Abstract[1]

We have developed a software framework for the design of reusable widget classes with OSF/Motif and C++. It is based on an encapsulation of the existing OSF/Motif widget set and the functionality of the X toolkit intrinsics into C++ classes. We have put emphasis on techniques for deriving complex composite widgets completely within C++, and on integrating them properly into the existing C++ widget set. The common behavior of similar OSF/Motif widget types has been factorized out into abstract classes. They provide a higher level of abstraction from user interface details and allow an easier exchange of interface components. The development of an application's user interface is supported by an interface builder which generates code for C++ class definitions. The class library is an example of embedding an existing software package written in a conventional language into an object-oriented environment. The class library is publicly available by anonymous ftp.

Keywords

Software Framework, Data Encapsulation, Object Oriented Class Library, Reusable Software, OSF/Motif, C++, X Window System

[1]Will be published in "Proceedings of the First Annual Motif Users Conference", Washington D.C., December 1991.

1 Introduction

The X toolkit intrinsics offer the basic mechanisms for developing widgets by introducing an object model, a single inheritance mechanism, virtual functions, and polymorphism. However, since C is not an object-oriented language, these mechanisms are merely based on programming conventions. Programming a new widget class then means following a significant number of conventions when defining types, data structures and functions. Implementing a new widget class is a complex and difficult task, even when the derived widget class differs only slightly from its base class.

In order to integrate the OSF/Motif toolkit in a clean way, the Motif widget set and the mechanisms of the X toolkit intrinsics are encapsulated into C++ classes. This is truly a more elegant way to integrate this software packages than simply importing the type definitions and functions by declaring them as external. It offers a means for hiding various unsatisfactory dangerous and concepts, e.g. resource vectors for accessing and modifying resource fields or lack of type checking on resources. Additionally, concepts like the callback mechanism which are sufficient for C, but inadequate for C++ can easily be adapted and extended. Details of the encapsulation are described in chapters 3 and 4.

It is desirable for C++ programmers to derive new widget classes in terms of the syntactical mechanisms provided by the language. Therefore, we have designed a framework which offers mechanisms for deriving new C++ widget classes from the existing C++ widget set. It clearly separates the object-oriented mechanisms provided by the X toolkit intrinsics from those of C++.

The framework's strength lies in the implementation of composite widgets and complex dialog boxes. The newly defined class may easily be reused by several programs or may serve as the base for further specialization. Even in case the new widget class is used only in a single program and is not further specialized, using the framework's mechanisms for data encapsulation leads to better structured and maintainable user interface code. We describe the framework's concepts in chapter 5.

Looking closely at the Motif widget classes reveals a number of different widget classes with similar behavior and semantics, but with a completely different API. For example, the option menu widget and the list widget both realize a one-of-many selection style and the arrow button and push button widgets are both used to initiate an arbitrary action when they are activated. Their common functionality and semantics have been factorized out in abstract classes. Different concrete realizations have been realized using Motif widgets and the concepts of the framework. The abstract base classes separates user interface details from widget semantics. The benefits are a unique handling of formerly different widget classes, simpler and uniform code and a higher degree of exchangeability of different interaction styles. We discuss these topics in greater detail in chapter 6.

2 Related Work

Observing the discussions on various networks reveals that there are a number of research and commercial projects developing software with C++ and OSF/Motif. Most of them use Motif and X by simply including their public C header files containing all required declarations, type definitions and function prototypes. The code is conditionalized and could be used both with C and C++. Programming with C++ and Motif on this level is roughly equal to the way one would program in C with the benefit of a better type checking on function arguments and assignments.

Currently, there are two Motif bindings available for C++:

- The Motif binding from the University of Lowell (UL) [SRM90] has been developed under contract from the Open Software Foundation in 1989. It is available as a commercial product from OSF for outdated versions of Motif and X.
- The Widget Wrapper Library (WWL) [Fek91] is a publicly available binding for Motif. It has been developed as a base for larger applications and is used in various projects.

Both bindings provide a rather thin layer on OSF/Motif, and focus on efficiency and minimization of storage use. Widgets are referred to as true C++ objects, and the OSF/Motif and Xt functions are accessed through member functions. A C++ widget object contains a private pointer to its associated Motif widget as its only member. These objects have the same size as a C pointer because both bindings neither use virtual functions nor multiple inheritance with virtual base classes. Callbacks are handled in the same technical way as in the X Toolkit, except that Motif widget-ids are mapped to their corresponding C++ object when a callback is executed. Adding arbitrary class member functions as callbacks is not fully supported, which leads to programs that do not take full advantage of all C++ features. Although it is possible to derive new widget classes, both bindings are not primarily intended to support this.

Our framework delimitates itself against application frameworks like ET++ [WGM89] and MacApp [Sch86] in that it neither covers concepts like graphics, documents and commands, nor does it provide things like basic data structures or an encapsulation of operating system functionality.

3 Basic Concepts

The basic concept of our framework is that the Motif widget classes are encapsulated into C++ classes. Although it is not really necessary to build up such a parallel class hierarchy, this allows to adapt certain Motif features like resource vectors and callbacks to the special capabilities of C++. Additionally, the object-oriented mechanisms of the X Toolkit Intrinsics (e.g., inheritance and polymorphism) which are merely based on programming conventions could be replaced by mechanisms which are manifested by the programming language itself.

Since the Motif widget classes are already constructed using a single inheritance mechanism, it has been a straightforward decision to build up a class hierarchy which contains a corresponding C++ widget class for each Motif widget class. However, the C++ inheritance graph (Figure 1) differs from the one of Motif in two important ways:

1. The different dialog classes (e.g., form dialogs) have been derived using multiple inheritance. Common aspects like handling of modal dialogs in secondary event loops are defined in the mix-in class `GnDialog`. A new dialog class then inherits its manager capabilities from one of the Motif manager widgets and its dialog capabilities from `GnDialog`.

