Enhancing Iconic Program Reusability with Object Sharing

Yuichi Koike, Yasuyuki Maeda and Yoshiyuki Koseki
C&C Research Laboratories, NEC Corporation
4-1-1 Miyazaki Miyamae-ku Kawasaki 216, JAPAN
email: koike@mmp.cl.nec.co.jp

Abstract

This paper describes how to improve the reusability of iconic program modules. In iconic programming systems, the most important features for reuse are the customization of a module and combination of multiple modules without changing their definitions. In order to realize these features, we propose an object sharing technique, which allows components of multiple modules to represent the same object instance. Since a component in a module can be related to an object in another module, by adding a new function to an object in the latter module, the former module can be customized without changing its own definition. In addition, by relating a component to multiple objects in different modules, different functions defined in each module, can be combined easily. Finally, we show that the proposed technique realizes a useful software development style using templates, which will contribute to the improvement of the productivity.

1 Introduction

Iconic programming is based on “node and wire” representation, in which program components are described as nodes and connected together by wires. Iconic programming languages are easy enough to read and understand so that non-expert programmers can use them. However, only a few practical applications have been implemented with iconic programming systems. This is because of scalability [1] and reusability problems. To solve the scalability problem, some techniques such as hierarchical representation [2] and abstraction of iconic programs [3] have been developed. However, there are few ways of improving the reusability of visual languages. In order to improve productivity, the reuse of iconic programming modules is necessary.

We think that customization of a module and combination of multiple modules are the most important features for iconic program reusability. If an iconic program module could be customized, it could be reused in various situations. If modules cannot be combined, there tends to be a number of modules with similar functions. Thus, in this paper, we propose a technique called *object sharing* which enables the customization and combination of iconic program modules. The mechanism to realize object sharing has three characteristics: component information management, the user interface to specify the object sharing relationship, and the instance creation sequence. By this mechanism, components in different modules can represent a common object instance. This object sharing technique plays a key role in enabling module customization and combination.

This paper is organized as follows. Section 2 illustrates the kind of reusability we need. Section 3 describes the basic concept of object sharing. Section 4 illustrates details of object sharing. Section 5 describes how to apply object sharing to improve software development productivity. Section 6 compares the proposed technique with related work and, section 7 gives some conclusions.

2 Reusability of Iconic Program Modules

In iconic programming languages, scalability and reusability problems are the most important from a practical point of view. The scalability problem arises when the application becomes large and complex, and working with programs becomes difficult because of the large number of nodes, wire intersections and wire loops. Hierarchical representation is one of the most effective techniques to scale up iconic programming languages. In this technique, clusters of nodes and wires are represented as a figure. This technique enables a large scale program to be divided into multiple modules so that the number of figures included in one module is small. Hierarchical representation is a technique used mainly for large-scale programs. In addition, Koike proposes a technique for complicated programs, which provides layout flexibility by representing an object as multiple nodes [4]. This flexibility allows a user to reduce the

wire loops and intersections of a complicated iconic program, thereby simplifying it. By integrating this technique with hierarchical representation, large and complicated programs can be represented with multiple modules each of which is small and simple.

These techniques are effective in reducing the number of nodes included in one module. On the other hand, in most cases, they do not reduce the total number of nodes used in the whole application. This is because of the reusability problem. When the reusability of iconic program modules is low, there are many modules with similar functions in a large application, and the number of modules is not reduced. This leads to low productivity. For textual languages, especially object-oriented languages, there are a number of ways of improving reusability, such as inheritance, delegation, and framework. However, there have been few attempts to improve the reusability of visual languages. Therefore, our target is to propose a technique to enhance iconic program module reusability.

Reusability is a vague concept. For example, when a program is flexible to changes in specifications or when a program can be adapted with few alternations to changed specifications, the program's reusability is said to be high [5]. Also, when a function has many arguments and can be customized, the function is said to be reusable. In this section, we define desirable reusability for iconic programming modules.

