList**, is used to reference the function tables for the methods defined in the class. Moreover, a class table also stores declaration information of attributes and/or methods defined in the class. Each function table main-

Table 1. Class interfaces for supporting the construction of symbol tables (partial).

```
class FuncDecl: public DeclarationModel {
public:
  CString Return Type, ClassName, FuncName,
      AccessType;
    //The access type of a function can be either
    //"Private", "Protectect", "Public", or "Global"
     CStringList ArgumentTypeList;
  ......
};
class VarDecl: public DeclarationModel {
public:
  CString VarType, className, VarName,
      AccessType;
        //The access type of variable can be either
        //"Local variable",
        //"Parameter", "Private", "Protected", or
        //"Public"
  FuncDecl *pfuncSignature;
  ...
};
Class ProgramDatabaseManager: public CObject {
public:
  CObList SymbolTable;
  //contains a list of ClassTable objects
  CObList Program Tree:
  //contains a list of  Function objects
  ...
};
```

```
class ClassTable: public Cobject{
public:
ClassTable *pBaseClassTable;
CObList DerivedClassTableList;
            //contains a list of ClassTable objects
CObList FuncTableList:
            //contains a list of FuncTable objects
CString ClassName;
CObList PrivateAttributeLiast,
    ProtectedAttributeList,
    ProtectedAttributeList;
            //contains a list of VarDecl object
  CObList PrivateMethodList,
     ProtectedMethodList, PublicMethodList;
            //contains a list of funcDecl objects
  ....
};
class FuncTable: public CObject {
public:
  ClassTable *pClassTable;
  FuncDecl *pFuncSignature;
  CObList ArgumentList, LocalVariableList;
            //contains a list of VarDecl objects
  ....
};
```

tains two attributes, called **ArgumentList** and **LocalVariableList**, to store declaration information of arguments and local variables (defined in a method), respectively. Note that methods with the same names may be defined in a class; in this case, the method is said to be *overloaded* [31]. An attribute called **FuncSignature**, which stores the method's *signature*, including the method name, the return type, and the type information of arguments, can be used to resolve this kind of ambiguity.

When the user edits a method, a corresponding program tree, an internal representation of the method, is created and maintained by the program-database manager. Fig. 7 shows a sample program tree representing an if-then-else statement, and illustrates the association relationships among model, view, and shape objects. The construction details of program trees can be found in [24].

3.3.2 A message server for tool communication and coordination

Meyers listed a number of integration mechanisms applicable to modern software development environments [16]. Typical integration mechanisms include *shared file systems*, *selective broadcasting*, *simple databases*, *view-oriented databases*, and *canonical repre-*

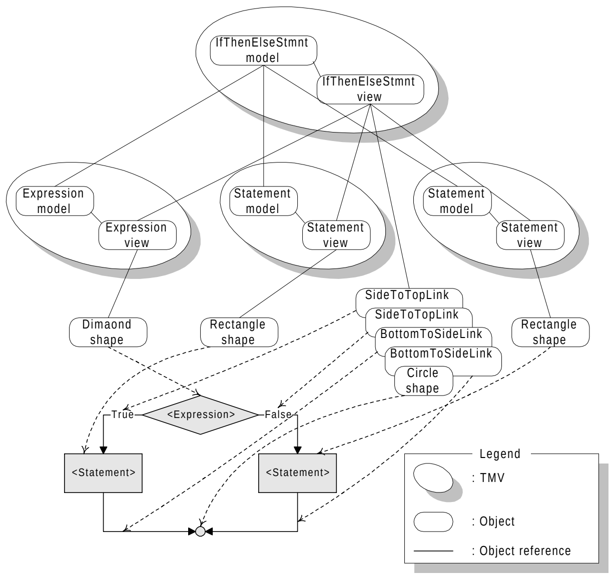


Fig. 7. Relationships among model, view, and shape objects.

sentations (i.e., shared data structures). In such an environment, a constituent integration mechanism is used for 1) communications between tools and 2) addition of new tools. Our program-database manager, which provides the integration mechanism, is intended to provide infrastructure support for *data*, *presentation*, and *control integration* [32] among all the tools in the VOOPE.

Data integration can be achieved easily because all tools access the internal program representations through the program-database manager. Presentation integration means that all front-end tools, which need to interact with users, have common and uniform user interfaces. Presentation integration can also be achieved easily in our VOOPE because the front-end tools use the same class hierarchy of view and shape objects to handle editing and display tasks. Control integration concerns mechanisms that enable one tool to activate other tools. In our VOOPE, the communication channels among the tools and the program-database manager are typically *message-driven*, like the one proposed in [17], and the program-database manager can be viewed as the message server. Therefore, a tool may be activated when it receives a message from another tool through the program-database manager.

3.3.3 An intermediator for storing and retrieving object-oriented programs

When the user edits object-oriented programs which contain a number of program units, such as classes and methods, the programming activities most concerned are: 1) what program units need be constructed, and 2) how they should be constructed. Storage management of these respective program units when interacting with the underlying file system may be an extra burden for users. This work can be made semi-automatic with the help of the program-database manager. To make the VOOPE act like a “fileless” environment [23], the program-database manager maintains a *program table* to record related information of program units, such as 1) types and names of program units, 2) the project name these program units belong to, and 3) their respective storage locations. The association relationship between a file and a program unit is generally one-to-many; i.e., a file may contain more than one program unit.

To modify existing program units, all a user has to do is issue an editing command, called “opening a program unit” with the unit name as a parameter, to the program-database manager. Upon receiving the command, the program-database manager 1) opens the appropriate file and locates the starting position where the program unit is stored; 2) loads the context, including the source code associated with comments, of the program unit into working memory; 3) transforms the context into the corresponding internal program representation(s); and 4) invokes the programming tools to display the visual and/or textual layouts of the internal program representations. Even if methods have the same names (i.e., are overloaded), the program-database manager can resolve the duplication by listing possible candidates for the user.

3.4 Tool Construction with the MVS Class Hierarchy

3.4.1 Design rationales

The MVS class hierarchy presented in this paper is basically an application framework for tool construction through compositional reuse of software components, especially graphical user interfaces. Our construction approach is to incorporate the tool’s functionality into the programming environment by extending the MVS class hierarchy. When a tool is to be introduced, the designer needs to study what functions the new tool will perform and then examine the MVS class hierarchy to locate the attributes and methods in the corresponding classes that can be reused as well as the new class(es) and associated attributes/methods that should be added. Reusing existing functions, such as those for the traversal of program trees, can help reduce the effort needed to construct a new tool.

Tools in our VOOPE can be generally classified as front-end and back-end tools. For those front-end tools that need to provide a variety of graphics-drawing facilities, shape classes are reusable because they are application-independent while model and view classes may be augmented with new functionalities. Moreover, front-end tools should maintain consistency among multiple views [7, 16, 33]. Table 2 lists a number of methods for en-

Table 2. Class interfaces to support tool communication and coordination (partial).

```
class ProgramDatabaseManager: public CObject {
public:
  CObList RegisteredToolList;
  ...
  int Register(CObject *pTool);
  int Deregister(CObject *pTool);
  int ChangeFrom(CObject *pFromTool,...);
};
```

```
class FlowBasedEditor, LanguageBased TextEditor,
MessageHandler : public CScrollView {
public:
  int UpdateFrom(CObject *pFrom Tool,...);
  ...
};
```

abling various front-end tools to coordinate their actions. Method **Register( )** is used to add tool-registration records to attribute **RegisteredToolList** while **Deregister( )** removes them. When a tool finishes modifying the internal program representations, it sends message **ChangeFrom( )**, which contains the modified data as parameters, to the program-database manager. Upon receiving message **ChangeFrom( )**, the program-database manager broadcasts message **UpdateFrom( )** to the tools registered in attribute **ToolList** to retrieve the modified data for further processing.

