Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

Constructing Flow-Based Editors with a Model-View-Shape Architecture*

Chung-Hua Hu Feng-Jian Wang

Institute of Computer Science and Information Engineering
National Chiao Tung University
Hsinchu, Taiwan, R.O.C.

Abstract

During the software design phase, different graphics applications such as flowcharts or data-flow diagrams may be used to depict the software structures and their contents. Although the functions provided by these graphics applications are different, the drawing facilities may be similar and the diagrams produced, in general, are flow-based. This paper presents a model-view-shape architecture, an extension of the model-view-controller architecture, to the construction of a general flow-based editor. On the basis of the model-view-shape architecture, we used Visual C++ to develop a sample application, called the flow-based program editor, and the associated class hierarchy on the Windows environment. The flow-based program editor is intended to provide a flow-based editing environment for the user to visualize and construct a program with the control-flow graph(s). To demonstrate the practicability of the architecture, we also constructed another programming tool, called the syntax-directed text editor, by extending the class hierarchy of the flow-based program editor.

Keywords: visual programming, flow-based editor, syntax-directed editor, model-view-controller architecture, control-flow graph

1. Introduction

It is commonly accepted that software visualization facilitates the comprehension and maintenance of the existing software [1]. During the software design phase, different graphics applications such as flowcharts or data-flow diagrams may be used to depict the software structures and their contents. Although the functions provided by these graphics applications are different, the drawing facilities may be similar and the diagrams produced, in general, are flow-based.

A flow-based diagram, mostly used to describe the information such as a control signal, an event, or a data item flowing from one entity to another, usually contains two kinds of primitive graphical notations: node and link. A node represents an entity such as a software element. A link, associating two nodes, describes the relationships such as control-flow or data-flow relationships between these nodes. Fig. 1 shows two typical flow-based diagrams applied in structured programming. For example, a hierarchical function-call graph is used to describe the calling relationships between functions, while a control-flow graph is used to describe the execution sequence of statements within a function.

In this paper, the *model-view-shape* architecture, which is adapted from the *model-view-controller* [2][3] and *document-view* [4] architectures, is proposed to constructing general flow-based editors. In terms of the functionality of the model-view-shape architecture, a flow-based editor is decomposed into three main modules with a layered structure. The top-layered module, implemented with a set of model objects, is responsible for managing the application's data structure and state. The middle-layered module, implemented with a set of view objects, is responsible for managing the application's presentation. The bottom-layered module, implemented with a set of shape objects, is responsible for drawing

* This research has been supported in part by the National Science Council, under contract number NSC 86-2213-E-009-019.

graphical primitives and handling input events. For each editing action performed by the user, these objects communicate with each other via message passing to complete the graphics display and application-dependent analysis.

On the basis of the model-view-shape architecture, a *flow-based program editor* and the associated class hierarchy have been constructed. The flow-based program editor, running like a *syntax-directed editor* [5][6], provides several useful editing facilities for the user to visualize and construct a program effectively with the control-flow graph(s). To demonstrate the practicability of the architecture, we also constructed another programming tool, called the *syntax-directed text editor*, by extending the class hierarchy of the flow-based program editor.