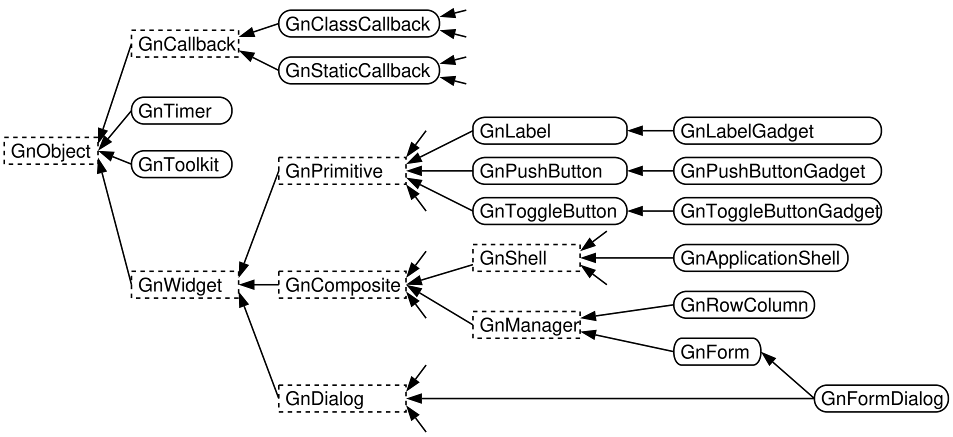

Figure 1: Part of the Framework's Inheritance Graph

2. The gadget classes, which form a separate subtree in the Motif widget tree, have been spread over the C++ class hierarchy and have been made subclasses of their corresponding widget class. For example, the push button gadget is no longer a subclass of gadget, but is now a subclass of the push button widget. The advantage is that programmers can write polymorphic code which accepts push button widgets, but still retain the flexibility to use either the widget or its gadget variant.

The class `GnWidget` forms the base class of all widgets and defines member functions that wrap the X Toolkit Intrinsics functions that operate on widgets. Wrappers for convenience functions (e.g., `XmListAddItem()`) operating on derived widget classes are defined by their corresponding subclass (e.g., `GnList`). The instances of the C++ widget classes contain a private pointer to the actual Motif widget and an additional private member for maintaining a resource vector.

The creation of a C++ widget is divided into three steps: First, the widget object must be created, either by defining it as an embedded member of a composite widget (e.g., a dialog box) or by allocating it dynamically on the heap using the operator *new*. In the second (optional) step, an arbitrary number of resources and constraints may be set using resource modifier functions. The third step creates and manages the associated Motif widget by calling the function `GnWidget::create()`.

```
GnLabel *label = new GnLabel("Hello!");
label->setR_labelAlignment(XmALIGNMENT_BEGINNING);
label->setR_recomputeSize(False);
label->create(parent, "hello_label");
```

Each class defines a set of overloaded constructor functions that cover the most commonly used resource settings. This results in a reduction of code size, requires less programming effort, and leads to clear and comprehensible programs. We also considered the more elegant solution of creating the Motif widget inside the constructor funtion making it possible to initialize and create a widget in a single statement, but have found several arguments against it:

1. The framework is heavily based on virtual functions and polymorphism (see chapter 5), which is especially true for the widget creation phase. This would require the execution of virtual functions from inside a constructor, which does not work in C++ (see [ES90], page 294). Therefore, we had to separate object initialization and the creation of its associated Motif widget.
2. We can cover only a small subset of all possible resource settings by overloaded constructors. One possible solution would be to set additional resources after the widget has been created, but this contradicts to the Motif philosophy that resource settings and changes belonging logically together should be made at once. There are also some resources that can not be changed after the widget was created. Another solution would be to define an extra boolean argument in each constructor which suppresses widget creation in cases where resources not covered by constructors should be set. We could then set additional resources after the widget object has been created and must create the associated Motif widget explicitly.

Our solution is to define a relatively small set of constructor functions which cover only those one or two resources that are almost ever changed. Additional resources must be set or modified using modifier functions. This scheme requires only little learning effort because programmers can easily remember the small number of constructors. Additionally, the mapping of modifier function names to resource names (described below) is straightforward and can easily be derived from the OSF/Motif documentation. The effect is that the framework's documentation is easy to write and to maintain and that programmers have to consult it only in rare cases. Also, much of a programmers previous knowledge about OSF/Motif remains useful when developing with C++.

The C++ widget classes define accessor and modifier functions for each resource of their corresponding Motif class. A resource `XmN`*resource* of type `t` has an associated modifier function `setR_`*resource*`(t)` and an associated accessor function `t getR_`*resource*`()`. These functions introduce strong type checking on resource fields and ensure that a certain resource field can only be set for a widget really owning it. The modifier functions are overloaded for certain resource types in order to provide automatic resource conversion. For example, modifiers for XmString resources could be called with simple character strings which are automatically converted to XmStrings. The accessor functions deliver the resource field's contents as their return value, making it unnecessary to allocate temporary storage for argument vectors and resource variables.

Although resource accession and modification is realized and controlled by member functions, it is still based on buffering name/value pairs in a resource vector. The C++ widget objects hide the error prone concept of resource vectors by internally managing a dynamically growing resource vector. The vector is automatically allocated when the first resource is accessed or modified and is deallocated after either the widget is created or the function `GnWidget::SetValues()` is called.

For each callback resource named `XmN`*callback* defined by a widget class, the two member functions `add_`*callback*`()` and `remove_`*callback*`()` are defined. Since callbacks are a more complex issue in our framework, we discuss them in detail in the next chapter.

All application-related functionality of the Motif toolkit (e.g., `XtAppMainLoop()`) is encapsulated by the class `GnToolkit`. Each application creates exactly one instance of this class. This instance maintains the Motif-related global data, such as an application context, display pointers, as well as a set of fallback resources.

4 Callbacks

Our goal was to provide a callback concept that allows all types of C++ functions, especially non-static member functions, to be used as callbacks. Additionaly, it should provide a means for using ad-hoc polymorphism (e.g., virtual functions) in callbacks. In order to support a clean programming style, it should be possible to pass any number of client data arguments to a callback function.

The X Toolkit Intrinsics require that callback function arguments follow a specific pattern: The first argument must be declared as a pointer to a widget and is getting instantiated with the widget executing the callback procedure. The second and third arguments must be both declared as `caddr_t`. They are instantiated with the widget's client data and call data, respectively. We have found this scheme to be inadequate for C++ for several reasons:

- For a non-static class member function to be used as a callback, its first argument must be the `this` pointer. In a simple mapping, this pointer could be provided by the callback's client data argument. We have found this to be too restrictive, because it limits the client data to be of a specific type.
- Although we could easily map the Motif widget pointer to its associated C++ object, we have found that callbacks operate on the widget object only in rare cases. Therefore, we decided that the programmer should pass in the widget object as a client data argument in places where it is really needed.
- Callbacks are limited to take only a single client data argument. If more than one is required, additional structures must be declared which hold them. This usually results in uncomprehensible programs and increases programming effort.