Back-end tools, which do not interact with users directly, are activated by changes in internal program representations or by commands from users. A back-end tool, such as a semantic or data-flow analyzer, is constructed by 1) adding *semantic attributes* and *evaluation methods,* which are used to implement the tool's functionality, to the model classes in the MVS class hierarchy, and 2) registering with the program-database manager using a specific evaluation method, called an *activation method*, that will be triggered (by the manager) to activate the tool's services. Each of our back-end tools, in general, executes with the *node-evaluating* and *-marking* process [34] operating on the program tree, while the process is excuted via message-passing between model objects in the program tree. When a model object receives a message, it evaluates the values of related semantic attributes and then sends messages to its parent and/or child model objects if needed. During the evaluation process, those language constructs, evaluated as outcomes, are indicated by marking the corresponding model objects. Moreover, the user interfaces of new back-end tools do not need to be constructed from scratch because existing view and shape objects, supported in the MVS class hierarchy, can be used to display the analysis results.

The design and implementation details of most tools in our VOOPE, such as the language-based text editor, message handler, incremental semantic analyzer, and data-flow analyzer, can be found in [24, 35-37]. In the following, we give a short tool-construction example to illustrate how a program slicer can be incorporated into our VOOPE.

3.4.2 A program slicer and its implementation

Program slicing, an automatic technique used to determine the statements which may potentially affect or be affected by a variable at a given statement, aids program understanding by reducing the code a programmer must examine, and by presenting only a relevant program subset of interest. Computation of program slices involves examining both the data-flow and control-flow dependencies of a program. One typical approach [38, 39] to computing program slices is to summarize and symbolize control-flow and data-flow dependencies as edges of a directed graph, called a *program dependence graph* [40], in which the vertices are the statements of a program. In this approach, a forward or backward slice is computed by identifying the set of statements in the slice through forward or backward transitive closure in this graph.

The most intuitive yet effective way to construct a program slicer is to reuse existing data-flow analysis facilities, i.e., the attributes and methods of model classes, to compute definition-use (DU) and use-definition (UD) chains [41], and to incorporate control-flow analysis facilities into the slicer. The construction cost, compared with the effort needed for the building-from-scratch approach, is reasonably low because the tool designer only needs to concentrate on how to reuse the existing code and add some new functionalities to the MVS class hierarchy. Another advantage is that our program slicer can work directly on the program tree without the need to create and maintain redundant data structures, such as program dependence graphs.

In our approach, a forward (or backward) slice is computed by reusing the functionality of the DU (or UD) analysis algorithm [35]) as a means of transitive closure. Specifically, the functionality of an intraprocedural (i.e., intra-method) program slicer is systematically handled using the following evaluation methods specified in the respective model classes, as shown in Table 3: **ComputeForwardSlice(), ComputeBackwardSlice()**, **GetBranchExpressionsBackwardUp( )**, and **GetBranchExpressionsBackward-Down( )**. The first method is responsible for computing a forward slice with respect to a variable defined, and the rest are responsible for computing a backward slice with respect to a variable used. The term "forward" (or "backward") shown in the method' names denotes that the computation sequence will basically follow (or reverse) the control flow of a program. **ComputeForwardSlice( )** and **ComputeBackwardSlice( )** serve as two activation methods for initiating forward and backward slicing, respectively. Methods **GetBranchExpressionsBackwardUp( )** and **GetBranchExpressionsBackwardDown( )** track and mark those expressions that may potentially affect the execution of a given statement being sliced.

Table 3. Model class interfaces for computing intraprocedural program slices (partial).

```
class Expression: public SimpleStmnt {
public:
   StringList UsedVariables;        // reused attribute

   void ComputeUDChain (...);     // reused method
   ....
   void ComputeBackwardSlice(String variableName,
                                    ModelList *pMarkedModels);
   void GetBranchExpressionsBackwardUp
                                    (StatementModel *pFrom,
   void GetBranchExpressionsBackwardUp
                                    (StatementModel *pFrom,
                                    ModelList *pMarkedModels);
};
```

```
class AssignmentStmnt: public SimpleStmnt {
public:
  StringList DefinedVariable, UsedVariables;
    //reused attributes
  void ComputeDUChain(...); // reused method
  void ComputeUDChain(...); //reused method
  .....
  void ComputeBackwardSlice(...);
  void ComputerForwardSlice(...);
  void ComputerForwardSlice(...);
  void GetBranchExpressionsBackwardUp(...);
};
class StmntList, IfThenElseStmnt, WhileStmnt....{
public:
  ......
  void GetBranchExpressionsBackwardUp(....);
  void GetBranchExpressionsBackwardDown(...);
  };
```

Table 4 shows a portion of the intraprocedural forward and backward slicing algorithms. The forward slicing algorithm, in brief, invokes **ComputeDUChain( )** transitively to facilitate the forward slicing process. Likewise, the backward slicing algorithm is highly associated with the functionality of the UD analysis algorithm. Fig. 8 shows an example of computing a backward slice with respect to variable a after the user issues a "show backward slice" command in the assignment statement "**f = a**". Fig. 9 shows such a message-passing flow based on the computation of UD chains.

Table 4. Intraprocedural forward and backward slicing algorithms (partial).

```
Function AssignmentStmnt: ComputeForwardSlice
(variable Name, pMarked Models)
declare
variableName (IN variable): the name of a variable that
                              is to be sliced
pMarkedModels(OUT variable): a list of model objects
                         constituting a forward slice
pModel: a pointer to a SimpleStnmt object
varName: a variable's name
begin
  ComputeDUChain(variableName, this, pMarkedModels)
/*Initiate a DU analysis w.r.t. variable 'variableName'.
  After ComputeDUChain() completes ececution,
  pMarkedModels will collect a list of model objects
  constituting a DU chain.*/
  for each pModel pMarkedModels do
    If pModel→ObjectType="AssignmentStmnt" or then
      for each varName ∈ pModel→DefinedVariables do
        pModel→ComputeDUChain(varName, this,
        pMarkedModels)
      od
    fi
  od
end
```

```
FunctionExpression: ComputeBackwardSlice
(variableName, pMarkedModels)
declare
variableName(IN variable): the name of a variable that
                              is to be sliced
pMarkedMOdels (OUT variable): a list of model objects
                         constituting a backward slice
pModel: a pointer to a SimpleStmnt object
varName: a variable's name
begin
  ComputeUDChain(variableName, this,
pMarkedModels)
  for each pModel ∈ pMarkedModels do
    for each varName ∈ pModel→DefinedVariables do
      pModel→ComputeUDChain(varName, this,
      pMarkedModels)
    od
    pModel→GetBranchExpressionsBackwardUp
    (NULL, pMarkedModels)
  od
end
```

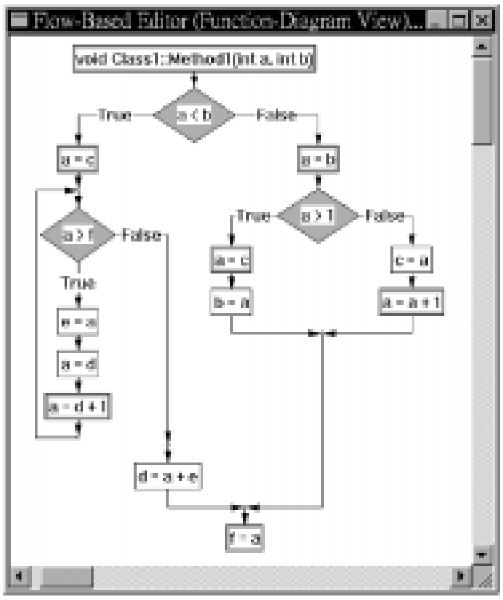


Fig. 8. A backward slice w.r.t. variable **a** in "**f = a**".

