Encapsulating a C++ Library

Mark A. Linton
Silicon Graphics Computer Systems

Abstract

Encapsulation is the hiding of internal details from the user of an abstract data type, class or module. Encapsulating a class library requires more than combining a set of classes that are encapsulated individually. Libraries need to hide the details of how objects are created because some kinds of objects may be represented by composites as opposed to single instances. Implementation classes and members also must be hidden from the user of a library, even if this hiding conflicts with the user's desire to reuse library code.

In this paper, we present the encapsulation techniques used in the InterViews 3.1, a C++ class library for building user interfaces. These techniques have been formulated from the experience of building and releasing InterViews over a period of several years.

1 Introduction

Building and maintaining a class library is more difficult than building and maintaining an application because the number of external interfaces is much larger. For example, consider a library of 20 classes, with an average of 5 public functions per class and 20 lines of code per function. This library has 100 external interfaces for 2,000 lines of code. In contrast, a 2,000-line application would typically have closer to 10 or 20 external interfaces. This example matches our actual experience with the InterViews class library[2] and the Dbx debugger[5]. Both InterViews and Dbx are about 25,000 lines of code, but InterViews has an order of magnitude more external interfaces.

To reduce the complexity of building and maintaining a class library, we must encapsulate as much of the library implementation as possible. Encapsulating a library is more than encapsulating each class individually. In particular, we need to hide the details of object creation to give the implementation the freedom to create an instance from a library class, clone an existing instance, or create a composite of several instances from several classes.

In this paper, we present three encapsulation techniques that we have applied in version 3.1 of the InterViews class library. The first technique is to create a layer of abstraction for object creation and has the most significant effect on users of the library. The second technique is to hide implementation-oriented class members even if the members could be useful to library user. The third technique is the use of implementation classes to hide external dependencies. We also discuss the effect of these library encapsulation techniques on documentation.

2 Object creation

The primary motivation for abstracting library object creation is to allow the implementation to change the way an object is instantiated without effecting the library user. For example, suppose the user wishes to write code to create a push button object. The library could implement a push button class, or a push button could be created by composing an object that implements button input behavior with an object that implements the output behavior associated with a push button.

The result of abstracting object creation is that library users *do not call constructors*. Instead, a library user calls a function to create an object. This function could be a global or static member function; however, we use a virtual member function on another class. This class contains functions for creating a variety of related kinds of objects. We call a creator class a “kit”; such a class is also sometimes called an “object factory”.

In the push button example, using a push button class one might write the following code to create an instance:

```
Button* b = new PushButton("hi mom!", button_callback);
```

Using a kit, one instead would write

```
Button* b = kit.push_button("hi mom!", button_callback);
```

A critical assumption here is that a PushButton class adds no protocol beyond what is supported by the Button class. From the library designer’s perspective, the existence of any class that does not define new protocol is an implementation decision and should be hidden from the library user.

Kits can help organize a library for the user, providing a higher-level structure to the classes. For example, InterViews 3.1 provides kits for stylistic components, such as buttons and menus, and layout components, such as boxes and glue for document formatting. We intend to add more kits in the future, but we anticipate the number of kits will be quite small (5-10) compared to the number of classes (around 100).

Kits also have four benefits with respect to class definitions:

- Classes provided for convenient construction can be eliminated.
- Classes that add no protocol can be hidden.
- Constructor overloading and default parameters can be avoided.
- Accessing existing objects is simplified.

In the remainder of this section, we use examples from InterViews to demonstrate the value of kits. We also show how making the kit member functions virtual allows us to provide alternate implementations of a kit.

2.1 Eliminating classes

An important feature of InterViews is high-level support for sophisticated layout. In particular, we have extended the TeX "boxes and glue" formatting model[4] from static pages to all user interface components. TeX boxes and glue have a horizontal or vertical orientation. For example, a "vbox" arranges components top-to-bottom and "vglue" has a specified height and vertical stretchability or shrinkability.

A user of the InterViews library would like to be able to create a vglue object. Our original approach was to define a VGlue class in C++, which derived from a base Glue class. For convenience, we provided both a VGlue constructor that specified only the natural size (assuming infinite stretchability) and a constructor that defined the natural size, stretchability, and shrinkability.