Our solution is based on the class `GnCallback`, which serves as a link between the Motif widget and the C++ callback procedure (Figure 3, page 9). Its instances hold all the information necessary to execute a callback procedure. Subclasses of `GnCallback` cover the different kinds of member functions as well as functions with different numbers of client data arguments (Figure 2). For example, an instance of the subclass `GnClassCallbackOne` contains the information necessary to call an ordinary class member function with one client data argument.

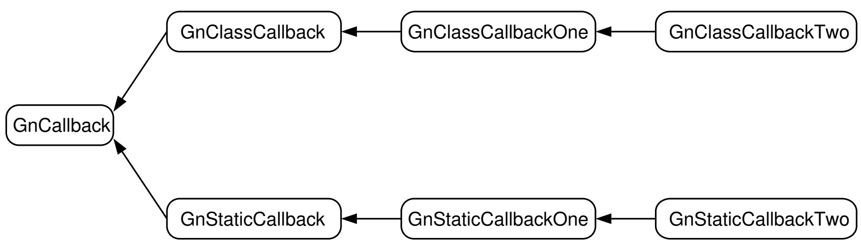

Figure 2: Abstract and Concrete Callback Classes

To attach a callback to a widget, the programmer must first create a callback object holding the information necessary to call a C++ function, namely the function pointer and the object on which the callback is to be executed, plus any number of client data arguments. Then the callback object must be added to the widget's callback list using one of the `add_`*callback*`()` functions mentioned in the previous chapter. Attaching callbacks is supported by a set of preprocessor macros and usually requires only a single statement. We show this by an example program:

```
class CallbackDemo {
  public:
    CallbackDemo(GnWidget &parent);
  private:
    void Greetings(char *, caddr_t);
    GnToggleButtonGadget button;
};

CallbackDemo::
CallbackDemo(GnWidget &parent)
{
  GnCallback *cb;

  button.create(parent, "greetings");
  cb = CALLBACK1(CallbackDemo, Greetings, this, char *, "Hello");
  button.add_valueChangedCallback(cb);
}

void CallbackDemo::
Greetings(char *s, caddr_t call_data)
{
    XmToggleButtonCallbackStruct *cbs =
        (XmToggleButtonCallbackStruct *)call_data;

    if( cbs->set )
        cout << s << endl;
}
```

The class member function `CallbackDemo::Greetings()` is attached as a callback procedure to a toggle button widget. The macro `CALLBACK1` creates a callback object and stores it on the temporary variable `cb`. In the next step, the callback object is registered with the widget. The macro's first argument determines the class where a function is to be called. The second and third argument point to the actual class member function and to the object on which this function is to be executed. The fourth and fifth argument determine the type and value of the client data argument.

The macro needs the class name and the client data's type to provide enough information to the compiler to perform any necessary type casts and to ensure that the function is really a member of the given class. However, the compiler can not statically check that types of the objects and client datas stored in a callback object actually match the argument types of the callback function. As it is also the case in the X toolkit intrinsics, it is still left to the programmer to ensure exact type matches.

4.1 Internals of the Callback Mechanism

The framework always attaches its private general callback handler `GnCallbackHandler` as the real callback to the Motif widget and stores a pointer to the callback object in the client data argument.

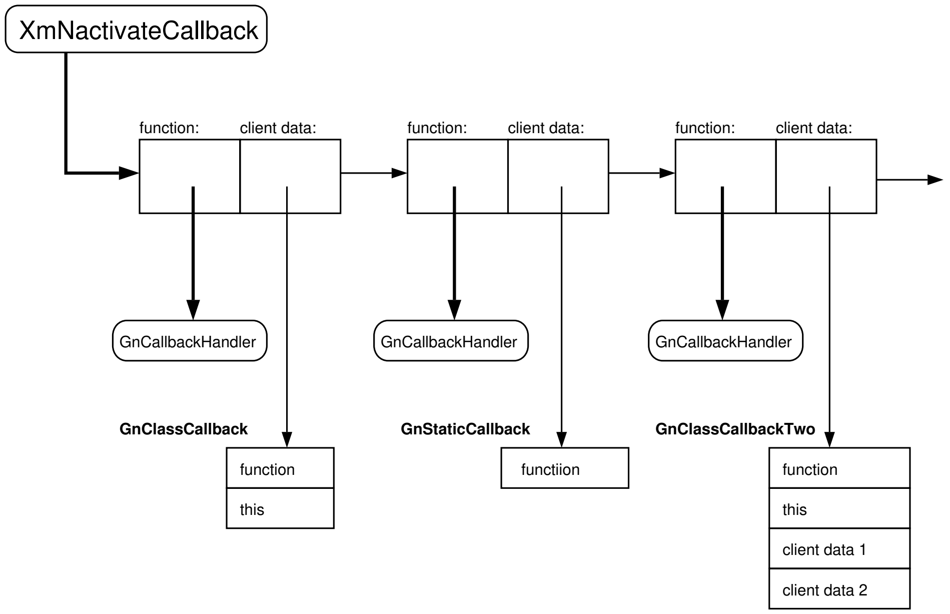

Figure 3: Connection between Motif Callback lists and C++ Callback Objects

When the callback gets executed, the generic callback handler calls the virtual function `GnCallback::execute()` on the callback object.

```
void GnCallbackHandler(Widget, caddr_t client_data, caddr_t call_data)
{
    GnCallback *callback_object = (GnCallback *)client_data;

    callback_object->execute(call_data);
}
```

This function is in turn responsible for calling the function stored in the callback object:

```
void GnCallback::execute(caddr_t call_data)
{
    (function_ptr)(call_data);
}
```