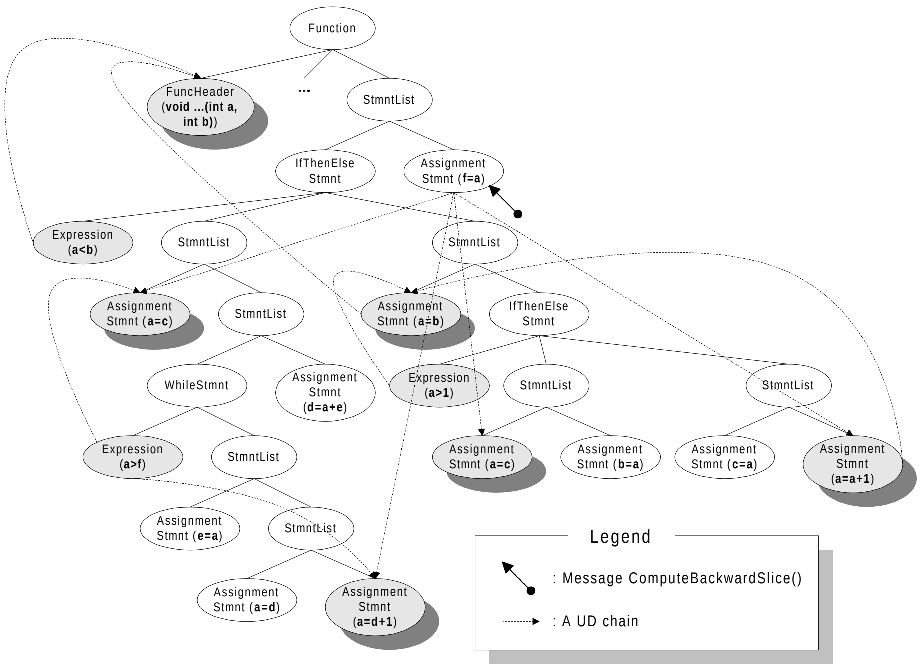


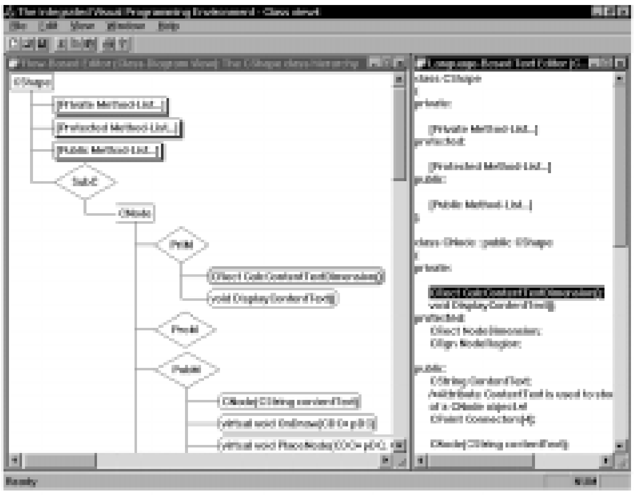
Fig. 9. Computing a backward slice for Fig. 8.

4. PROGRAMMING ACTIVITIES IN OUR VOOPE

4.1 A Two-Phase Model for Object-Oriented Program Construction

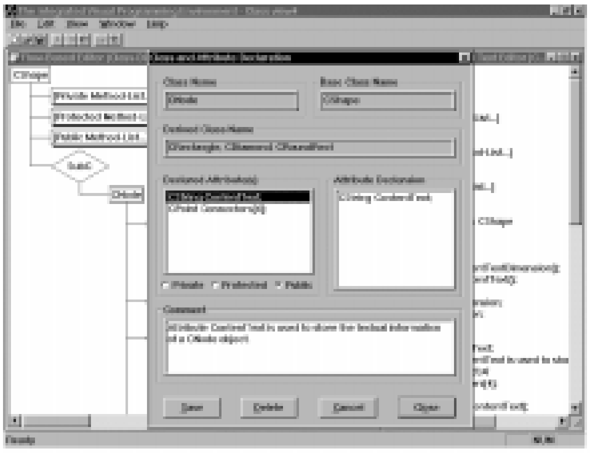
In general, writing an object-oriented program consists of the following two phases [42]:

1. **Identifying the classes and defining their interfaces.**
   This phase is responsible for identifying classes, which are abstractions of the problem domain, and their relationships, such as inheritance, aggregation, and association. During the class-identification process, *class interfaces* are defined. A class interface usually specifies the following information: 1) the class name, 2) the base class(es) of the class, and 3) the definitions of attribute(s)/method(s) associated with their access mechanisms, such as public, protected, and private. The class interfaces are usually specified in *header files*, e.g., files called "***.h**" in C++.
2. **Implementing functional details of classes.**



```
class CShape
{
private:

    [Private Method List...]
protected:

    [Protected Method List...]
public:

    [Public Method List...]
}

class CNode : public CShape
{
private:

    CRect CalcContentTextDimension()
    void DisplayContentText()
protected:
    CRect NodeDimension;
    CRgn NodeRegion;

public:
    CString ContentText;
    Attribute ContentText is used to sho
    of a CNode object
    CPoint Connections[4];

    CNode(CString contentText);
```

Fig. 10. Defining method **CalcContentTextDimension( )** of class **CNode**.



Class Name: CNode

Base Class Name: CShape

Derived Class Name: CRectangle, CDiamond, CRoundRect

Declared Attribute(s): CString ContentText; CPoint Connections[4];

Attribute Declaration: CString ContentText;

Comment: Attribute ContentText is used to show the textual information of a CNode object

Fig. 11. Defining attribute **ContentText** of class **CNode**.

After the first phase, the prototypical implementation of the classes may begin. In this phase, methods defined in a class are implemented with respect to the implementation language. For example, in the C++ language, class implementations are usually specified in *source files* (i.e., “***.cpp**”).