We assume that customization of a module and combination of multiple modules are the most important features in enhancing reusability. Customization means that in order to be able to use a module in various situations, the behavior of the module needs to be able to be changed. If a module cannot be customized, there will be many modules with similar functions. To accomplish customization, there are two conventional methods. The first one is to use parameters. If a module has a public method with many parameters with which module behavior can be changed, its reusability becomes high. However, it is difficult to use methods with many parameters in iconic programming languages, because such methods need many wires which make the program complex. The second method is to copy and modify the module (Figure 1). If a programmer wants to customize a module, he can copy contents of the module into the current application. Then he can modify it by reconnecting wires. However, when the module definition is changed, the programmer has to adapt the program. If a module is used in many applications, the cost of reflecting the change would become excessively large.

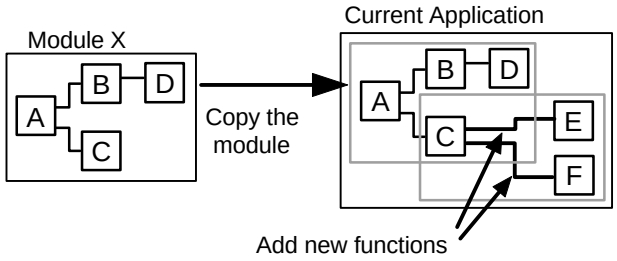

Figure 1 Conventional customization method: copy and modify

Combination means that the functions of a number of modules work together. If modules could not be combined, it would be difficult to make a reusable module. If a module had more functions than needed, situations, in which the module could be reused, would be limited. On the other hand, if a module had fewer functions than needed, productivity could not be improved very much. To accomplish combination, there are some conventional methods. The first one is to copy and join modules (Figure 2). If a programmer wants to combine modules, he can copy the contents of modules into the current application. Then he can relate components in both modules by reconnecting wires. However, as described above, change reflection cannot be achieved with this method. By using textual language methods, such as multiple inheritance or delegation, modules can be combined. However, there are no standard ways of realizing such methods in iconic programming languages. In addition, since these techniques are based on textual information, such as method and variable names, some technical breakthrough is necessary to be able to realize them in iconic programming languages.

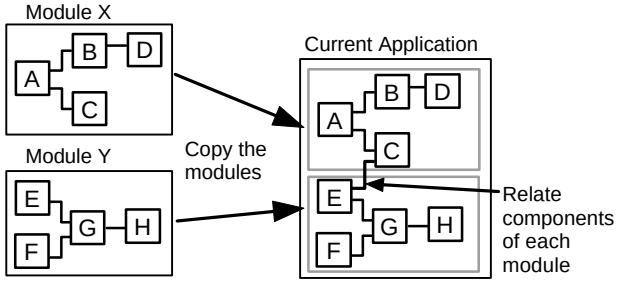

Figure 2 Conventional combination method: copy and join

3 Object Sharing

In order to accomplish module customization and

combination effectively, we propose a technique, called *object sharing*. Object sharing is a technique which allows a number of modules to share an object instance as their component by changing the relationships between objects. Figure 3 shows how objects relate to each other. Without using object sharing, each module has its own components, and they are independent. With object sharing, on the other hand, a number of modules can share an object instance.

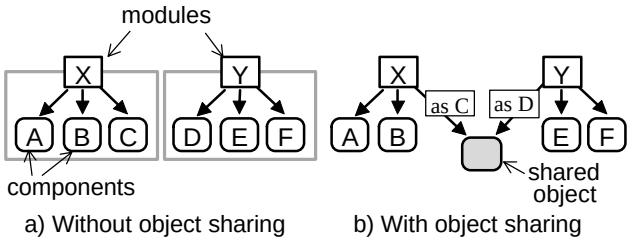

Figure 3 Relationships between objects

Figure 4 shows how to customize a module with object sharing. Module *X* is placed on the current application, and by using object sharing, component *C* of module *X* and component *C*¢ of the current application can represent the same instance. If a programmer adds some functions to *C*¢, this means that the functions are added to *C* of module *X*. That is, module *X* is customized. Both applications in Figure 1 and 4 have the same function. However, if the definition of module *X* is changed, the change is reflected only to the current application in Figure 4.