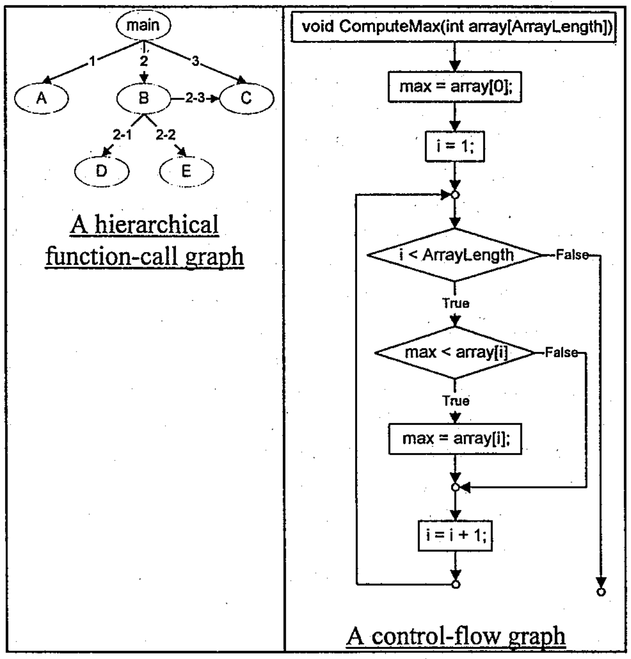


Fig. 1: Two typical flow-based diagram applied in structured programming.

2. The model-view-shape architecture

To construct an interactive and flow-based editor with the capability of presenting and manipulating graphics, we decompose a flow-based editor into three modules: the *application module*, the *user-interface module*, and the *graphics-handling module*. As shown in the left side of Fig. 2, the application module is responsible for managing and manipulating the application's data structure. The user-interface module is responsible for managing the application's display and directing the interaction. The graphics-handling module, which interacts with the user directly, is responsible for drawing the graphical primitives and handling the details of the actual interaction. By treating the three modules as independent components, it is helpful for realizing encapsulation and software reusability.

In this paper, a modified object-oriented approach to the implementation of the three modules, called the model-view-shape architecture, is presented. The association relationship between the three modules and the model-view-shape triad is shown in Fig. 2. That is, the application, user-interface, and graphics-handling modules correspond to the three groups of model, view, and shape objects, respectively. For each module, the functionality is handled by objects of the same group. These objects are cooperative, i.e., they communicate with each other via message passing in order to handle each editing action. Communications between different modules, such as sending a command from one module to another, are also implemented via message passing between objects of different groups. The functionality and design issues of the model-view-shape architecture are discussed as follows.

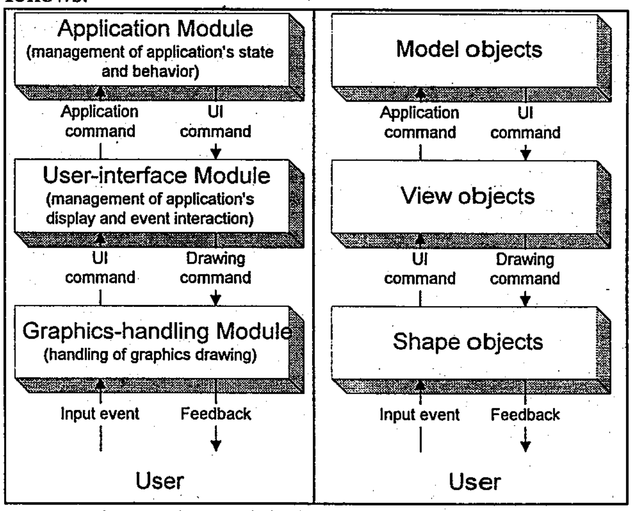


Fig. 2: The model-view-shape architecture.

The shape objects

As mentioned in Section 1, a flow-based diagram usually contains two kinds of primitive graphical components: node and link. From the viewpoint of object-orientation, all nodes and links are treated as shape objects (or say graphical objects) such that each kind of graphical components can be defined with a class. In general, attributes contained in a shape class are used to store the graphical layout and the relevant information such as the dimension and coordinates of a graphical component. Methods contained in a shape class are classified into the following two types:

- Graphics-handling methods. For example, there should be methods to draw the graphical layout at the specific location.
- Event-handling methods. For example, there should be methods to detect and interpret the user-input

event.

The main design issues concerned with shape classes involve: 1) How many kinds of graphical components need to be constructed to shape classes? 2) How can a new graphical component be easily added to the existing component library, i.e., the shape class hierarchy? 3) How can a graphical component be general enough so that it can be reused for the construction of other flow-based editors? In addition, a graphical component may represent different semantic meaning in different kinds of flow-based diagrams, *application* (or *diagram*) *semantics* should be involved in elsewhere rather than in graphical components.