The VGlue class added no protocol to Glue; indeed the only member functions defined were the constructors. Furthermore, the implementation of the constructors simply passed the appropriate parameters to the parent Glue constructor.

InterViews 3.1 provides a LayoutKit class that has an overloaded set of vglue member functions. The functions return simply a pointer to a *glyph*, which is the base class for objects that are allocated screen space.

Using the layout kit, there is no longer any need for the VGlue class at all, as the layout kit member function can call the Glue constructor directly. Figure 1 shows the VGlue constructor code and corresponding layout kit member functions.

An unexpected effect is that the kit approach is *more* efficient than the class approach. The runtime performance is equivalent. Although the kit call has an extra level of indirection over a direct constructor call because the vglue member function is virtual, the VGlue constructor must do some work to set up the virtual function table. Both approaches call the Glue constructor.

The savings from the kit approach is the code size necessary to define the VGlue constructors and the VGlue virtual function table. A knowledgeable compiler might be able to eliminate the VGlue code and table, but this optimization comes for free when using a kit.

```
VGlue::VGlue() : Glue() { }
VGlue::VGlue(Coord natural) : Glue(Dimension_Y, natural, fil, 0.0, 0.0) { }
VGlue::VGlue(
  Coord natural, Coord stretch, Coord shrink
) : Glue(Dimension_Y, natural, stretch, shrink, 0.0) { }
```

```
Glyph* LayoutKit::vglue() { return vglue(0); }
Glyph* LayoutKit::vglue(Coord natural) { return vglue(natural, fil, 0); }
Glyph* LayoutKit::vglue(Coord natural, Coord stretch, Coord shrink) {
  return new Glue(Dimension_Y, natural, stretch, shrink, 0.0);
}
```

Figure 1: VGlue class and layout kit implementations

Introducing the LayoutKit class allowed us to remove 25 classes from the InterViews library. Figure 2 shows the effect on the class library diagramatically, showing the old class hierarchy with obsolete classes in italics.

On the other hand, the kit approach could continue to use a VGlue class if there were some execution or memory usage benefits. For example, LayoutKit::vglue() could return an instance of a special class that does not store any size information–it simply returns fil (infinite) stretchability and zero for everything else. The important idea is that the kit isolates the library user from the presence or absence of the VGlue class.