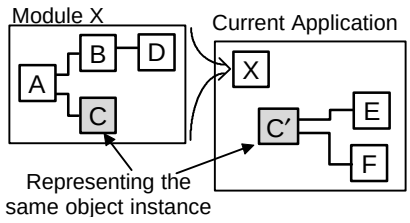

Figure 4 Customizing *X* with object sharing

Figure 5 shows how to combine modules with object sharing. Module *X* and *Y* are placed on the current application, and by using object sharing, component *E* of module *Y* and *E*¢ of the current application can represent the same instance. *C* and *C*¢ also represent the same instance. If a programmer relates *C*¢ and *E*¢, this means that *C* and *E* are related. That is, functions of module *X* and *Y* are combined. Both applications in Figure 2 and 5 have the same function. However, if definitions of module *X* and *Y* are changed, the changes are reflected only to the current application in Figure 5.

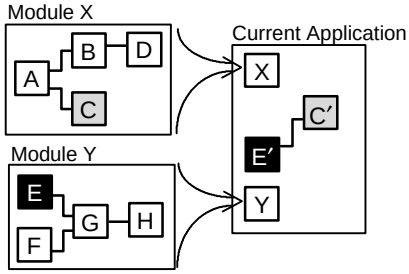

Figure 5 Combining modules with object sharing

4 Object Sharing Details

In this section, object sharing technique details are described. Object sharing is based on object-oriented visual programming languages. Usually, iconic programming modules represent a set of nodes and wires, and incorporate data and procedural abstraction. Before we illustrate object sharing, we will briefly explain the outline of an object-oriented iconic programming system called HOLON/VP, to which the proposed technique has been applied.

4.1 Object-oriented Iconic Programming

In HOLON/VP, an iconic program module is composed of nodes which represent objects and wires which represent control and data flow. The following types of objects are defined in HOLON/VP:

1. Primitive objects, such as strings or numbers.
2. GUI window objects, constructed with a GUI builder
3. DB access objects, constructed with a DB modeler
4. Iconic program modules

On an iconic program editor, a node displays the methods and subcomponents of the corresponding object. To create a node, the programmer first selects an object type. The programmer can then select some public methods and subcomponents of the object that the programmer wants to use. Also, selected methods and components are shown in the node. By connecting them with wires, the programmer can specify control and data flows. An iconic programming module, defined in this way, is also an object, and it can be used in the same way as other objects.

Figure 6 is a sample HOLON/VP iconic program module. There are two nodes, *Window1* and *Window2*, where *Window* is the class name and *1* and *2* are instance IDs. This program defines that two instances of the same type GUI window are created when the pro-

gram is executed. The wire between *Window1* and *Window2* represents control flow, and indicates that the *click* event on *Button1*, a subcomponent of *Window1*, invokes the *hide* method of *Window2*.

Since one node corresponds to one object instance, if a programmer places two nodes of the same class, two different object instances will be created when executing the program.

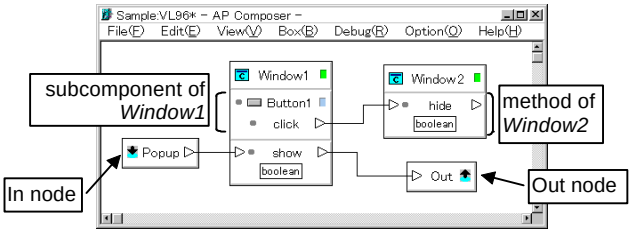

Figure 6 HOLON/VP sample program

An iconic program module in HOLON/VP is defined as a class and represents data and procedures, and it has the following characteristics.