The class `GnCallback` is responsible for callbacks which are free functions without any client data arguments. It defines the member `function_ptr` holding the function's address. The class `GnClassCallback` is responsible for callbacks which are class member functions without client data arguments. It defines an additional member `obj` which contains the address of the class object. In the example above, the macro `CALLBACK1` returns an object of class `GnClassCallbackOne`, which is a subclass of `GnClassCallback` and defines an additional member `client_data` which holds the client data argument. In order to cooperate well with the general callback handler, both classes refine their virtual function `execute()`:

```
void GnClassCallback:: execute(caddr_t call_data)
{
    (function_ptr)(obj, call_data);
}

void GnClassCallbackOne:: execute(caddr_t call_data)
{
    (function_ptr)(obj, client_data, call_data);
}
```

4.2 Extending the Callback Mechanism

The framework provides callback classes for both class member and free functions with zero, one, or two client data arguments. Our experience shows that in most cases we did not require a client data argument at all, because the object on which the callback procedure is being executed contains all the required state information. However, if more than two client data arguments are required, programmers can easily extend the generic callback mechanism by defining a further subclass of `GnCallback` holding the additional client data arguments, refining its member function `execute()` and by writing a supporting macro.

4.3 Other Applications of Callback Objects

Callback objects can also be used as event handlers and protocol callbacks handlers:

```
GnCallback *cb;

// Set up an event handler which
// reacts on button release
cb = CALLBACK1(CallbackDemo, Greetings, this, char *, "Hello");
label.AddEventHandler(cb, ButtonReleaseMask);
```

Callback objects can even be used for purposes which have nothing to do with widgets. For example, they could be used as a callback for a timer object, which serves as an encapsulation for the functions `XtAppAddTimeout()` and `XtAppRemoveTimeout()` from the X toolkit intrinsics:

```
// Create a timer which triggers its
// callback every 250 milliseconds
GnTimer timer(250);
cb = CALLBACK1(CallbackDemo, Greetings, this, char *, "Hello");
timer.AddCallback(cb);
```

4.4 Improvements of the Callback Mechanism

The callback mechanism is still somewhat unsatisfactory because typecasts are always necessary when a callback is added. If we view adding callbacks as specializing a widget's behavior, we could adapt a technique already known from MacApp [Sch86]: All callbacks are mapped on virtual functions. This mapping hides all unclean constructs. Instead of adding a callback, we must derive a subclass which refines these virtual functions. Since these functions have a fixed signature, we must keep all client data arguments in the widget object itself.

The main advantages of this concept are better structured code and no more unsafe type casts when adding callbacks. Its main drawback is that a subclass must be defined for every widget being used in an application's user interface. This becomes somewhat less important when user interface design is aided by an interface builder that could generate the bulk of code.

5 Framework Concepts

The framework provides mechanisms for deriving new C++ widget classes. Emphasis is put on the implementation of composite widgets and complex dialog boxes. The new widget classes integrate cleanly into the existing widget tree. From an application programmer's point of view, these classes are handled using the same mechanisms as for the existing classes.

We discuss our concepts by describing the specification and implementation of the class `GnLabeledText`. This class is a combination of a label and a text widget, which is often required in user interfaces, for example when designing input forms for database applications. Since our aim is to increase software reuse, we implement this combination as a new widget class.

From a *structural* point of view, a labeled text widget is a composite widget. It consists of a label widget and an editable text widget. An additional row column widget arranges the label and the text either horizontally or vertically, with the label widget being positioned either above or to the left of the text widget.

From a *behavioral* point of view, a labeled text widget is a primitive widget. It is a specialization of the Motif text widget which controls an additional label string and the arrangement of the two components.

To bridge the gap between structural and behavioral aspects, we *inherit* behavior from the text widget class and *use* the label and row column classes for the implementation (see [Mey88]).

We decided that the text widget's functionality is of primary interest, whereas the label string and the component arrangement plays only a secondary role. Although this seems to be quite natural, it would have been possible to focus on structural aspects. In this case, we would have *inherited* from the row column class and would have *used* the text and label classes.

The above discussion leads to a first class definition:

```
typedef enum GnLabelPosition { GnTop, GnLeft };

class GnLabeledText : public GnText {
  public:
    GnLabeledText(GnLabelPosition pos = GnLeft);
    GnLabeledText(char *label_string, GnLabelPosition pos = GnLeft);
    GnLabeledText(XmString label_string, GnLabelPosition pos = GnLeft);
  protected:
    GnRowColumn     row_column;
    GnLabel         label;
```

```
};
```

The enumeration type `GnLabelPosition` determines the arrangement of the label and the text widget. Internally, these values are mapped onto the row column's `XmNorientation` resource. We define a set of overloaded constructor functions whose arguments provide initial values for both the label's string and position. The constructors can handle both ordinary C strings (`char *`) and Motif strings (`XmString`). We show the implementation of the second constructor here:

```
GnLabeledText::
GnLabeledText(char *string, GnLabelPosition pos)
  : label(XmStringCreateLtoR(string, XmSTRING_DEFAULT_CHARSET)),
    row_column(pos == GnLeft ? XmHORIZONTAL : XmVERTICAL)
{ }
```

The constructor's purpose is to propagate its arguments to the constructors of the label and row column widgets. The label string is converted to a Motif string before it is passed to the label's constructor. However, this conversion is not really necessary because the constructor of the label class is also overloaded to handle both C and Motif strings.

The label and row column widgets are both declared as embedded objects. This simplifies the code for object initialization because embedded objects are automatically allocated, initialized and deallocated with their enclosing object. A further, but less important, benefit is that embedded objects consume less space and time than objects allocated on the heap.

A labeled text widget is created by declaring the widget object and then calling the function `GnWidget::create()`. The function arguments are the parent and name of the widget:

```
GnLabeledText text("Name:");;
text.create(parent, "nameField");
```

In this example, the labeled text widget is automatically managed after creation. The widget's preferred managed state is determined by calling the virtual function `preferred_managed()`. This function returns `False` for subclasses of `GnDialog` and `True` for all other classes. In order to create widgets in an arbitrary initial managed state, the function `create()` has been overloaded to accept an additional third boolean argument determining whether the widget is automatically managed after creation.