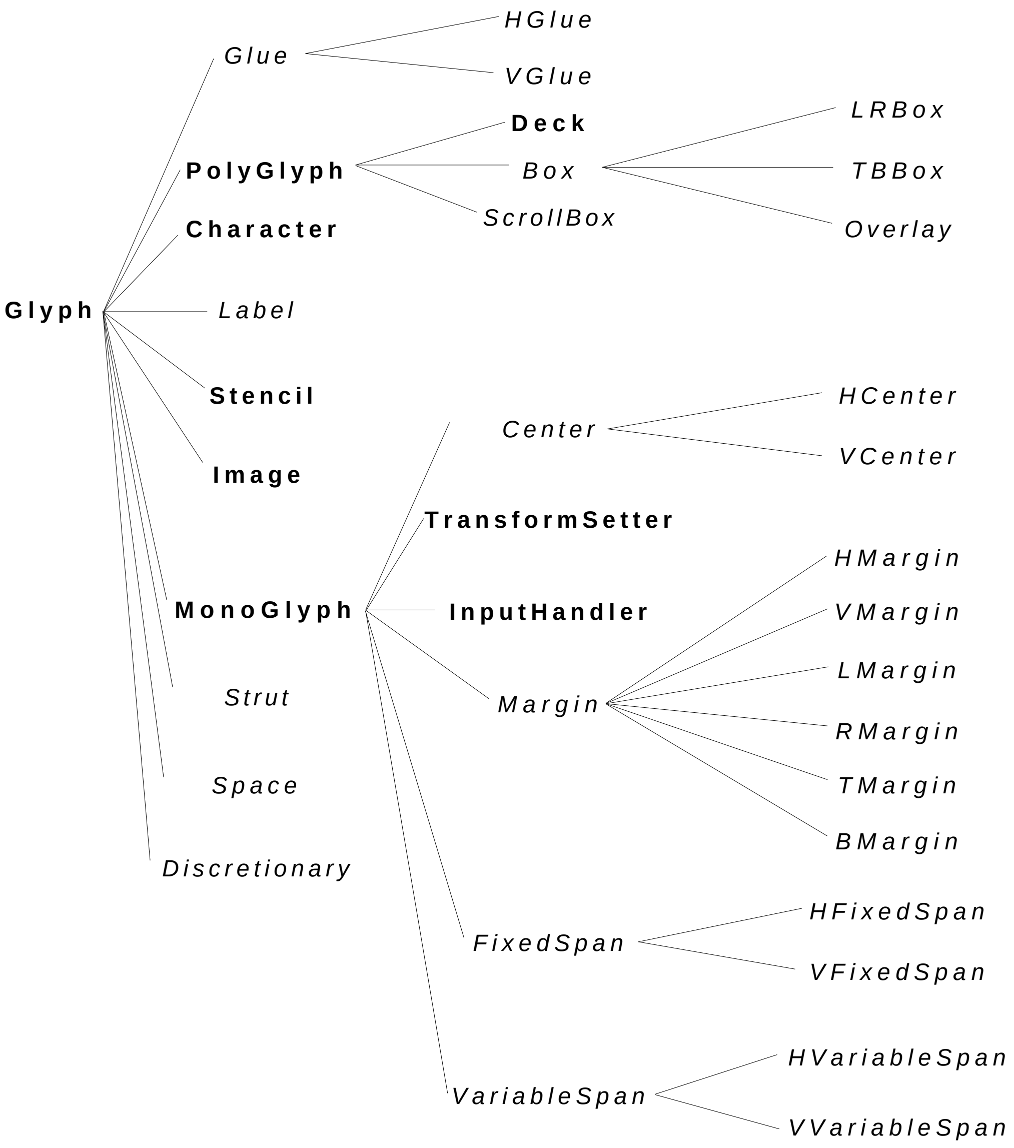

Figure 2: Layout classes

2.2 Hiding classes

Even when the use of a kit cannot eliminate a class, the class may become irrelevant to the library user. In this case, the library developer need not support or document the class. An example from the InterViews layout kit is the discretionary class. In TeX, a discretionary is an object that defines a “penalty” for generating a line break and a potentially different appearance in the case that a line break occurs. For example, a paragraph break is a “good” place to break, white space is an “ok” place to break, and a hyphenation point between characters in a word is a relatively “bad” place to break. If a break occurs on white space in justified text, then the white space becomes zero-width.

The LayoutKit class provides member functions for creating discretionaries. The glyph base class defines a *compose* member function that returns a potentially different glyph in the case of a break. The LayoutKit implementation of discretionaries therefore must use a new glyph subclass that implements compose; however, this subclass need not be visible to the library user.

The key characteristic of classes that can be hidden is that they do not add any protocol beyond their base class. Whether a class adds protocol often depends on whether associated state is editable. For example, we can eliminate Glue because there are no operations to change the stretchability of an existing glue object. An application will create a new glue object instead of changing an existing one. In contrast, we cannot eliminate TransformSetter because we want to be able to modify its transformation matrix without creating a new object.

The disadvantage of hiding classes is that users cannot use subclassing to reuse the behavior of these classes. The library designer must compare the cost of maintaining a class that is a public part of the library against the value of subclassing. In the case of the Discretionary class, the implementation is so simple (under 100 lines of code) that the subclassing value is negligible.

2.3 Defining constructors

In C++, kit member functions have an advantage over constructors in that they can use different names when a call might otherwise be ambiguous. Before defining a layout kit for InterViews, this problem arose when we wanted to be able to fix the size of an object in either one or both dimensions. We had defined the following class:

```
class FixedSpan : public MonoGlyph {
public:
  FixedSpan(Glyph*, DimensionName, Coord span);
  FixedSpan(Glyph*, Coord x_span, Coord y_span);
  // other functions
};
```

DimensionName is an enumerated type with values Dimension_X and Dimension_Y. We found that some compilers could not resolve the calls

```
FixedSpan(g, Dimension_X, 10.0),
FixedSpan(g, 10.0, 10.0)
```

because of the possibility of an implicit conversion from an integer to a float. The only solution with a class-based approach (other than forcing the user to put in a cast) is to define additional classes, such as FixedXSpan and FixedYSpan. Using LayoutKit, we can simply define member functions with different names when overloading resolution is not sufficient:

```
Glyph* fixed_span(Glyph*, Coord x_span, Coord y_span);
Glyph* fixed_span_dimension(Glyph*, DimensionName, Coord span);
```

A kit can maintain state that must be passed to constructors but is inconvenient for users to pass through parameters. We use this ability to maintain a current style in the kit that creates buttons, menus, and scrollbars. However, this state could also be used to specify what location parameter to pass to *operator new* without requiring that all object creation calls be aware of a location option.