Data abstraction: A node placed in a module represents a component of a module. Each component is defined as a slot of the class. When module instance is created, instances of components are created, and when module instance is destroyed, component instances are destroyed. Thus, a module abstracts its components. For example, the module shown in Figure 6 has two components, *Window1* and *Window2*. An iconic program module also can be a component of another module. Therefore, hierarchical representation is possible.

Procedural abstraction: Modules contain both control and data wires. A chain of control wires is defined as a method. Special nodes, called *In* and *Out* nodes, define public methods of the module which can be invoked from outside the module. As the module shown in Figure 6 has an *In* node, named *Popup*, the module has one public method named *Popup*.

4.2 Object Sharing Mechanism

In order to realize object sharing, the following elements are essential.

1. Component information management
2. User interface to specify shared object
3. Instance creation sequence

These mechanisms are related to each other as shown in Figure 7.

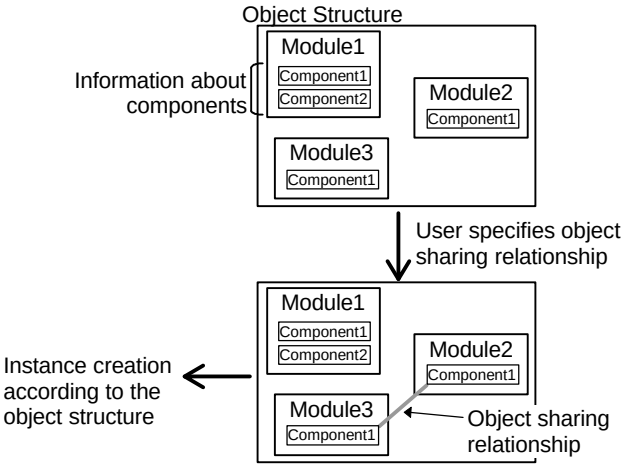

Figure 7 Object Sharing Mechanism

These elements are explained below based on the sample module *Sample*, shown in Figure 8.

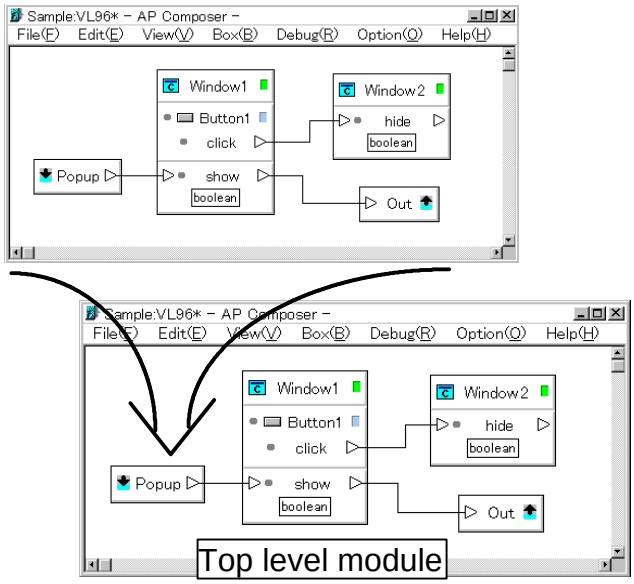

Figure 8 Sample module

Component information management:
To reuse an iconic programming module, it should be placed on another module. Here we call the former the child module, and the latter the parent module. When the child module is placed on the parent module, the child module gives the names and the types (class) of all components to the parent module. Therefore, the parent module has information about all components of all child modules. In Figure 8, one *Sample* module and an instance of *Window*, a GUI window class, are placed on the top level module *Main*. Then, *Main* has the following information:

component class	component name
Sample	Sample1
Window	Sample1.Window1
Window	Sample1.Window2
Window	Window4

User interface to specify shared object:
To specify the sharing object, the programmer uses the *component editor* shown in Figure 9, as follows:

1. Selects a module
2. All components of the module are shown in the list on the component editor
3. Select a component from the list
4. All objects which can be shared as the component are shown in the drop down list on the component editor.
5. Select an object to share

For example, when the programmer selects *Sample1* module, *Window4* and *Sample1.Window2* are shown in the drop down list. If the programmer selects *Window4*, *Sample1* module and *Main* module would share the same object.