The function `GnWidget::create()` serves as a driver for creating widgets. It calls several virtual functions which are introduced pure virtual in the class `GnWidget` and which make up the framework's general behavior.

```
void GnWidget ::
create(GnWidget *parent, char *widget_name, Boolean managed /* = True */ )
{
    before_create(parent,widget_name,managed);
    widget_id = create_widget(parent, widget_name);
    toolkit->widget_table.insert(this, widget_id);
    reset_args();
    after_create(parent,widget_name,managed);
```

```
        if( managed )
            manage();
    }
```

The most important one is `GnWidget::create_widget()`, which creates the main widget of a C++ widget. In our example, this widget is a text widget. Since the base class `GnText` already creates a widget of this class, we normally would not have to override this function in our class. We first give the implementation of this function for the base class:

```
Widget GnText::
create_widget(GnWidget *parent, char *name, Arg *args, int n_args)
{
    return XmCreateText(parent->widget_id_for_sons(), name, args, n_args);
}
```

However, this implementation is not sufficient for our widget class because the text widget's parent must be the local row column widget instead of the labeled text's parent. We solve this by refining the `create_widget()` function:

```
Widget GnLabeledText::
create_widget(GnWidget * /* parent */, char *name, Arg *args, int n_args)
{
    return XmCreateText(row_column.widget_id_for_sons(), name, args, n_args);
}
```

The next problem we have to solve is that the row column widget must be created before the text widget because otherwise we can not use it as a parent for the text widget. The row column widget also demands that the label widget must be created before the text widget in order to position it above or to the left of it. To achieve this, we refine the function `before_create()`, which is called by `create()` immediately *before* the function `create_widget()`. We first have to extend the class definition:

```
 protected:
   virtual void before_create(GnWidget *parent, char *name, Boolean managed);
```

The function is declared as protected, which ensures that it is hidden for client classes, but is accessible for the framework and for subclasses we might derive in the future. The function `GnLabeledText::before_create` is called with the same actual parameters that were passed to `GnWidget::create()`. The implementation of `GnLabeledText::before_create` is then straightforward:

```
void GnLabeledText::
before_create(GnWidget *parent, char *name, Boolean managed)
{
    row_column.create(parent, "LabeledTextRow");
    label.create(row_column, "LabeledTextLabel");
}
```

The function `after_create()` provides an additional important hook for intervening the widget creation procedure. It is called by the framework immediately *after* the primary widget has been created. It is intended to be refined for classes whose primary widget is itself composite and contains further child widgets. It might also be used to attach callbacks to the primary widget or its children.

The labeled text widget is conceptually a primitive widget, but is internally modelled as a composite widget that creates additional hidden widgets above itself in the widget tree. This affects several functions which operate on whole widget groups, such as managing, moving, or destroying widgets, because they must be called on the local root of a composite C++ widget. For the labeled text class, this means that all operations on the complete group must be executed on the row column widget. The framwork already takes care of this by calling the virtual function `get_group_widget()` in order to determine the widget id each time an operation is to be performed on a group of widgets. We therefore must refine this function to return the row column's address. We first extend the class definition with

```
protected:
  virtual GnWidget *get_group_widget();
```

and then implement the function as follows:

```
GnWidget *GnLabeledText:: get_group_widget()
{
    return &row_column;
}
```

The `GnLabeledText` class is a useful example for demonstrating the framework's capabilities. But in real life one would more often use it to derive more complex composite widgets. A new composite class usually inherits from one of the predefined manager classes (e.g., `GnForm` or `GnRowColumn`), and uses the same mechanisms for refinig its behavior (i.e., specializing virtual functions).

5.1 Defining Additional Callback Lists

A higher level of information hiding and locality is reached by mapping the callbacks of a composite widget's private components on a new public C++ callback list. A callback list is of class `GnInternalCallbackList`, which is a subclass of `GnCallbackList`, which is in turn a subclass of `GnLinkedList`. A new callback list should be defined as follows:

```
public:
  GnCallbackList &ValueChangedCallback() { return value_changed_callback; }
protected:
  GnInternalCallbackList value_changed_callback;
```

Clients access the callback list by an inline accessor function. The type exported to clients is `GnCallbackList`. This ensures that clients can only call list operations like adding or removing callbacks:

```
GnCallback *cb = ...

widget.ValueChangedCallback().Add(cb);
widget.ValueChangedCallback().Remove(cb);
widget.ValueChangedCallback().RemoveAll();
```

The class `GnInternalCallbackList` is internally used by the new widget class. It defines the additional operation `Execute(caddr_t)`, which calls any callback on the list (i.e., it calls `execute()` for any callback object on the list). This technique ensures that execution of callbacks is completely controlled by the widget class and may not be performed by clients. To preserve information hiding and data encapsulation, the new dialog class must add only non-public member functions to the callback lists of its subwidgets in order to prevent clients from simulating an execution of the callback list by calling those callback procedures directly.

6 Separating Presentational and Behavioral Aspects

The set of Motif widget classes could be divided into a number of subsets whose members share common interaction semantics. We give some examples:

- The push button, arrow button and drawn button widgets are all used to perform an action by pressing and releasing the mouse inside of them.
- The scale and scrollbar widgets are used to select a numeric value from a continuous range of values. A numeric value could also be entered through a text edit widget which is configured in such a way that only numerical values are accepted.
- There are several widgets realizing some kind of selection. Examples are the option menu widget, the row column widget configured either as a radio box or a toggle button group, and the list widget with its four different selection policies.

Programming with these widgets involves programmers in tasks dealing with presentational details. This means that a programmer not only has to specify *what* is to be presented, but also *how* it is to be presented. We explain this for the subsets mentioned above:

- All button types require an XmString for their label string, which must be created by the programmer. Here, the information to be displayed is a text string, and an additional character set defines how the string is presented. For a programmer, it would be much easier to provide both the character string and the character set as separate resources and let the button widgets create and free the XmString. The same as above holds for `XmList` widgets, where the item list must be given as a vector of `XmStrings`.
- The `XmList` widget requires a vector of XmStrings as its item list. These strings represent the objects (for example, personal records or file names) which could be selected through the list widget. The callback structure passed to a callback executed by a list widget contains a vector of indicees into the vector of XmStrings denoting the set of selected items. In order to be able to find the object for a certain index, the programmer must keep a table of associations between the objects and the XmStrings representing them.

  A first simplification would be if the programmer could pass the table of associations to the list widget instead of a vector of XmStrings. The callback structure could then contain a vector or some other kind of collection which contains (references to) the selected objects.