Finally, kits also avoid ambiguity problems associated with default parameters. Because the user creates objects solely through a kit, the need for constructors with parameters is reduced to the case where an object or some data associated with an object is immutable.

2.4 Extensible modules

Kits are essentially modules–classes with a single instance in C++. We could implement a kit as a class with static member functions, but by making the member functions virtual we can make it possible to choose from several implementations of a kit at runtime.

For example, the InterViews 3.1 WidgetKit defines functions for creating common user interface components such as buttons, menus, and scrollbars. WidgetKit is an abstract base class; subclasses create objects that support a particular look-and-feel. For example, the MFKit implements a subset of the OSF/Motif look-and-feel. Similarly, an OLKit could implement the OpenLook look-and-feel. This approach to supporting multiple styles is similar to the functionality of the Solbourne OI toolkit[1], though OI uses global function calls instead of virtual calls on a kit class.

WidgetKit defines a single static member function called *instance* that returns a pointer to the kit object. The first time this function is called, it will create the object and store a pointer to it in a static data member. Subsequent calls simply return the stored pointer. WidgetKit::instance creates the appropriate subclass depending on a user or system-defined property.

A kit also normally defines a protected member function to assign the kit instance pointer directly. Thus, a user can define a subclass, redefine any virtual functions as desired, and create an instance of that class directly before the kit is first accessed.

This technique is applicable to other one-of-a-kind objects. The InterViews Dispatcher class, which routes input and timer events, also defines a static member function for returning the instance and a function for setting the instance. All other dispatcher functions are virtual, allowing the library user to redefine the behavior of the dispatcher if so desired.

2.5 Accessing objects

For objects such as colors and fonts, application code often wants to access an existing object instead of always creating a new one. For example, one might wish to find the color named “red”. If the name is known but a color object has not yet been created, the object should be created. If the name is unrecognized, then one should not receive a valid object.

This functionality does not lend itself intuitively to using constructors because the caller is not creating an object so much as looking for an object. Constructors also cannot easily return an invalid object. Kits provide the opportunity to lookup an object by name, create one if necessary, and return a nil pointer if the object cannot be created.

3 Hiding class members

Encapsulation of an individual class involves a tradeoff between what is exposed for potential reuse and what is hidden for potential future change. In an application, the use of a class is often limited and the cost of change relatively low. In contrast, a class in a library has much more widespread use and the cost of changing a public or protected interface is much greater. This dichotomy causes the library user, who is developing on an application, to want to reuse as much of the library implementation as possible. The library implementor, on the other hand, wants to hide as much as possible.

In the past, InterViews exposed implementation through both protected data and functions. Our current strategy is to make *all* data members private and to make the "default" access to functions be private. We treat "over-protection"–access that is private when it should be public or protected–as an enhancement request that is straightforward to provide. The opposite–access that should be changed to private–cannot be achieved without the possibility of breaking existing code.

3.1 Data members

Making data members private gives the implementation freedom in moving data to another object or in changing a representation. Access functions for reading and writing allow users to store and retrieve the data.

An example from InterViews demonstrates the benefits of using access functions. The Telltale class defines an appearance that depends on a set of flags, including whether the telltale is disabled or enabled, highlighted or not, and chosen or not. The first Telltale implementation stored each flag in a separate bit field data member of the Telltale object. Access functions were provided to test the current state of the Telltale, such as whether it was enabled. Figure 3 shows the original Telltale class definition.

In the most recent implementation, the flags are stored in a separate object so that the state can be shared among several objects. The Telltale access functions have been modified to access the state indirectly, and code that uses the original Telltale interface continues to work without modification. Figure 4 shows the new Telltale class definition and the implementation of the *enabled* functions.