To share the same object, class of *Window4* and *Sample1.Window1* must be the same class or have a super-subclass relationship. Otherwise, invalid method calls may occur in the execution phase. Since only such objects as belong to the valid classes are displayed in the drop down list, the programmer cannot select invalid objects.

Conflicts may occur when sharing an object. In such cases, the programmer will be informed of them. For example, there are two modules which include a GUI window, and both of them use the same event handler. Since an event handler is an exclusive resource, a conflict occurs when a programmer wants to make the two modules share the same GUI window object. In such cases, the conflict is notified and the user is asked to determine which one to invalidate.

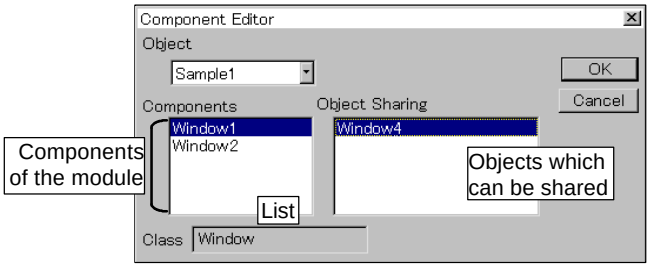

Figure 9 Component editor

Instance creation sequence:
The two steps above are done in the definition phase. To realize object sharing, the instance creation sequence in the program execution phase is also important. Without object sharing, object instances are created as follows:

1. *Main* module, the top level module, is created by the system.
2. *Main* creates *Window4* and *Sample1*.
3. *Sample1* creates *Window1* and *Window2*.

Since each module creates object instances as its own components, modules cannot share a common object. On the other hand, with object sharing, when a shared object is created in a module, it is exported to the related modules. Therefore the modules can share a common object. With object sharing, object instances are created as follows:

1. *Main* module, the top level module, is created by the system.
2. *Main* creates *Window4* and *Sample1*.
3. Since *Window4* and *Sample1.Window1* are the same instance, *Main* sets *Window4* to *Sample1.Window1*.
4. *Sample1* does not create *Window1*, because it has already been set.
5. *Sample1* creates *Window2*.

With this sequence, several modules can share the same object as their components.

5 Applying Object Sharing

This section describes how to apply object sharing to customize a module and to combine modules, using some examples .

Figure 10 shows a module which defines the GUI window transition process. *TransitWin* is a GUI window which has a *Prev* and *Next* button on it. Four instances of *TransitWin* are placed in the module *TransitModule*, and control wires define that the *click* event on the *Prev* button opens the previous window, and the *click* event on the *Next* button opens the next window. By using this module as an application template, it would be possible to construct an application without defining the GUI windows transition process.

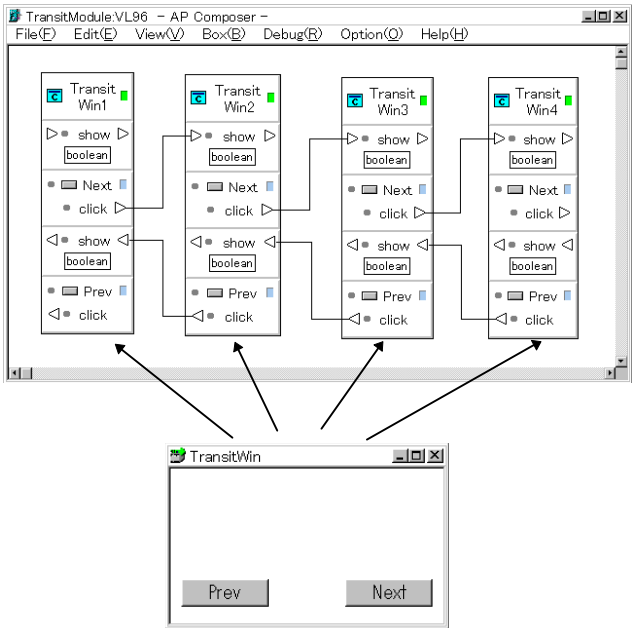

Figure 10 GUI window transition module

5.1 Customizing a Module

This section explains steps to customize the module *TransitModule* by placing a picture on one of the GUI windows (Figure 11).