The model and view objects

Since the model and view objects are responsible for managing the application's behavior and presentation, they may be viewed as *application-dependent* objects. That is, for different flow-based applications such as control-flow or state-transition diagrams, different model and view classes may need to be identified and constructed. Application semantics, usually specified via attributes associated with the handling methods, should be placed on the model and view classes.

In a model class, attributes are used to store application-dependent information such as the application's data structure and state information, while methods are responsible for managing the application's state and behavior. For each model object, it is usually associated with a (default) view object, which is responsible for managing the visual presentation. Thus, when a model class is constructed, the construction of the associated view class should be examined subsequently.

A view class has an *aggregation relationship* with one or more shape classes. That is, a view object consists of a or a set of shape objects, which are responsible for actual presentation. Attributes contained in a view class are used to store the higher-level presentation information such as the view dimension. Methods contained in a view class are classified into the following two types:

- View-management methods. For example, there should be methods to calculate and retrieve the view's dimension.
- View-presentation methods. For example, there should be methods to present the view's layout.

The main design issues concerned with model and view classes involve: 1) How many kinds of model and view classes need to be identified and constructed? 2) Which methodology (or guideline) should be employed to classify these model and view classes into the class hierarchies in a systematic and effective manner?

3. A sample application: the flow-based program editor and its implementation

On the basis of the model-view-shape architecture, we used Visual C++ to develop a sample application, called a flow-based program editor (FBPE for short), on the Windows environment. The FBPE is intended to provide a flow-based editing environment for the user to construct a program by drawing the associated control-flow graph. This section beings with an overall description of the user interface, including program editing and presentation, whereas the design and implementation issues of the FBPE will be described later on.

3.1 Graphical representations for language constructs

To facilitate the user to visually and rapidly depict a control-flow graph, it's desirable for the FBPE to provide graphical representations (i.e., *graphical templates*) for high-level language constructs. After studying the language constructs supported in various kinds of imperative programming languages, we designed a set of graphical templates for some well-known language constructs with the syntax employed in the C language subset. Fig. 3 shows several sample graphical templates for structured statements. These graphical templates, which are the building blocks for program construction, can be regarded as the graphical extensions (or visual expressions) to a conventional text-based programming language.

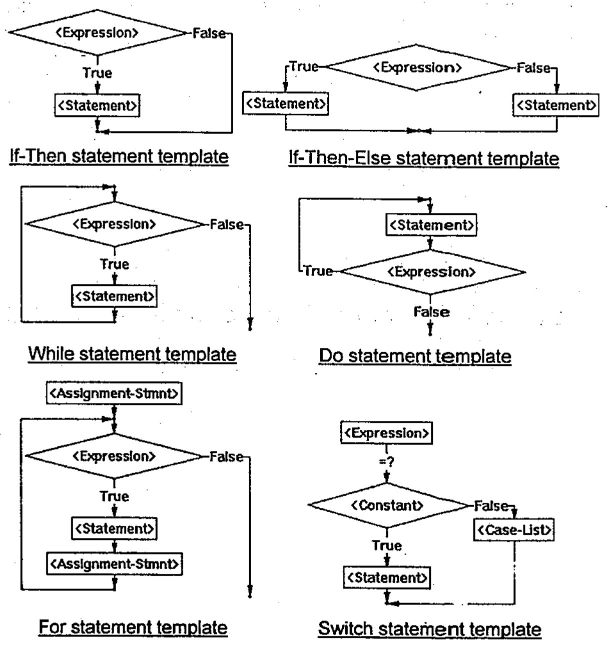


Fig. 3: Example graphical templates for associated language constructs.