3.2 Protected vs. private

Given the data members of a class are private, one still must decide whether to provide access functions for the data and whether those access functions should be public or protected. Permitting access promotes the greatest reuse, but also exposes the implementation.

```
class Telltale : public Glyph {
public:
   // constructors, other public functions
   void enabled(boolean);
   boolean enabled() const;
private:
   boolean enabled_ : 1;
   // other data, private functions
};
```

Figure 3: Original Telltale definition

```
class Telltale : public MonoGlyph {
public:
   // same public interface
private:
   TelltaleState* state_;
   // other data, private functions
};

void Telltale::enabled(boolean b) {
   state_->set(TelltaleState::is_enabled, true);
}

boolean Telltale::enabled() const {
   return state_->test(TelltaleState::is_enabled);
}
```

Figure 4: New Telltale definition

An early mistake we made was to define implementation functions as protected. From a maintenance point of view, protected is really no different from public. In either case, a class user may become dependent on the function. Furthermore, we found that protected encouraged subclassing for greater access even in the cases where instancing would have been more appropriate.

Our current approach is to define functions as "private" by default. If the function is needed externally, then it should probably be public. We use protected primarily for constructors of abstract classes.

4 Implementation classes

The representation and implementation of several InterViews classes depends on the X Window System, but the interface to these classes is independent of X. We introduce an extra level of indirection to isolate the class interface from this implementation dependency. This approach is similar to the requester and implementor separation in CommonView[3], but in our case the interface class directly implements the external protocol.

```
class WindowRep;
public:
   // public functions
private:
   WindowRep* rep_;
};
```

Figure 5: Window class interface

For example, the definition of the InterViews window class is shown in Figure 5. The WindowRep class definition depends on X data types, which are not visible to the Window class user. WindowRep's data members are public, which violates one of our principles for library classes. However, the WindowRep class is not visible to the library user and the exposure of WindowRep simplifies the implementation of other X-dependent classes. Of course, we would prefer a more abstract interface but library internals are similar to an application in that the scope of a change is limited.

5 Documentation

The same goal for code that uses a class library holds for documentation: changes in the library implementation should not require changes to the documentation. This approach means that class documentation is a *subset* of the public and protected members defined in a class interface. In particular, functions that are present only for implementation reasons should not be documented.

The most obvious example of a public function present solely for implementation is a destructor, which is typically defined or not defined depending on whether an object contains pointers to other objects that might need to be deallocated. More subtle examples can occur for virtual functions that are defined or inherited depending on the implementation of the class. For example, the InterViews base class glyph defines a virtual function *undraw* to notify an object that its window area has been allocated to another object. A class will define undraw if information associated with its window position is cached. Thus, adding or removing caching affects whether the undraw function is present in the class' public interface, yet does not change the semantics of the object.

6 Conclusions

An encapsulated library of classes is not the same as a library of encapsulated classes. Through examples from our experience with InterViews, we have shown the benefits of hiding the implementation of object creation using a layer between the library user and a potential constructor call. Individual library classes have a greater need for protection than application classes. A library implementor should therefore make members private by default and treat a user's desire for a protection change as an enhancement request.

Implementation classes add a level of indirection to avoid pollution of a library user's namespace. In addition, this level of indirection eliminates the need to recompile the user code when the implementation class changes.

To avoid implementation dependencies, class library documentation is slightly different from the C++ public and protected interface. Functions that are present solely because of the implementation, such as related to memory management or caching, are not present in the user-level library documentation.

References

[1] G. Aitken. OI: A Model Extensible C++ Toolkit for the X Window System. *Proceedings of the 4th Annual X Technical Conference*, Boston, Massachusetts, January 1990.

[2] P. Calder and M. Linton. Glyphs: Flyweight objects for user interfaces. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Snowbird, Utah, October 1990, pp. 92-101.

[3] F. Dearle. Designing Portable Application Frameworks for C++. *Proceedings of the Second USENIX C++ Conference*, San Francisco, California, April 1990, pp. 51-61.

[4] D. Knuth. *The TeX Book*. Addison-Wesley, Reading, Massachusetts,. 1984.

[5] M. Linton. The Evolution of Dbx. *Proceedings of the Summer USENIX Conference*, Anaheim, California, June 1990, pp.211-220.