A further aid would be if the programmer could simply pass in the set of objects and let the list widget automatically compute the object's representation. Representational details should not be kept in the objects. They should in fact be determined by the list widget.

- An option menu also selects an item from a set of items (e.g., a color from a set of colors). As with the `XmList` widget, the programmer must compute an XmString for each object he wants to offer for selection. Furthermore, a push button widget must be created for each item.

Our aim is to reach a higher level of abstraction from presentation details for an application. The first step is to separate the specification of behavior from the specification of appearence. Using object-oriented techniques, this means that we must define two separate (abstract) classes, one responsible for the behavior and the other one responsible for the appearence. Then we use multiple inheritance to combine behavioral and presentational aspects in a concrete class. The concrete classes could be further subclassed in order to emphasize certain presentational aspects.

Given the above examples of the list and option menu widget, we express the functionality for selecting a number of items from a set of items in an abstract class. Then we define two concrete selection classes that inherit from this abstract class and from the `XmList` widget and the `XmRowColumn` widget, respectively. The concrete selection class using an option menu widget is then further subclassed under presentational aspects in a way that it uses bitmaps instead of XmStrings representing its items. Another concrete selection type might be realized through a character based I/O device, for example by using a *curses*-like package.

Applications can implement their functionality independently from the underlying presentation mechanisms by using only the subset of functionality that is defined in the behavioral part of a widget. The advantage is that interface components could be exchanged with minimal effort. This becomes even more important when the design of an application's user interface is aided by tools like an interface builder.

We give a more detailed description of the ideas sketched so far in the following sections by the example of the selection classes. The resulting inheritance tree is shown in figure 4. Abstract classes are shown in dashed rectangles. The pure Motif widget classes are shown in ovals. The concrete realizations of selection interaction types are shown in rectangles with rounded corners. The solid arrows denote an inheritance relationship with their tips pointing towards the base class.

6.1 Specification of an Abstract Selection Class

We model the behavior of a selection widget in the abstract class `GnSelection`. This class does not cover any presentational details, but already takes into account the fact that further subclasses must add this functionality.

6.1.1 Presenting Items

In general, a selection widget is used for selecting a number of *items* from a set of items. Any object might serve as an item in a selection widget. Because we have a single rooted class tree, an item is an object of class `GnObject`. An item is either represented by a text

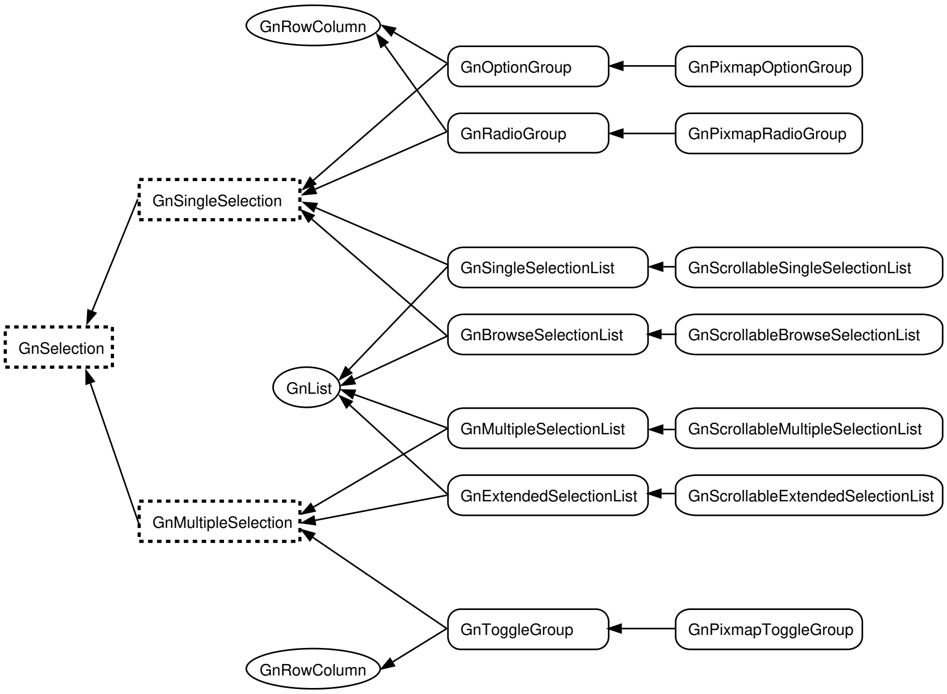

Figure 4: Abstract and Concrete Selection Classes

string or a bitmap icon. The concrete realization of a selection widget decides which kind of representation is actually used. For these purposes, the class `GnObject` defines virtual functions which deliver the information required to present an object:

```
class GnObject {
  public:
    virtual char *PrintOn(char *buffer, unsigned int buffer_length);
    virtual char *BitmapFileName()
        { return 0; }
    virtual unsigned char *BitmapBits(int &width, int &height)
        { width = 0; height = 0; return 0; }
  ....
};
```

It is important to notice that the representation information could be computed without knowing any details about the underlying I/O devices and window systems. The default implementation of the function `PrintOn()` prints the object's class name and address on the string buffer. The function `BitmapFileName()` returns a path name of a bitmap file. The default return value is a `NULL` string, indicating there is no bitmap representation available. The function `BitmapBits()` returns a bitmap's bits and dimensions and may be used in cases where the bitmap information is stored in memory.

In most cases, a class would not provide a bitmap representation for its instances. One possible canditate could be a class `PathName`, enabling files and directories to be displayed as icons in a graphical shell.