3.2 Editing and display activities in the FBPE

Language-based editing model for program construction

The editing model supported by the FPBE is basically syntax-directed. That is, the FBPE directs the user to insert a predefined graphical template into the *placeholder* of another template. The above insertion operation, regarded as a *language command* [5], is performed when the user selects a valid (i.e., syntactically-correct) template from a *template-transformation menu.* The locations of the graphical templates, including coordinates and dimensions, are calculated automatically by the FBPE. Fig. 4 shows the control-flow graph of the `ComputerMax` function before and after the user inserts an `if-then statement` template into a `statement` placeholder.

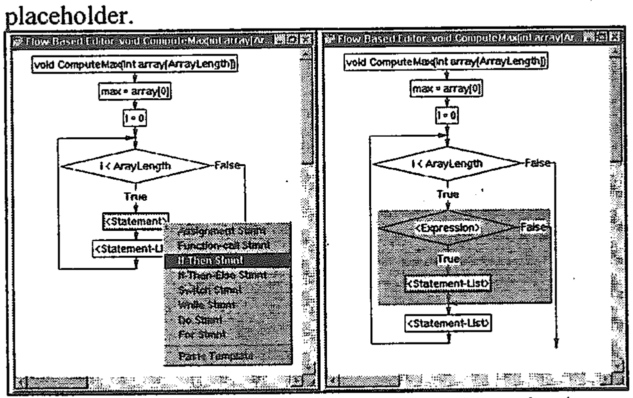


Fig. 4: An example of menu-driven template selection.

When a placeholder represents a simple language construct such as an `expression` or an `assignment statement`, the FBPE provides the visual and on-line editing facility for the user to input program texts on the placeholder directly. Fig 5 shows the animation layout of a control-flow graph while the user is modifying the content of an `expression`. In summary, the basic editing model for program construction proceeds in top-down fashion by inserting templates or program texts into the existing placeholder at the current editing position.

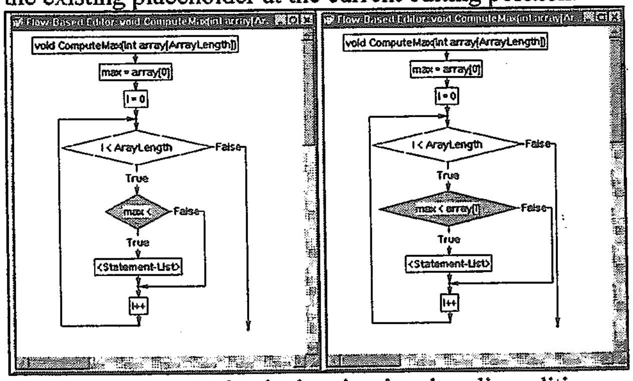
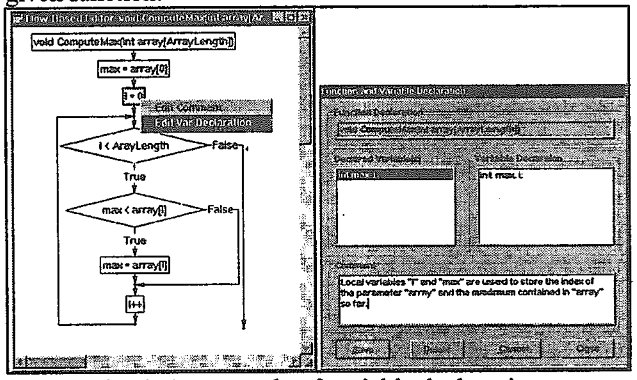

Flow Based Editor: void ComputeMax(int array[Ar...
void ComputeMax(int array[ArrayLength])
max = array[0]
i = 0
i < ArrayLength
False
True
max <
max < array[i]
<Statement-List>
i++

Fig. 5: An example via the visual and on-line editing facility.

Variable declaration

Since a control-flow graph is used to depict the control-flow information of a program, an auxiliary dialog box is provided (by the FBPE) for the user to examine and edit the declaration information of variables in the program. For example, as shown in Fig. 6, while the user edits an `assignment statement`, "`i = 0`", one can issue a language command called "Edit Variable Declaration" to examine all (local) declared variables in a given function.