1. Place *TransitModule* on the current application to reuse the module.
2. Define a GUI window class *PictureWin* which is a subclass of *TransitWin*. *PictureWin* has a picture on it.
3. Place *PictureWin* on the current application
4. Specify object sharing relationship between component *TransitWin4* of module *TransitModule1* and component *PictureWin1*.
5. Since *PictureWin* is a subclass of *TransitWin*, an instance of *PictureWin* class will appear in the execution phase.

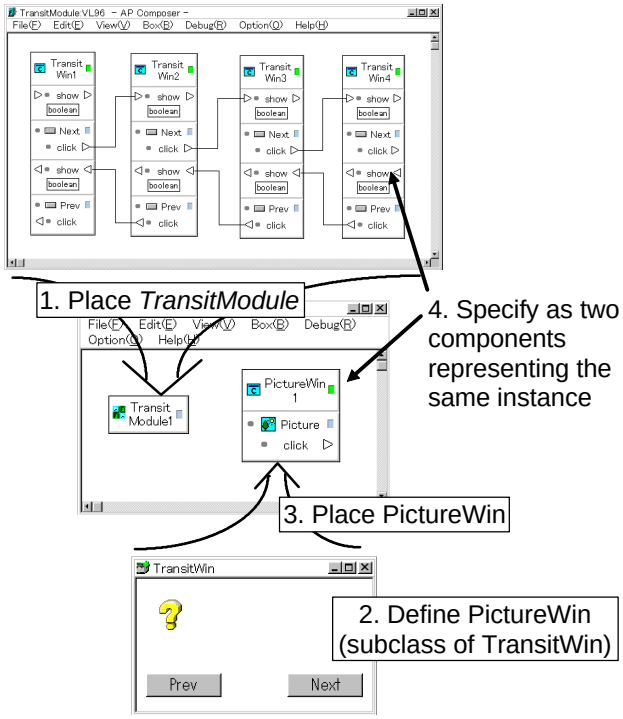

Figure 11 Customizing a module using subclass

Then, another customization is done by connecting wires to the *PictureWin1* component of the current application (Figure 12). Using this customization, a message window opens when the picture on *PictureWin1* is clicked,

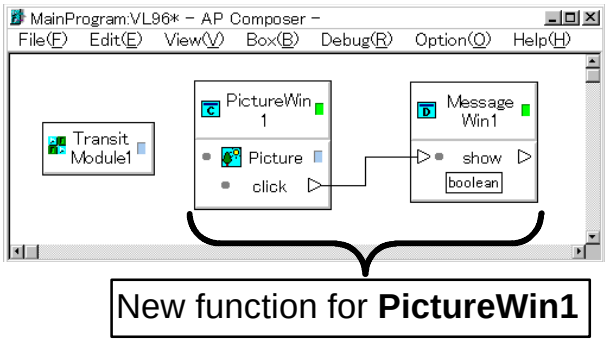

Figure 12 Customizing a module by adding new functions

5.2 Combining Modules

To add a new function to the window, not only customization but also module combination can be used. The *BeepWhenUnload* module defines that a beep emits when a window is closed (use *unload* event which occurs when a window is closed). To combine the *BeepWhenUnload* module with the *TransitWin* module, the programmer should:

1. Place the *BeepWhenUnload* module on the current application
2. Specify the object sharing relationship between the component *TransitWin3* of module *TransitModule1* and the component *Window* of module *BeepWhenUnload1*.

The function defined in *BeepWhenUnload* is added to the *MainProgram* by these steps. As a result, the two modules *TransitModule* and *BeepWhenUnload* are combined.