Items are added either one by one or in groups. In the first case, a client may optionally specify whether the item is initially selected and whether it is sensitive to input events. In the second case, the contents of a collection object (class `GnCollection`) are added. The item list may also be changed by removing single items, by inserting items at certain positions, and by exchanging the whole item list by providing another collection of objects. The description given so far leads to a first definition of the class `GnSelection`:

```
class GnSelection : public virtual GnObject {
  public:
    GnSelection();

    // Add an item
    GnObject &Add(GnObject &object, Boolean selected = False,
                  Boolean sensitive = True);
    GnObject &Add(GnCollection *items);

    // Remove an Item
    GnObject &Remove(GnObject &object);

    // Insert an item into the list
    GnObject &Insert(GnObject &object, GnObject &before,
                     Boolean selected = False, Boolean sensitive = True);

    // Manipulate the item list
    void Change(GnCollection &items);
};
```

Creating a selection object then is a three-step process: First, the selection object must be allocated. Then the items must be added, and last, the associated widget must be created:

```
GnSelection *selection = new GnRadioGroup;
selection.Add(new Color("red"));
selection.Add(new Color("green"));
selection.Add(new Color("blue"));
selection.create(parent, "selection");
```

Only the first statement depends on the concrete selection class (i.e., `GnRadioGroup`) being used. Anything else is coded by using the functionality inherited from the abstract base class `GnSelection`. If a programmer wishes to use another selection class, for example `GnOptionGroup`, he must change only a single statement, namely:

```
GnSelection *selection = new GnOptionGroup;
```

A selection widget notifies its clients by a callback whenever the selection changes. We define an additional callback list for this purpose (see chapter 5). In concrete subclasses, the callbacks of widgets associated with an items are filtered and mapped onto this callback list. When a callback is executed, a pointer to the selection object is passed as the call data argument to the callback function. Programmers may obtain access to the list of selected items, deselected items or even to all items by calling the call data object's member functions which return collection iterators for these lists. We extend the definition of `GnSelection` in the following way:

```
public:
  GnCallbackList &SelectionCallback() { return selection_callback; }
protected:
  GnInternalCallbackList selection_callback;

public:
  GnItemIterator & AllItems();
  GnItemIterator & SelectedItems();
  GnItemIterator & UnselectedItems();
  GnItemIterator & SensitiveItems();
  GnItemIterator & InsensitiveItems();
```

6.2 Subclassing under Behavioral Aspects

The selection interaction type are now further subclassed under behavioral aspects. The dominating characteristic here is whether a single selection or a multiple selection is realized. We model this difference in an additional abstract subclass `GnSingleSelection`, which defines functionality for accessing the currently and previously selected items:

```
    class GnSingleSelection : public GnSelection {
      public:
        GnSingleSelection();

        GnObject *SelectedItem() { return current_selection; }
        GnObject *History()      { return history; }

      protected:
        GnObject *current_selection;
        GnObject *history;
};
```

The single selection type may require that either exactly one element is selected at a time or that there is at most one selected element. We consider the first case to be used in most cases and therefore define a function which disables this feature:

```
public:
  void AllowNoSelection(Boolean);
```

A multiple selection type may either automatically notify its clients each time the selection changes or only once when a selection is complete. In the latter case, the user making the selection must explicitely tell the widget when a selection is complete, for example by double-clicking on the last selected item. We again model this behavioral difference by a boolean attribute of the multiple selection class.

```
public:
  void ImmediatelyNotify(Boolean);
```

6.3 Concrete Selection Classes

The classification given so far allows the specification of an application's user interface at an abstract level. For a real user interface, the selection classes must be further subclassed.

The concrete selection widget classes are derived by inheriting from both an abstract selection class and a (concrete) Motif widget class.

There are four subclasses realized using the Motif list widget, each with a different selection policy. The `XmList`-based selection widgets are the preferred choice in cases when the set of items changes dynamically and when the number of items is too large to be displayed all at once. For this reason, these widgets are further subclassed in order to provide scrolling capabilities (`GnScrollable...`).

Three selection widgets are based on the row column widget. The classes `GnRadioGroup` and `GnToggleGroup` both use toggle buttons for representing their items. The class `GnOptionGroup` is based on the option menu widget and requires an additional menu pane and push buttons for displaying its items. These classes are prefered when the number of items is small and fixed.

The selection widgets described so far all use text strings for representing their items. The selection widgets based on the row column widget have been further subclassed under presentational aspects (`GnPixmap...`) and use bitmaps to represent their items. The selection widgets based on the `XmList` widget could not be subclassed under this aspect because the list widget does not provide the functionality required for displaying bitmaps.

Other selection styles could be implemented using the abstract selection classes. For example, items could be displayed in a scrolled window by a combination of an icon and a text string. This technique is often applied by graphical shells to display files and directories.

Selection widgets could even decide not to visualize their items and to offer only facilities to select items from their invisible item set. For example, a regular expression could be entered through a text widget. The set of selected items is then determined by all items whose printed representation matches this expression. Although this selection style seems quite unusual, it is used for example in the Motif file selection box widget. Here an item is a filename and the set of items is formed by all files contained in a particular directory. The editable file filter may contain a regular expression that determines the subset of files that are displayed in a list widget and offered for selection.

6.4 Implementation Aspects

Since the discussion of all implementation details would be beyond the scope of this paper, we discuss only the most important things.

The first topic is what information a selection widget must store about its items. We already said that an item may be selected or unselected. An item may also be sensitive or insensitive to (mouse) input events. In order to be selectable, an item must be sensitive. These attributes should be accessible independently from the concrete realization of the selection widget, so the selection object must store them together with the item. It is also possible that an item is contained in multiple selection objects, having a different selected or sensitive state in each selection object. It is therefore necessary to keep this state information independently from the item. For this purpose, we define the supporting class `GnSelectionItem`, whose instances hold an item's address as well as its state information:

```
class GnSelectionItem : public GnObject {
  public:
    GnSelectionItem(GnObject *item, Boolean selected, Boolean sensitive);
```

```
        GnObject *Item()        { return item; }
        Boolean   IsSelected()  { return selected; }
        Boolean   IsSensitive() { return sensitive; }
      protected:
        GnObject *item;
        Boolean   selected;
        Boolean   sensitive;
};
```

Each selection object keeps a private list of `GnSelectionItem` objects. Each time an item is added to a selection object, a new selection item is created and inserted into this list. Changing the selected state of an item then simply means to change the `selected` attribute in the selection item associated with it. This is sufficient for an abstract selection object, but in a concrete one, additional actions must be taken to visualize these changes.