Figs. 10 and 11 show sample construction processes of class interfaces using two programming tools, the flow-based editor and the language-based text editor. The editing process supported by our editors is basically *syntax-directed* [13, 14]. For a placeholder of structured statements, such as those shown in Fig. 4, the editor guides the user to replace it with an instance of some structured statement. For a placeholder of “simple” language constructs, such as assignment statements and expressions, the user can input program text on it directly (i.e., *in-place editing*). In addition to the basic syntax-directed and in-place editing facilities mentioned above, a number of useful facilities, such as *zooming* [18, 27, 43] and *folding* [44], are also supported by the editors. For example, the shadowed rectangles shown on the flow-based editing window denote that some program details are hidden by the user using the folding facilities. Moreover, as Fig. 12 shows, when the user

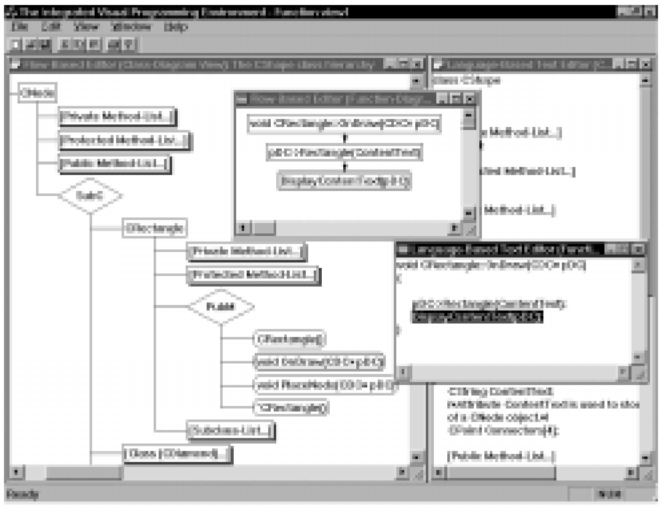


Fig. 12. An example of two-phase object-oriented program construction.

Syntax-directed editing guarantees that the program structure is syntactically correct. In-place editing, however, may cause programming errors because users may input unpredictable program text due to typing errors. This motivated us to design an assistant for in-place editing, which can effectively minimize the occurrences of potential programming errors in object-oriented programs.

4.2 An In-Place Editing Assistant

Consider the internal structure of an object-oriented program; several *statement patterns* appear frequently in such a program. These statement patterns, which are listed below, specify the flows of passing messages to objects:

Pattern 1. Message1(...) or Object1 = Message1(...)
Pattern 2. Object1.Message1(...) or Object2 = Object1.Message1(...)
Pattern 3. Object1.Object2.Message2(...) or Object3 = Object1.Object2.Message2(...)
...
Pattern n. Object1....Objectx.Messagex(...) or Objectn = Object1....Objectx.Messagex(...)

Let these statement patterns be called *message-passing statements*. Appendix A lists the implementation details of a sample method and shows how the message-passing statements are distributed in the corresponding program text. From the example in Appendix A, it can be seen that these message-passing statements make object-oriented programs somewhat complex and hard to understand. We have designed an *in-place editing assistant* to facilitate the construction of such statements. The main design issues for the editing assistant involve: 1) How can the internal program information be systematically collected while users are constructing object-oriented programs? 2) How can the editing assistant analyze the information in order to provide useful programming assistance?

The symbol table, which consists of class and function tables for storing binding information of identifiers within various program scopes, provides valuable information support for the editing assistant. For example, the declaration information of local variables and arguments of a method can be found in the corresponding function table whereas the declaration information of attributes and methods of a class can be found in the corresponding class table. From the viewpoint of object orientation, local variables, arguments, and attributes are all treated as objects.

Before further discussion, an object-oriented program used to illustrate what the editing assistant does will be introduced. The program, including two header files, "**CLASSDEF1.H**"and "**CLASSDEF2.H**", and one source file, "**PUBMETHODD. CPP**", is presented in Appendix B. Fig. 13 shows the symbol table and program tree for these programs. In Fig. 14, an example shows how the *scope rules*, and the *scope-resolution algorithm*, adopted in contemporary object-oriented languages [10, 31], resolve an identifier specified in method **PubMethD( )** of class **ClassD** by traversing the symbol table.

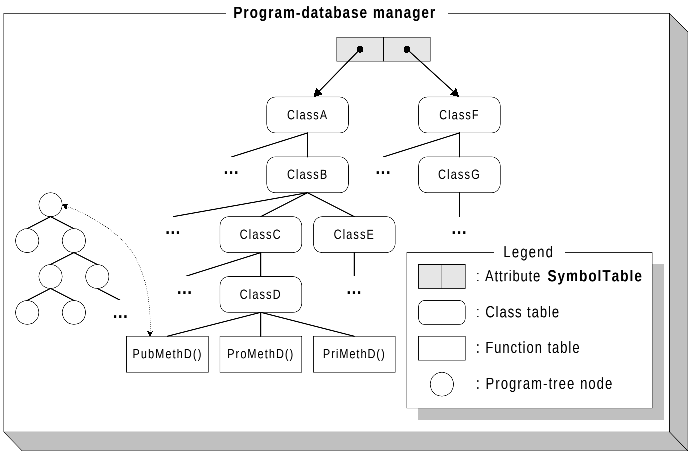


Fig. 13. Internal program representations of object-oriented programs presented in Appendix B.

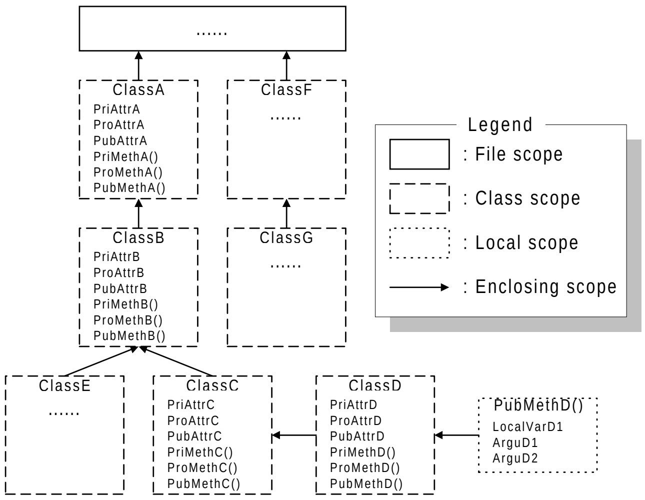


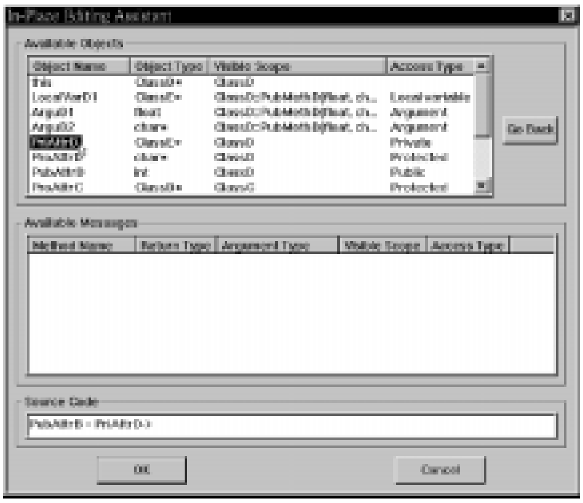
Fig. 14. An example of scope resolution for the symbol table shown in Fig. 13.

Message-passing statements usually consist of two main components: objects and messages. To assist users in constructing such statements, the editing assistant may systematically traverse the symbol table by following the scope rules to collect the objects and messages that are *accessible*. These objects and messages are referred to as *available objects* and *available messages*, respectively. The following design guidelines summarize how the editing assistant collects these available objects and messages associated with an example of specifying a message-passing statement in method **M( )** of class **C**. To make the editing assistant more applicable to object-oriented languages, such as Smalltalk and Java, three specific C++ features involving *friend*, *multiple inheritance*, and *nested classes* have not been included, and all derived classes are assumed to inherit their base classes via the "public" access type (i.e., **class DerivedClass : public BaseClass** {};).