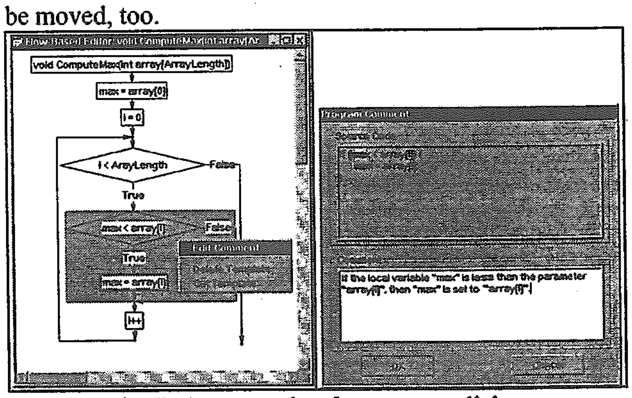


Local variables "i" and "max" are used to store the index of the parameter "array" and the maximum contained in "array" so far.

Fig. 6: An example of variable declaration.

Comment editing

It's commonly accepted that program documentation such as comments is helpful for understanding existing programs. To maintain the relationships between the source codes and the accompanying comments effectively, for each language construct (e.g., a statement or a function) our FBPE allows the user to edit the comment (in a dialog box) by issuing a language command called "Edit Comment" on the language construct. Fig. 7 shows an example of comment editing to illustrate how the user examines and organizes the comment for an `if-then statement.` The input comment is then bound to the corresponding language construct, and will be shown along with the language construct. When the language construct is moved to the other place, the accompanying comment will be moved, too.



If the local variable "max" is less than the parameter "array[i]", then "max" is set to "array[i]".

Fig. 7: An example of comment editing.

The template-based editing facilities

A structured program usually can be viewed as a hierarchical tree structure of nested blocks. A block in the FBPE corresponds to a (graphical) template which may represent a structured statement or a function. The concept of the template-based editing model entails that the user interacts with the FBPE in terms of high-level and well-defined language constructs, rather than low-level and ill-defined symbol (or character) sequences. Thus, the template-based editing model introduces the following two main advantages. First, a program can be constructed more effectively through the composition and manipulation of the constituent templates. Second, the structural correctness of a program is ensured since the constituent templates are well-defined, i.e., syntactically correct.

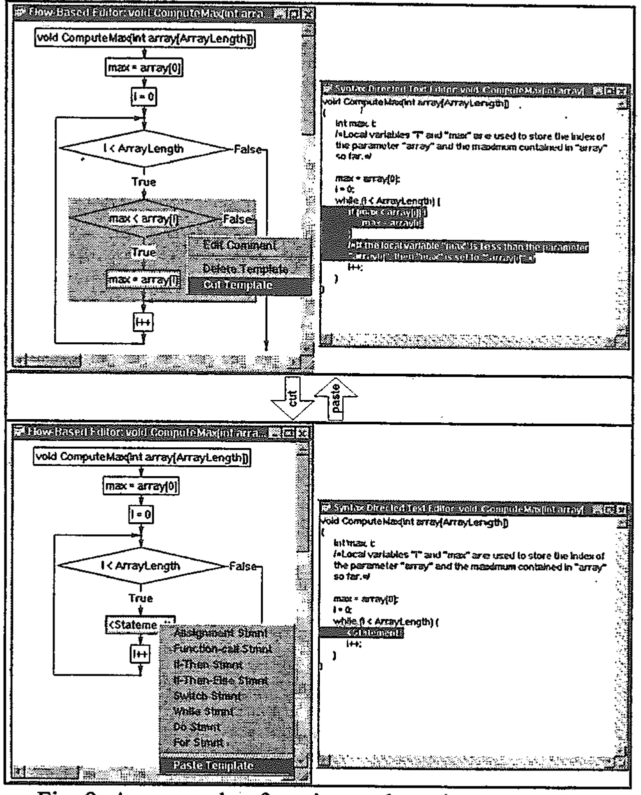
The current FBPE provides three kinds of template-based editing facilities: 1) the cut/paste, 2) the zooming, and 3) the expansion/reduction facilities. The cut/paste facilities are described as follows, while the zooming and expansion/reduction facilities are left to be discussed in [7].