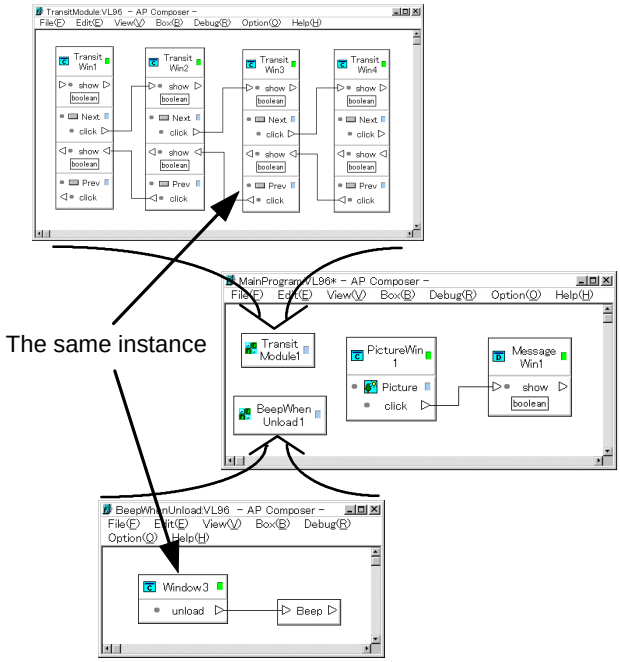

Figure 13 Combining two modules

5.3 Characteristics of Object Sharing

The examples of customization and combination shown in 5.1 and 5.2, indicate some characteristics of object sharing.

The customization achieved by object sharing is flexible, because it not only allows the customization of some attributes, but also the replacement of a component with a subclass instance or the addition of new functions to a component from outside the module.

Figure 14 is an application described without using object sharing. Though it has the same function as Figure 13, module definitions have been modified to construct the application. In contrast with this, modules can be customized without modifying the definition in Figure 13. This means that the same module can be reused in several situations. Moreover, it also means that the definition change of *TransitWin* will be reflected to all applications using *TransitWin*. Therefore, the application using object sharing has flexibility to the definition change of modules.

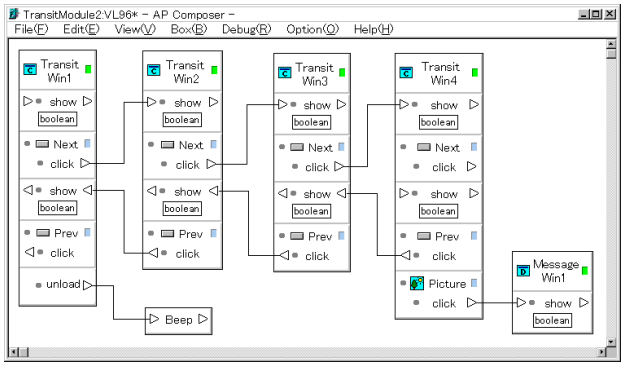

Figure 14 Application without object sharing

The capability of module combination means that a useful software development style is achieved with object sharing. That is, when constructing an application, the programmer can select modules, combine them, and customize them so as to satisfy the target application specification.

6 Comparison with Related Work

There are many GUI builder systems, in which GUI windows are designed visually, and functions attached to the GUI windows are described with textual languages. Some of them have a GUI window inheritance mechanism. The programmer can define a subclass of a GUI window with a graphical editor. Though they are not iconic programming systems, making a GUI window subclass with a GUI builder is much easier than with normal textual languages, and it leads to reusability improvements. In these systems, a GUI window and functions are tied together. Thus, though suitable for reusing a function related with only one GUI window, it is difficult to reuse a function which spreads over multiple GUI windows. In contrast with these systems, object sharing is suitable for such functions.

Vista [6] uses replaceable subcomponents as part of the module's interface. The programmer can customize the module by replacing new subcomponents. "Vista" shows replaceable subcomponents on the node, so that the programmer can easily understand which subcomponents need to be replaced. This technique is similar to ours when customizing a module. However, our technique also enables the combination of a number of module functions.