Guideline 1. If the object previously specified is declared as a type of class **C** (e.g., **C* Obj** or **C Obj**), then the subsequent available objects are classified into four categories: 1) local variable(s) of method **M( )**, 2) argument(s) of method **M ( )**, 3) private, protected, and public attribute(s) of class **C**, and 4) protected and public attribute(s) of the base class(es) of class **C**. The base classes of class **C** are those with class scopes enclosing the **C** class scope. For example, the base classes of class **E** in Appendix **B** are classes **B** and **A**. On the other hand, the subsequent available messages are classified into two categories: 1) private, protected, and public method(s) of class **C**, and 2) protected and public method(s) of the base class(es) of class **C**.

Guideline 2. If the object previously specified is declared to be a type of class **B**, not class **C**, then the subsequent available objects are classified into two categories: 1) public attribute(s) of class **B**, and 2) public attribute(s) of the base class(es) of class **B**. On the other hand, the subsequent available messages are classified into two categories: 1) public method(s) of class **B**, and 2) public method (s) of the base class(es) of class **B**.

Fig. 15 shows the user-interface part of the editing assistant, which consists of three panels: the *available-object*, *available-message*, and *source-code* panels. The available-object panel lists four object items: *object name*, *object type, visible scope*, and *access type*.



Object Name	Object Type	Visible Scope	Access Type
this	ClassD*	ClassD	
LocalVarD1	ClassE*	ClassD::PubMethD(float, ch...	Local variable
ArguD1	float	ClassD::PubMethD(float, ch...	Argument
ArguD2	char*	ClassD::PubMethD(float, ch...	Argument
PriAttrD	ClassE*	ClassD	Private
ProAttrD	char*	ClassD	Protected
PubAttrB	int	ClassD	Public
ProAttrC	ClassB*	ClassC	Protected

Method Name	Return Type	Argument Type	Visible Scope	Access Type

```
PubAttrB = PriAttrD->
```

Fig. 15. Selecting an available object.

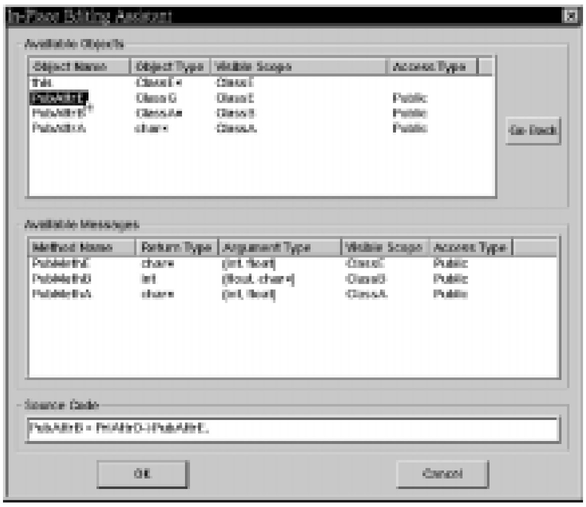
The visible scope is the scope within which an object is declared, while each object is associated with one-of the following five access types: local variable, argument, private, protected, and public. For example, the local variable **LocalVarD1** of method **PubMethD ( )** is declared to be an object pointer to class **ClassE**, and the available-object panel will display the following object information:

Object Name	Object Type	Visible Scope	Access Type
LocalVarD1	ClassE*	ClassD::PubMethD(...)	Local variable

The available-message panel lists five items associated with a method: *method name, return type, argument type, visible scope,* and *access type*. The visible scope is the scope within which a method is declared. There are four access types: private, protected, public, and global, and a method is associated with one type. Overloaded methods can easily be distinguished on this panel because the return and argument types of overloaded methods are different. The source-code panel is used to show the program text of message-passing statements. The user can directly input program text on the source-code panel or use a pointing device (e.g., the mouse) to select the names of objects/messages on the available-object/available-message panels. During the object (or message) selection process, the corresponding names are displayed on the source-code panel.

4.3 Some Examples Using the Editing Assistant

To clarify the guidelines mentioned above, a programming example based on the sample programs given in Appendix B is used to show how the user specifies message-passing statements in method **PubMethD( )** of class **ClassD**. As Fig 16 shows, let the user edit a message-passing statement which looks like "**Object3 = Object1.Object2.Message2 (...)**". After the user selects an object called "**PriAttrD**" with the object type **ClassE***, the editing assistant employs Guideline 2 to collect and display the associated available objects and available messages, as shown in Fig. 16. Figs. 17 and 18 show similar editing activities. In Fig. 19, the source-code panel shows a message-passing statement which is syntactically

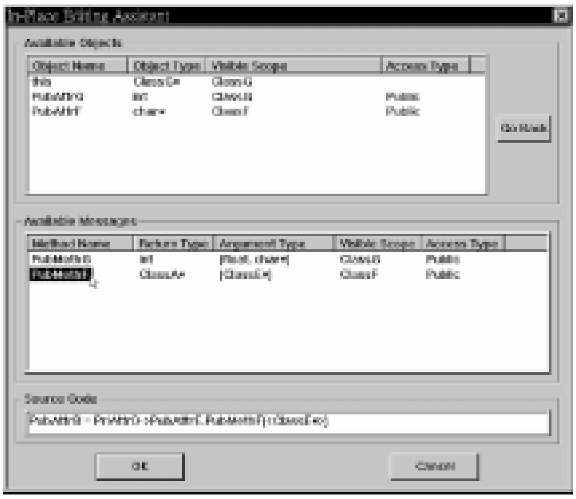


Object Name	Object Type	Visible Scope	Access Type
this	ClassE*	ClassE	
PriAttrE	ClassG	ClassE	Public
PubAttrB	ClassA*	ClassB	Public
PubAttrA	char*	ClassA	Public

Method Name	Return Type	Argument Type	Visible Scope	Access Type
PubMethC	char*	(int, float)	ClassC	Public
PubMethB	int	(float, char*)	ClassB	Public
PubMethA	char*	(int, float)	ClassA	Public

```
PubAttrB = PriAttrD->PubAttrE.
```

Fig. 16. Selecting an available object.

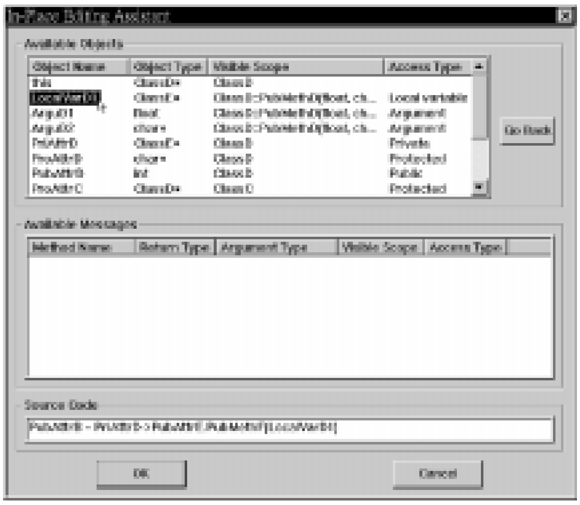


Object Name	Object Type	Visible Scope	Access Type
this	ClassG*	ClassG	
PubAttrG	int	ClassG	Public
PubAttrF	char*	ClassF	Public

Method Name	Return Type	Argument Type	Visible Scope	Access Type
PubMethG	int	(float, char*)	ClassG	Public
PubMethF	ClassA*	(ClassE*)	ClassF	Public

```
PubAttrB = PriAttrD->PubAttrE.PubMethF(ClassE*)
```

Fig. 17. Selecting an available message.