The cut/paste editing facilities

The cut/paste editing facilities are more and more popular and can be viewed as the basic editing operations for the modern text editors. By utilizing the facilities, the user can issue a *cut* command for ordering the FBPE to move a piece of selected texts into a temporary storage buffer, called the *clipboard*, and then retrieve the cut texts back by issuing a *paste* command. From the viewpoint of the FBPE, the piece of selected texts to be cut or pasted always corresponds to a high-level and well-defined (i.e., syntactically-correct) language construct. For example, as shown in Fig. 8, while the user selects the graphical template of an `if-then statement` and issues a cut command, the source code and comment of the `if-then statement` will be moved to the clipboard and the original placeholder reappears.

3.3 A class hierarchy based on the model-view-shape architecture

For a flow-based program editor applying the functionality discussed in the preceding sections, a class hierarchy based on the model-view-shape architecture was designed for the construction of our FBPE. As shown in Fig. 9, the `CModel`, `CView`, and `CShape` class hierarchies correspond to model, view, and shape classes, respectively. The `CShape` class hierarchy consists of two subclass hierarchies; one for node classes and the other for link classes. For these nodes and links, common attributes and generic drawing/interaction methods are defined in classes `CNode` and `CLink`. On the basis of the `CShape` class hierarchy, a new graphical component can be constructed as a customized subclass, which is able to inherit (i.e., reuse) attributes and methods defined in the superclass(es).



```
void ComputeMax(int array[ArrayLength])
{
    int max, i;
    /*Local variables "i" and "max" are used to store the index of
    the parameter "array" and the maximum contained in "array"
    so far.*/

    max = array[0];
    i = 0;
    while (i < ArrayLength) {
        if (max < array[i])
            max = array[i];
        }
        /*If the local variable "max" is less than the parameter
        "array[i]", then "max" is set to "array[i]".*/
        i++;
    }
}
```

```
void ComputeMax(int array[ArrayLength])
{
    int max, i;
    /*Local variables "i" and "max" are used to store the index of
    the parameter "array" and the maximum contained in "array"
    so far.*/

    max = array[0];
    i = 0;
    while (i < ArrayLength) {
        <Statement>
        i++;
    }
}
```

Fig. 8: An example of cutting and pasting a template.

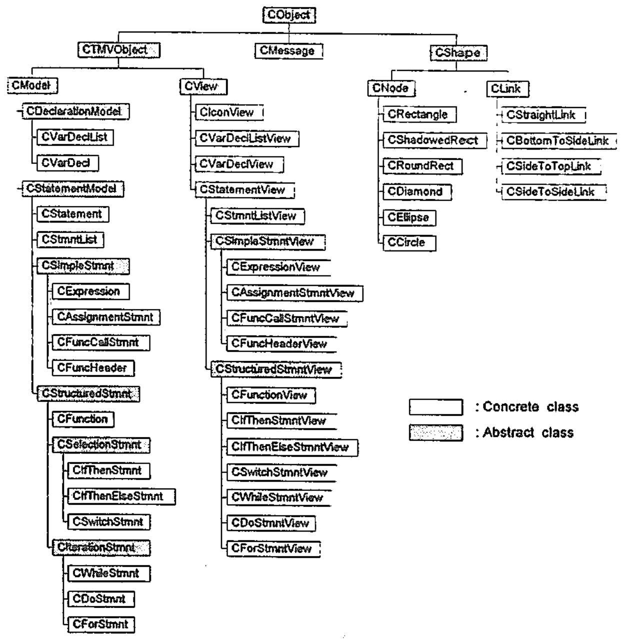


Fig. 9: A class hierarchy for the FBPE.