To solve this problem, we could consider to make the `Add()` function virtual and refine it in a concrete selection subclass. The drawback to this solution is that the refined version must repeat the functionality to insert the new item into the selection item list. A solution which leads to a higher degree of code reuse is to define a protected virtual function `AppendItem()` that visualizes changes in the set of items. The `Add()` function would then look like:

```
GnObject &GnSelection::
Add(GnObject &object, Boolean selected, Boolean sensitive)
{
    GnSelectionItem *item = new GnSelectionItem(object, selected, sensitive);
    selection_items.append(item);

    // Visualize change
    AppendItem(item);
}
```

The `AppendItem()` function is introduced as pure virtual in the class `GnSelection`:

```
  protected:
    virtual void AppendItem(GnSelectionItem *item) = 0;
```

A concrete selection class must always refine (i.e., implement) this function. For example, the class `GnSingleSelectionList`, which is based on the `XmList` widget, implements it in the following way:

```
void GnSingleSelectionList::
AppendItem(GnSelectionItem *item)
{
    char buf[100];
    XmString str = XmStringCreateLtoR(item->Item()->PrintOn(buf, 100),
                                      XmSTRING_DEFAULT_CHARSET);

    if( item->IsSelected() )
        XmListAddItem(get_widget_id(), str, 0 );  // 0 = last position
    else
```

```
        XmListAddItemUnselected(get_widget_id(), str, 0 );

    // Free string because XmList widget copies it.
    XmStringFree(str);
}
```

The mechanism described above separates functionality which can already be provided by the abstract selection classes from those which can only be realized in a concrete subclass. As a consequence, the functions defining the behavior of a selection object that is accessible for clients are all non-virtual. They call protected virtual functions which realize presentational aspects.

Another topic we address here is how the framework concepts described in the previous chapter are used for implementing the concrete selection classes. Since all concrete selection classes inherit from both the abstract selection classes and from one of the widget classes, they must refine the functions `before_create()`, `after_create()` and `create_widget()` to visualize their items and to create all required subwidgets. Inside these functions, they can access the list of selection items, which provides all necessary information. For example, a simplified version of the function `before_create()` of the class `GnSingleSelectionList` configures the list widget and creates a vector of `XmString`s from the list of selection items:

```
void GnSingleSelectionList::
before_create(GnWidget *parent)
{
    setR_selectionPolicy(XmSINGLE_SELECT);

    XmStringTable  vec = new XmString[selection_items.length()];
    GnItemIterator it(selection_items);

    for(int i = 0, it.start(); ! it.off(); i++, it.forth() ) {
        char buf[100];
        vec[i] = XmStringCreateLtoR(it.Item()->PrintOn(buf, 100),
                                    XmSTRING_DEFAULT_CHARSET);
    }
    setR_items(vec);
    setR_itemCount(i);
}
```

The concrete selection classes must map callbacks of their widgets to the `SelectionList` callback list. This is relatively easy for the `XmList` widget because the callback structure passed to the callback functions contains the positions of the selected items, which can easily be mapped into the list of selection items. The same task requires more programming effort for a `GnOptionGroup` widget, because each selection items is visualized by its own push button widget. In this case, we must attach a callback to every button. The callback's client data then is a reference to its associated selection item.

7 Further Ideas and Conclusions

The framework should be extended to an application framework to simplify application development. We have already implemented the prototype application framework

GINA [BB91] based on ideas from ET++ and MacApp. To avoid reinventing the wheel, we are currently investigating ways to reuse parts of ET++ and other class libraries.

The concept of *pluggable views* would be another direction for an abstraction from presentational details. Additionaly, we must investigate universal construction principles for interactive graphical applications.

We have shown that our framework helps in stucturing and modularising the user interface part of an application. Using object-oriented techniques to encapsulate the behavior of widgets increases reusability of interface components. The framework does not provide a fully-fledged user interface toolkit, which would require the reimplementation of the Motif and X toolkit and Xlib in C++. Here, the InterViews toolkit [LCI+91] and ET++ could serve as models.

Possibly, the feature mostly missing in the framework is the capability to set a widget's new resources through the resource database, because this would avoid numerous recompilations when fine-tuning of the user interface takes place. Adding this functionality would require the implementation of a meta-object protocol for C++, where classes, objects and their members and types could be referenced by their names. As an intermediate solution, new derived widgets could fetch initial resource values in their constructor, using the Xt resource manager's capabilities.

The implementation of a widget that integrates cleanly and fully into the existing Motif widget set is not possible. Nevertheless, we can use the framework as an experimental platform and recode the final product version of a widget in C using the Motif and X toolkits.

References

[BB91] Robert Babatz and Andreas Bäcker. *"GINA++ – The Generic Interactive Application for C++ and OSF/Motif, Reference Manual, Version 1.1"*. GMD, August 1991.

[Ber91] Thomas Berlage. *"OSF/Motif, Concepts and Programming"*. Addison Wesley, Wokingham, England, 1991.

[ES90] Margarret A. Ellis and Bjarne Stroustrup. *"The Annotated C++ Reference Manual"*. Addison Wesley, 1990.

[Fek91] Jean-Daniel Fekete. *"WWL, a Widget Wrapper Library for C++, Version 1.2"*. Laboratoire de Recherche en Informatique, Orsay Cedex, France, February 1991.

[Jon89] Oliver Jones. *"Introduction to the X window system"*. Prentice-Hall, 1989.

[Kee88] Sonya E. Keene, editor. *"Object-Oriented Programming in Common Lisp"*. Addison Wesley, Reading, MA, 1988.

[LCI+91] Mark A. Linton, Paul R. Calder, John A. Interrante, Steven Tang, and John M. Vlissides. *"InterViews Reference Manual, Version 3.0"*, 1991.

[Mey88] Bertrand Meyer. *"Object-Oriented Software Construction"*. Prentice Hall, New York, 1988.

[SBBB91] Mike Spenke, Andreas Bäcker, Thomas Berlage, and Christian Beilken. "GINA – A User Interface Development Environment Based on OSF/Motif". In *Proceedings of the First International Lisp Users and Vendors Conference*, Gaithersburg, October 1991.

[Sch86] K. J. Schmucker. *"Object Oriented Programming for the Macintosh"*. Hayden, Hasbrouck Heights, New Jersey, 1986.

[SRM90] K. Seetharaman, C. F. Rei, and B. R. Montague. *"A C++ Binding for OSF/Motif"*. University of Lowell, Lowell, MA, April 1990.

[WGM89] A. Weinand, E. Gamma, and R. Marty. "Design and Implementation of ET++, a Seamless Object-Oriented Application Framework". *Structured Programming*, 10(2), June 1989.