Object Name	Object Type	Visible Scope	Access Type
this	ClassD*	ClassD	
LocalVarD1	ClassE*	ClassD::PubMethD(float, ch...	Local variable
ArguD1	float	ClassD::PubMethD(float, ch...	Argument
ArguD2	char*	ClassD::PubMethD(float, ch...	Argument
PriAttrD	ClassE*	ClassD	Private
ProAttrB	char*	ClassB	Protected
PubAttrB	int	ClassB	Public
ProAttrC	ClassD*	ClassC	Protected

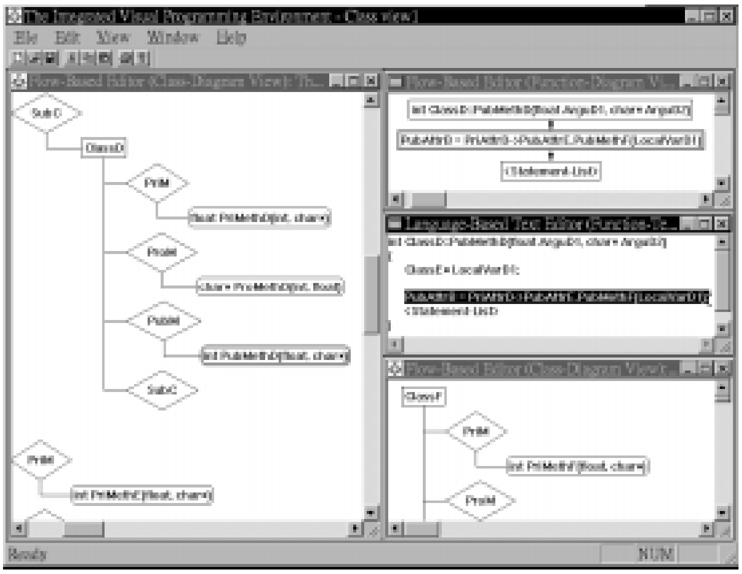
Method Name	Return Type	Argument Type	Visible Scope	Access Type

```
PubAttrB = PriAttrD->PubAttrE.PubMethF(LocalVarD1)
```

Fig. 18. Selecting an available object.

and semantically correct. The member selection operator, "**.**" or "**->**", is chosen automatically by the editing assistant according to the type information of the object previously specified. By interacting with the editing assistant, the user is free from having to type this operator and, thus, can avoid making potential typing errors.

Although the editing assistant provides language-level assistance for specifying such message-passing statements, the user still needs to take care of the cases in which an available object may be un-initialized (i.e., has no memory allocation). For example, the message-passing statement shown in Fig. 19 contains the local variable **LocalVarD1** as the parameter of message **PriAttrD->PubAttrE.PubMethF( )**. Any reference to the contents of **LocalVarD1** will incur a runtime error because **LocalVarD1** is un-initialized as yet.


The Integrated Visual Programming Environment - Class view]
File Edit View Window Help
Flow-Based Editor (Class-Diagram View): Th...
SubD
ClassD
PriM
float PriMethD(int, char*)
ProM
char* ProMethD(int, float)
PubM
int PubMethD(float, char*)
SubC
PriM
int PriMethE(float, char*)
Flow-Based Editor (Function-Diagram V...
Language-Based Text Editor (Function-Te...
Flow-Based Editor (Class-Diagram View):...
ClassF
PriM
int PriMethF(float, char*)
ProM
Ready
NUM

```
int ClassD::PubMethD(float ArguD1, char* ArguD2)
{
    ClassE* LocalVarD1;

    PubAttrD = PriAttrD->PubAttrE.PubMethF(LocalVarD1);
    <Statement-List>
}
```

Fig. 19. Specifying a message-passing statement in method **PubMethD()** of class **ClassD**.

5. RELATED WORK

Many studies on visual programming environments can be found in the literature [1, 2, 4, 46, 47]. On one hand, some of them work on object-oriented languages, and some work on procedural or special-purpose languages. On the other hand, some of them are based on visual syntax and some on textual syntax. This section mainly discusses a number of visual environments that are able to construct object-oriented programs. To make more consistent and objective comparisons of these existing VOOPEs [5, 7-9], we discuss in the following the main features and mechanisms of these VOOPEs in terms of three criteria: *program construction, program visualization,* and *environment evolution*. Program construction is concerned with how the interaction facilities help users to effectively construct object-oriented programs in a visual, textual, and/or incremental way. Program visualization is concerned with how the VOOPE manages multiple and consistent views of (large) programs. Environment evolution refers to how the VOOPE can be integrated with new tool, and customized according to different users' needs.

Visual C++ [9] provides an integrated environment for creating windows-based applications, a framework to begin with, and tools which allow the user to customize his applications. With program construction, Visual C++ is a purely textual, not visual, programming environment. Nevertheless, Visual C++ provides a rich set of text-based, multi-window editing facilities that enable experienced users to efficiently construct programs. For example, it can highlight keywords and format code and parentheses by performing lexical analysis on programs being edited. With program visualization, Visual C++ facilitates program understanding by displaying associated class hierarchies and call graphs, and by providing a navigation facility to locate the definition and reference(s) of an identifier. Consistency maintenance of multiple views is ensured by employing a *update-notification* mechanism based on the document-view architecture. With environment evolution, a Microsoft Foundation Classes (MFC) library [26], an object-oriented framework for Visual C++, can be reused and extended in constructing other, new tools. The Visual C++ environment itself also can be easily customized by enabling (or disabling) existing functions.

SPE (**S**nart **P**rogramming **E**nvironment) [7] is an integrated, yet extensible, environment for all phases of software development in the Snart language. With program construction, SPE supports multiple visual and textual views, each of which can be used to construct and visualize a program. Specifically, the user can create visual windows to depict inheritance and aggregation relationships among classes and/or call graphs denoting the calling relationships among methods. Textual windows are mainly used to work on program details, such as method implementations and class interfaces. This means that SPE is unable to construct program details in a visual way. Moreover, SPE provides simple editing assistance, such as undo and redo facilities, for program construction. With program visualization, SPE employs a mechanism called *update records* to automatically maintain consistency among multiple views of a program. A navigation facility equipped with hyperlinks helps the user navigate through various views. With environment evolution, MViews [48], a generic object-oriented framework, provides basic support for customizing a multiple-view based programming environment and for constructing and integrating new tools. SPE and GERNO, a visual debugger for Snart programs, are specialization examples of MViews.

MCPE (**M**ultitasking **C**++ based **P**rogramming **E**nvironment) [8] is an integrated programming environment for programming in C++. The main goal of MCPE is to solve two problems: *response degression* (i.e., an increasing execution delay when more tools have been included in an environment) and *poor extensibility* (i.e., difficulties in adding new tools to an environment). With program construction, MCPE can contain a number tools, such as a flow editor and a syntax-directed editor, to perform visual and textual programming. According to its present implementations, programming activities are still done textually, so MCPE may be better classified as a visual environment, not as a visual programming environment. With program visualization, MCPE supports multiple views and employs *shared data structures* to maintain consistency among these views. With environment evolution, the MCPE framework simplifies the addition of new tools by using a similar model-view-controller architecture [49] and an *event notification* scheme.