The construction methodology for the model classes is summarized as follows:

1. For each type of language constructs in a target language, a corresponding model class is identified. A model class usually contains 1) attributes for storing language-dependent information such as source code, comment, and *semantic attributes*, and 2) methods for performing both syntactic and semantic analyses as well as language-dependent functions.
2. The identified model classes are then classified into a hierarchy according to the *functionality* of language constructs. For example, both `if-then-else` and `switch statements` are the selection statements, each of which is also a kind of structured statements. Thus, classes representing different types of language constructs can be easily classified into a hierarchy shown in Fig. 9.

The construction methodology for the view classes is summarized as follows:

1. For each model class, it is associated with a (default) view class. A view class usually contains 1) attributes for storing visual-dependent information such as the template-transformation menu, the view's dimension, and the shape objects for actual presentation, and 2) methods for managing visual-dependent information.
2. The view classes are classified into a hierarchy on the basis of the following two factors: 1) the structure of the model class hierarchy, and 2) the *granularity* of a view object. The term "granularity" is applied to characterize whether a view object is a *unit view*, i.e., a view containing a single node, or a *composite view*, i.e., a view containing a set of nodes and associated links. Example unit and composite views shown in Fig. 9 are the view objects instantiated from the `CSimpleStatementView` and `CStructuredStmntView` class hierarchies, respectively.

While the user edits (or modifies) a program, a *program tree*, an internal representation of the program, is constructed and maintained by the FBPE. Each node in the program tree represents a specific kind of language constructs such as a `statement` or an `expression`. During the construction of a program tree, a tree node is represented by a model object associated with a view object. Fig. 10 shows an example program tree representing an `if-then-else statement` and illustrates the relationships between model, view, and shape objects.

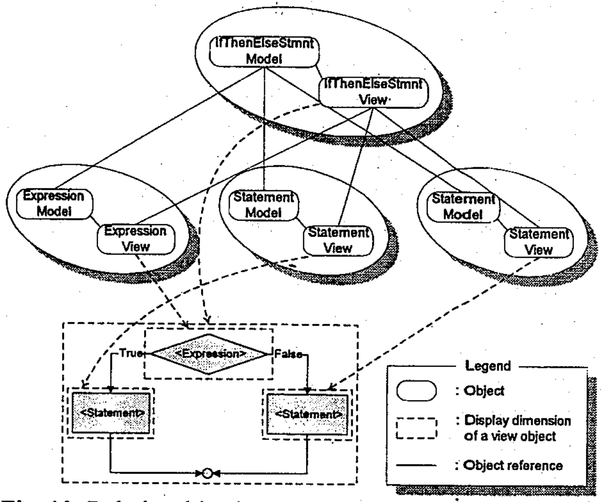


Fig. 10: Relationships between model, view, and shape objects.

4. The second sample application: the syntax-directed text editor and its implementation

Visual programming [8][9][10] is intended to make the programming process easier and efficient, and make a program more readable and maintainable by displaying the program in the form of diagrams. The FPBE, rooted at the technique of visual programming, is expected to meet the above goals. From the user's viewpoint, the visual programming model is more attractive than the traditional programming model, i.e., text-based editing. In terms of the display capability of a program, the textual layout usually takes less screen space than the graphical layout. To enhance the practicability of the FBPE, the syntax-directed text editor[1] (SDTE for short) was constructed as an alternative tool for programming. The right side of Fig. 8 shows the textual layout of the `ComputeMax` function in the SDTE. By implanting knowledge such as language-dependent information in the SDTE, the editor has the capability of displaying a program, including source codes and comments, in the pretty-printed textual layout.

Compared with the FBPE, the SDTE is also a (control) flow-based program editor. The control-flow information of a program is represented implicitly via a sequence of control statements such as selection and iteration statements. The only difference between the two editors is that the SDTE is of no need to draw graphics such as nodes and links. Apart from the graphics-handling facilities, the SDTE performs the same functions as provided by the FBPE. That is, all language and editing

[1] In our previous work [11], we had implemented a syntax-directed text editor, called the *language-based editor*, on the Smalltalk environment.

commands supported by the FBPE are also supported by the SDTE. Thus, the SDTE can be viewed as a simplified version of the FBPE.