An Artist's Studio [7] is an iconic programming system which can improve program reusability through the use of a layer mechanism. Common functions are defined in a particular layer and specific functions are defined in other layers. Then the user selects and piles the necessary layers. Since a layer mechanism has been used, the combined functions are displayed as if all functions had been defined in one layer. Compared with our approach, it is easier for the user to understand how a number of functions combines. However, module combination in **An Artist's Studio** is restricted to layout, while our technique allows flexible combinations of modules.

Yang [8] proposes a technique which makes a reusable and abstract program by analyzing the logical relationships of concrete programs. This technique focuses on making reusable modules, while we focus on reusing existing modules. In object-oriented iconic programming languages, such as HOLON/VP, our technique is more suitable, because the programmer has to describe somewhat abstract programs. In contrast with Yang's technique, our technique is not suitable for systems in which the programmer uses concrete data interactively.

7 Conclusion and Future Work

In iconic programming systems, customization of a module and combination of multiple modules are the most important features for reuse. In order to make program modules more reusable, we proposed an object sharing technique, which allows multiple modules to share the same object instance as their component. Since a component in a module is shared by another module as its component, it is possible to customize the module without changing the definition by adding some function to the component of the latter module. In addition, if a number of modules share an object, functions defined in each module will be added to the object. This means that functions of modules are combined. To realize object sharing, three mechanism are important. They are, component information management, the user interface to specify the object sharing relationship, and the instance creation sequence. Finally, we showed how iconic programming productivity can be improved with our technique.

An important future work is to improve the user interface. In the current system, the programmer needs to combine multiple modules by specifying which components of which modules share the same object instance with one another. Therefore, some mechanism to visually indicate the component sharing relationships among the multiple modules will help the programmer to understand program structures.

Another topic for future study is capability check and the navigation of module combination. As described in 4.2, when combining multiple modules, some functions may conflict. Therefore, it is necessary to show the programmer what kind of conflicts occur when combining modules. In addition, to identify multiple module components, the components must either be the same class or have a super-subclass relationship. This means that, not all modules can be combined. Therefore, we should develop a way of searching for a module from the module library which can be combined with the module in question.

References

[1] Burnett M., Baker M.J., Bohus C., Carlson P., Yang S. and Zee P., "Scaling Up Visual Programming Languages," *IEEE Computer*, pp. 45-54, March, 1995

[2] Gorlick M. and Quilici A., "Visual Programming-in-the-Large and Visual Programming-in-the-Small," *Proceedings of the 1994 IEEE symposium on Visual Languages*, pp. 137-144, 1994.

[3] Ford L. and Tallis D., "Interacting Visual Abstractions of Programs," *Proceedings of the 1993 IEEE symposium on Visual Languages*, pp. 93-97, 1993.

[4] Koike Y., Maeda Y., and Koseki Y., "Improving Readability of Iconic Programs with Multiple View Object Representation," *Proceedings of the 1995 IEEE Symposium on Visual Languages*, pp. 37-44, 1995.

[5] Gamma E., Helm R., Johnson R. and Vlissides J., "Design Patterns," Addison-Wesley, 1995

[6] Schiffer S. and Fröhlich J., "Visual Programming and Software Engineering with Vista," *Visual Object-Oriented Programming: Concepts and Environments*, Prentice Hall, Englewood Cliffs, N.J., 1995.

[7] Sengupta S., Kimura T.D., and Apte A., "An Artist's Studio : A Metaphor for Modularity and Abstraction in a Graphical Diagramming Environment," *Proceedings of the 1994 IEEE Symposium on Visual Languages*, pp. 128-136, 1994.

[8] Sherry Y. and Burnett M., "From Concrete Forms to

Generalized Abstractions through Perspective-Oriented Analysis Of Logical Relationships," *Proceedings of the 1994 IEEE Symposium on Visual Languages*, pp. 6-14, 1994.