Prograph [5] is a general purpose, object-oriented, visual programming language and environment. The Prograph language combines both object-oriented and data-flow paradigms while the integrated environment consists of a graphical editor, an interpreter, and an application builder. With program construction, all programming activities are done visually by connecting graphical shapes with a mouse. Specifically, methods are implemented by drawing the associated data-flow diagrams. It should be noted that these drawings are the code; there is no corresponding textual representation. Moreover, the editor provides *context sensitive interpretations* of mouse clicks, so that only appropriate actions are performed, thereby preventing the construction of any syntactically incorrect programs. With program visualization, Prograph supports multiple views by creating an edit window for each major component of a program. With environment evolution, a rich class library, called Application Building Classes, may become available to construct and integrate new tools in the Prograph environment.

6. CONCLUSIONS AND FUTURE WORK

Compared with the above VOOPEs, the features and deficiencies of our VOOPE can be summarized as follows. With program construction, our VOOPE provides visual and textual editors that can be invoked on demand to construct and visualize object-oriented programs in any level of detail. For example, a two-phase programming model equipped with syntax-directed editing facilities has been designed to help construct class interfaces and method implementations. However, syntax-directed editing is often criticized in that it puts more restrictions on (experienced) users [43, 50]. To alleviate this difficulty, our VOOPE provides an in-place editing assistant that writes some program statements directly without the need to expand templates further. This editing assistant is quite general and is thought to be widely applicable to most object-oriented languages. A syntax analyzer associated with an incremental semantic analyzer can deal with incomplete program fragments so as to locate possible errors during programming. Our current semantic analyzer provides some primitive functions for checking naming and type errors, such as undeclared/ redeclared identifiers and type incompatibility.

With program visualization, a program-database manager maintains multiple and consistent views by providing a channel, similar to the update-notification mechanism, for tool communication and coordination. Our VOOPE also provides zooming and folding facilities to effectively display large programs. With environment evolution, the MVS class hierarchy can potentially be reused to construct new tools for an existing VOOPE or an integrated visual programming environment supporting more than one language. Our current programming methodology with respect to tool construction and integration is still *imperative*, i.e., it requires manual detailing of source code to the MVS class hierarchy. This methodology may require more effort than the environment-generation approaches [51].

Usability analysis [52] can be employed to make the VOOPE more practical and robust. We plan to perform usability analysis on the current VOOPE, including analysis of friendliness and user acceptance, for users with various levels of programming experience. Their comments will be the basis for revising the current version. We also plan to construct a syntax-recognizing editor [19] that may be preferred by experienced users. On the other hand, some object-oriented language features also make object-oriented programs somewhat difficult to understand and debug. One of our future projects is to construct an incremental code generator and a dynamic visualization tool for examining the dynamic (i.e., runtime) structures of object-oriented programs, so that "incremetality" and "interactivity" can be fully applied to various phases of program development, including coding, compiling, and debugging.

APPENDIX

A. Message-passing statements located in a sample C++ program.

```
Int StatementModel::LoadPTStructure(CDC *pDC, int modelID, IPTFormat *IPTArray)
{
  CString sourceCode;
  Rect oldViewDimension;
  Point midTop;
  int result;
  sourceCode = *(IPTArray[modelID].sourceCode);
  SetSourceCode(sourceCode);                               //Pattern 1: Message1(...)
  if (!sourceCode.IsEmpty()){                              //Pattern 2: Object1.Message1(...)
    if (pCurrentView->pDefaultView !=NLL){                 //Pattern 2: Object->Message1(...)
        pCurrentView->SetViewContent(sourceCode);          //Pattern 2: Object2 = Object1->Message1(...)
        midTop = pCurrentView-> Get Mid Top P6 (  );       //Pattern 2: Object1->Message1(...)
        pCurrentView->PlaceView(pDC, midTop);              pCurrentview, oldviewdimension);
        pCurrentView->pParentView->ChildViewChanged(pDC,   //Pattern 3: Object1->Object2->Message2(...)
    }
  }
  SetComment(*IPTArray[modelID].comment));                 //Pattern 1: Message1(...)
  result = Parse(pDC);                                     //Pattern 1: Object = Message1(...)
  return result;
}
```

B. Sample object-oriented programs.

```
/CLASSDEF1.H
 Include 'CLASSDEF2.H'
class ClassA {
private:
 int PriAttrA;
 int PriMethA(float, char*);
protected:
 float ProAttrA;
 float ProMethA(int, char*);
public:
 char* PubAttrA;
 char* PubMethA(Int, flaot);
};
class ClassB: public ClassA{
private:
 float PriAttrB;
 float PriMethB(int, char*);
};
class ClassD; //forward decl.
class ClassC: public ClassB{
private:
 char* PriAttrC;
 char* PriMethC(int, float);
protected:
 ClassD* ProAttrC;
 int ProMethC(float, char*);
public:
 float PubAttrC;
 float PubMethC(int, char*);
};
class ClassE: public ClassB{
private:
 int PriAttrE;
};
class ClassD: public ClassC
private:
 ClassE* PriAttrD;
 float PriMethD(int, char*);
protected:
 char*ProAttrD;
 char*ProMethD(int. float);
public:
 int PubAttrD;
 int PubMethD(float, char*);
};

//PUBMETHOOD.CPP
 #include 'CLASSDEF1.H'
 int ClassD::PubMethd(
 float ArguD1, char* ArguD2)
{.....}
//CLASS DEF2.H
class ClassA, ClassE;
 class ClassF{
private:
 int PriAttrF;
 int PriMethF(float, char*);
protected;
 float ProAttrF;
 float ProMethF(int, char*);
public:
 char* PubAttrF;
 ClassA* PubMethF(ClassE*);
};
class ClassG: public ClassF{
private:
 float PriMethG(Int, char*);
};
```

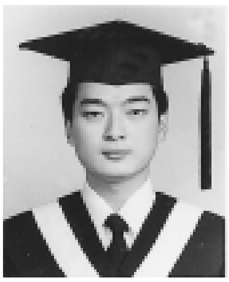
REFERENCES

1. *Visual Programming*, N. C. Shu (ed.), Van Nostrand Reinhold, 1988.
2. *Principles of Visual Programming Language Systems*, S. K. Chang (ed.), Prentice-Hall, 1990.
3. A. Ambler and M. M. Burnett, "Influence of visual technology on the evolution of language environments," *IEEE Computer*, Vol. 22, No. 10, 1989, pp. 9-22.
4. *Visual Object-Oriented Programming: Concepts and Environments*, M. M. Burnett, A. Goldberg and T. Lewis (eds.), Prentice-Hall, 1994.
5. P. T. Cox, F. R. Giles and T. Pietrzykowski, "Prograph: a step towards liberating programming from textual conditioning," in *Proceedings of IEEE Workshop on Visual Programming*, 1989, pp. 150-156.
6. W. Citrin, M. Doherty and B. Zorn, "The design of a completely visual object-oriented programming language," in *Visual Object-Oriented Programming: Concepts and Environments*, M. M. Burnett, A. Goldberg and T. Lewis (eds.), Prentice-Hall, 1994, pp. 67-93.
7. J. C. Grundy, et al., "Connecting the pieces," in *Visual Object-Oriented Programming: Concepts and Environments*, M. M. Burnett, A. Goldberg and T. Lewis (eds.), Prentice-Hall, 1994, pp. 229-252.
8. P. C. Wu and F. J. Wang, "Framework of a multitasking C++ based programming environment MCPE," *Journal of Systems Integration*, Vol. 2, No. 1, 1992, pp. 181-203.
9. Kruglinski, D. J., *Inside Visual C++*, 4th edition, Microsoft press, 1997.
10. A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, 1983.
11. A. N. Habermann and D. Notkin, "Gandalf: software development environments," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 12, 1986, pp. 1117-1127.
12. *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe and E. Sandewall (eds.), McGraw-Hill, 1984.
13. T. Teitelbaum and T. Reps, "The Cornell program synthesizer: a syntax-directed programming environment," *Communications of the ACM*, Vol. 24, No. 9, 1981, pp. 563-573.
14. R. Medina-Mora and P. H. Feiler, "An incremental programming environment," *IEEE Transactions on Software Engineering*, Vol. 7, No. 5, 1981, pp. 472-481.
15. B. Shneiderman, "Direct manipulation: a step beyond programming languages," *IEEE Computer*, Vol. 16, No. 8, 1983, pp. 57-68.
16. S. Meyers, "Difficulties in integrating multiview development systems," *IEEE Software*, Vol. 8, No. 1, 1991, pp. 49-57.
17. S. P. Reiss, "Connecting tools using message passing in the Field program development environment," *IEEE Software*, Vol. 7, No. 4, 1990, pp. 57-66.
18. K. Halewood and M. R. Woodward, "A uniform graphical view of the program construction process: GRIPSE," *International Journal of Man-Machine Studies*, Vol. 38, No. 5, 1993, pp. 805-837.
19. R. A. Ballance, S. L. Graham and M. L. Van De Vanter, "The pan language-based editing system," *ACM Transactions on Software Engineering and Methodology*, Vol. 1, No. 1, 1992, pp. 95-127.
20. T. Tenma, et al., "A system for generating language-oriented editors," *IEEE Transac-*