To customize the SDTE, the class hierarchy for the FBPE is reexamined. On the basis of the model-view-shape architecture, the class hierarchy is extended as follows:

- The `CModel` class hierarchy. Since the model classes are responsible for managing language-dependent information, all attributes and methods contained in these classes are reused.
- The `CView` class hierarchy. Although the user interfaces of the two editors look different; one is graphics-based and the other is text-based, the view-management methods such as `CalculateViewDimension` and `GetViewDimension` can be reused. In terms of the view-presentation methods such as `PlaceView`, `ReduceView`, and `ExpandView`, they need to be refined.
- The `CShape` class hierarchy. The `CLink` subclass hierarchy is of no use for the SDTE. In terms of the `CNode` subclass hierarchy, the event-handling methods such as `PtInNode` are reused, and the graphics-handling methods such as `DrawNode` need to be refined.

From the above procedures of customizing the SDTE, it can be seen that, instead of being reconstructed or performed a major modification, the existing class hierarchy is extended through the refinement of attributes and methods contained in the existing classes. Moreover, new attributes and methods may be added to the existing class hierarchy if necessary.

5. Conclusion

In this paper, a model-view-shape architecture is proposed to the construction of flow-based editors. For an application designer who wants to construct such a flow-based editor, the functionality and design issues (of the architecture) discussed in the paper provide useful design guidelines. By following the model-view-shape architecture, we constructed a sample application, called the flow-based program editor, on the Windows environment. The flow-based program editor provides several useful editing and display facilities for the user to visualize and construct a program effectively with the control-flow graph(s). Although these editing and display facilities are specified for the customized editor, the conceptual editing models such as comment editing, template-based editing facilities are also applicable for other flow-based editors.

During the construction of the flow-based program editor, the associated class hierarchy was implemented in C++. To demonstrate the class hierarchy of good extensibility, we constructed the second sample application, called the syntax-directed text editor, by refining the class hierarchy. The refinement process shows that, with the object-oriented techniques such as inheritance and polymorphism, the class hierarchy is extended only through the refinement of attributes and methods contained in the existing classes.

Reference

[1] Linos, P. K. et al., "Visualizing program dependencies: an experimental study," *Software - Practice and Experience*, Vol. 24, No. 4, April 1994, pp. 387-403.

[2] Lalonde, W. R. and Pugh, J. R., *Inside Smalltalk*, Vol. 2, Prentice-Hall International, 1990.

[3] Goldberg, A., *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, 1983.

[4] *Class Library Reference - for the Microsoft Foundation Class Library*, Microsoft, 1996.

[5] Medina-Mora, R. and Feiler, P. H., "An incremental programming environment," *IEEE Transactions on Software Engineering*, Vol. 7, No. 5, Sep. 1981, pp. 472-481.

[6] Teitelbaum, T. and Reps, T., "The cornell program synthesizer: a syntax-directed programming environment," *Communications of the ACM*, Vol. 24, No. 9, Sep. 1981.

[7] Hu, C. H. and Wang, F. J., "Implementing multi-layered editing facilities in a flow-based editor," *Proceedings of the 7th Workshop on Object-Oriented Technology and Applications*, Taiwan, pp. 388-396.

[8] Ambler, A. and Burnett, M., "Influence of visual technology on the evolution of language environments," *IEEE Computer*, Oct. 1989, pp. 9-22.

[9] Burnett, M., Goldberg, A., and Lewis, T., *Visual Object-Oriented Programming*, 1995.

[10] Shu, N. C., *Visual Programming*, 1992.

[11] Hu, C. H., Wang, F. J., and Wang, J. C., "Constructing a language-based editor with object-oriented techniques," *Journal of Information Science and Engineering*, Nov. 1995, pp. 1-25.