tions on Software Engineering, Vol. 14, No. 8, 1988, pp. 1098-1109.

21. C. Rich and R. C. Waters, *The Programmer's Apprentice*, ACM press, 1990.
22. G. E. Kaiser, P. H. Feiler and S. S. Popovich, "Intelligent assistance for software development and maintenance," *IEEE Software*, Vol. 5, No. 3, 1988, pp. 40-49.
23. G. E. Kaiser and P. H. Feiler, "Intelligent assistance without artificial intelligence," *Thirty-second IEEE Computer Society International Conference*, 1987, pp. 236-241.
24. C. H. Hu and F. J. Wang, "Constructing an integrated visual programming environment, " *Software - Practice and Experience*, to appear, 1998.
25. C. H. Hu and F. J. Wang, "Constructing flow-based editors with a model-view-shape architecture," in *Proceedings of the International Conference on Distributed System, Software Engineering, and Database System*, 1996, pp. 391-397.
26. *Microsoft Foundation Class Library Reference*, Microsoft Press, 1997.
27. W. Citrin, R. Hall and B. Zorn, "Addressing the scalability problem in visual programming," Technical Report, CU-CS-768-95, Department of Computer Science, University of Colorado, Boulder, 1995.
28. K. J. Lieberherr, I. Silva-Lepe and C. Xiao, "Adaptive object-oriented programming using graph-based customization," *Communications of the ACM*, Vol. 37, No. 5, 1994, pp. 94-101.
29. K. Koskimies and H. Mossenbock, "Scene: Using scenario diagrams and active text for illustrating object-oriented programs," in *Proceedings of International Conference on Software Engineering*, 1996, pp. 366-375.
30. W. De Pauw, D. Kimelman and J. Vlissides, "Modeling object-oriented program execution," in *Proceedings of European Conference on object-orient Programming '94*, 1994, pp. 163-182.
31. B. Stroustrup, *The C++ Programming Language*, 2nd edition, Addison-Wesley, 1991.
32. I. Sommerville, *Software Engineering*, 5th edition, Addison-Wesley, 1996.
33. S. P. Reiss, "PECAN: program development systems that support multiple views," *IEEE Transactions on Software Engineering*, Vol. 11, No. 3, 1985, pp. 276-285.
34. A. M. Sloane and J. Holdsworth, "Beyond traditional program slicing," in *Proceedings of the 1996 International Symposium on Software Testing and Analysis*, 1996, pp.180-186.
35. C. H. Hu, F. J. Wang and W. C. Chu, "Incorporating flow analysis into a flow-based editor," in *Proceedings of the National Computer Symposium*, 1997, pp. D53-D60.
36. H. C. Liao, M. F. Chen and F. J. Wang, "A domain-independent software reuse framework based on a hierarchical thesaurus," revised and submitted to *Software - Practice and Experience*, 1997.
37. C. H. Hu, F. J. Wang and J. C. Wang, "Constructing a language-based editor with object-oriented techniques," *Journal of Information Science and Engineering*, Vol. 11, No. 4, 1995, pp. 1-25.
38. M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, Vol. 10, No. 4, 1984, pp. 352-357.
39. S. Horwitz, T. Reps and D. Binkley, "Interprocedural slicing using dependence graphs, " *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 1, 1990, pp. 26-60.
40. J. Ferrante, K. Ottenstein and J. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 5, 1987, pp. 319-349.

41. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
42. S. B. Lippman, *C++ Primer*, 2nd edition, Addison-Wesley, 1990.
43. J. Welsh, B. Broom and D. Kiong, "A design rationale for a language-based editor," *Software - Practice and Experience*, Vol. 21, No. 9, 1991, pp. 923-947.
44. H. Mossenbock and K. Koskimies, "Active text for structuring and understanding source code," *Software - Practice and Experience*, Vol. 26, No. 7, 1996, pp. 833-850.
45. C. H. Hu and F. J. Wang, "Implementing multi-layered editing facilities in a flow-based editor," in *Proceedings of the 7th Workshop on Object-Oriented Techniques and Applications*, Taiwan, 1996, pp. 388-396.
46. *Visual Programming Environments: Applications and Issues*, E. Glinert (ed.), IEEE CS Press, 1990.
47. *Visual Programming Environments: Paradigms and Systems*, E. Glinert (ed.), IEEE CS Press, 1990.
48. J. C. Grundy and J. G. Hosking, "Constructing multi-view editing environments using MViews," in *Proceedings of 1993 IEEE Symposium on Visual Languages*, 1993, pp. 220-224.
49. G. E. Krasner and S. T. Rope, "A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80," *Journal of Object-Oriented Programming*, Vol. 1, No. 3, 1988, pp. 26-49.
50. S. Minor, "Interacting with structure-oriented editors," *International Journal of Man-Machine Studies*, Vol. 37, No. 10, 1992, pp. 399-418.
51. G. Costagliola, et al., "Automatic generation of visual programming environments," *IEEE Computer*, Vol. 28, No. 3, 1995, pp. 56-66.
52.T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: a 'cognitive dimensions' framework," *Journal of Visual Languages and Computing*, Vol. 7, No. 2, 1996, pp. 131-174.



Chung-Hua Hu (-J¥µØ) received his B.S., M.S. and Ph. D. degrees in computer science and information engineering from National Chiao Tung University, Taiwan, R.O.C., in 1992, 1994, and 1998. He is currently an associate researcher working at Chunghwa Telecommunication Laboratories. His research interests include software engineering, object-oriented techniques, visual programming, and network management.

Feng-Jian Wang (¤ýÂ×°í) graduated from National Taiwan University, Taiwan, R.O.C., in 1980. He received his M.S. and Ph. D. degrees in E.E.C.S. from Northwestern University, U.S.A., in 1986 and 1988. He is currently a professor in the Department of Computer Science and Information Engineering, National Chiao Tung University. His research interests include software engineering, compiler construction, object-oriented techniques, and distributed system software.