Building user interfaces with shared objects

Building User Interfaces with Shared Objects

Paul Calder **Upasak Weston**

Discipline of Computer Science
The Flinders University of South Australia
Email: {calder,weston}@cs.flinders.edu.au

Abstract

A whole-hearted use of object-oriented techniques to build interactive application interfaces implies that objects should be used for even the smallest components in the interface. This approach often leads to a straightforward implementation that simplifies the programmer's task, but it can be costly in its use of computer resources. This paper explores the idea of using sharable objects to reduce some of these costs, and it discusses design techniques for making objects sharable.

Sharing reduces resource usage in two key ways: it reduces the number of object instances needed to represent application data, and it eliminates some of the structural redundancy inherent in a straightforward use of objects to represent views of hierarchical data. Our approach to making objects sharable is to define dynamic object protocols that reduce the amount of use-specific data that must be stored within object instances.

Keywords: Lightweight objects, shared objects, user interfaces, glyphs

1 Introduction

This paper focuses on the use of objects to build applications with interactive graphical interfaces. In particular, we argue that an object-oriented approach offers the application programmer considerable leverage, but that careful attention to object design is needed if the approach is to be practical.

Traditionally, object-oriented techniques have resulted in applications that are costly in their use of computer resources; often such applications run more slowly and use more memory than similar applications implemented with other techniques.

We have investigated the idea of using sharable objects to reduce an application's resource usage. Sharable objects reduce memory requirements by reducing the number of object instances, and they improve performance by eliminating some of the structural redundancy common in applications that use objects to represent views of hierarchical data.

Section 2 describes the use of object-oriented techniques to build interactive applications, and it outlines the advantages and disadvantages of the approach. Section 3 shows how sharable objects can reduce some of the cost of using objects; section 4 describes techniques that we have used to promote object sharing. In particular, section 4 shows how dynamic object protocols increase the opportunity for sharing by reducing the amount of use-specific data stored in objects. Section 5 presents some results that demonstrate the success of the techniques in the object-oriented implementation of a document editor. Finally, section 6 draws some conclusions and describes ongoing work in the area.

2 Building with objects

Object-oriented techniques simplify the construction of applications where objects can directly represent application-specific data. In particular, programs with interactive user interfaces are often suited to object-oriented implementation because program-level objects are a natural representation for the objects that the user manipulates (Calder 1993).

For example, an object-oriented implementation of an interactive drawing editor could use objects for the lines, circles, and polygons that comprise the drawing, while an object-oriented implementation of a document editor could use objects for the characters and words of a document.

But to gain full benefit from the object-oriented approach, a programmer must be able to use objects for even the smallest components that the user manipulates; to reflect the find-grained structure in some applications, programmers may need hundreds of thousands of objects. This approach can dramatically simplify an implementation, but it will be practical only if objects are simple and cheap.

2.1 The object-oriented approach

Applications that use an object-oriented approach assemble hierarchies of objects to represent application data. Actions that modify the data, such as operations that insert new data or change the attributes of existing data, can be implemented as operations on objects.

To use this technique, an application programmer creates a base class that defines a common object protocol, then derives subclasses that implement the methods of the protocol according to the needs of particular types of components. For example, an object-oriented drawing editor might define a base class for graphical objects with methods for drawing, picking, transformation, and attribute changes. Subclasses that represent particular types of graphical objects, such as rectangles and circles, would implement the methods in different ways.

2.2 Advantages of objects

An object-oriented approach aids programmers in three key ways: it offers a straightforward implementation technique, it promotes object reuse through

```
TBBox* page = new TBBox();
LRBox* line = new LRBox();

int c;
while ((c = getc(file)) != EOF) {
  if (c == '\n') {
    page->append(line);
    line = new LRBox();
  } else {
    line->append(new Character(c));
  }
}
```

Figure 1: Code for simple file text view

composition, and it facilitates efficient incremental update following changes to application data.

An object-oriented implementation is straightforward because the application code can treat all objects belonging to a particular class hierarchy in the same way. In a drawing editor, for example, the application can apply the same transformation (rotate, for example), to any graphical object in the drawing simply by calling the appropriate common method; each object will respond in the way characteristic of its particular type.

In effect, the programmer's task is reduced to one of composition: once the objects have been designed, the programmer can compose object instances into compound objects that, in turn, can be further composed. This characteristic has encouraged the development of libraries of predefined object classes (toolkits) that programmers can use to build parts of their applications. Toolkit designers define the protocols for a class of objects and provide a range of common object subclasses; programmers simply choose which toolkit objects they need, then arrange them in application-specific ways.

The toolkit approach has been widely adopted in the area of user interfaces. For example, the X Toolkit (Scheifler & Gettys 1986, Swick & Weissman 1988), the Andrew Toolkit (Weinand, Gamma & Marty 1988), ET++ (Weinand et al. 1988), and InterViews (Linton, Vlissides & Calder 1989, Linton, Calder, Interrante, Tang & Vlissides 1991) all use an object-oriented programming model to provide common user interface objects such as on-screen buttons, menus, and scrollbars.

InterViews, which is written in C++, takes the toolkit approach further than most other toolkits by defining lightweight objects, called glyphs, that applications can use to build views of fine-grained application data (Calder & Linton 1990). For example, Figure 1 shows how an InterViews application programmer might build a view of the contents of a text file by assembling glyphs that represent individual characters. In addition to the Character glyphs, the example uses composite glyphs to represent the structure of the text. An LRBox is a composite that arranges its components left-to-right, while a TBBox arranges its components top-to-bottom. The code iterates through the file, creating a character glyph for each character, an LRBox to arrange the characters in each line, and a TBBox to arrange the lines on a page. Figure 2 shows the resulting object structure.

The final benefit of the object-oriented approach stems from its support for incremental update following change. Objects can store information about their use that allows them to short-circuit potentially expensive update operations. For example, InterViews glyphs store information about their visible bounding region; when part of a view needs to be redrawn (after a change to the data, perhaps), a glyph need not

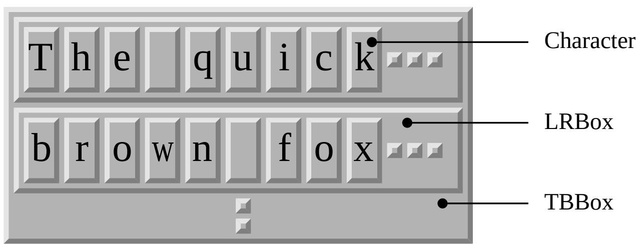

Figure 2: Text view object structure

redraw itself if its bounding region does not intersect the damaged area.

If a graphical component is changed in a drawing editor, for example, only those glyphs that represent components overlapping the changed component need redraw themselves. For drawing editors (and for many other interactive applications), typical operations affect only a few objects in a hierarchy. Consequently, techniques for incremental update can considerably reduce the amount of work an application needs to do following changes to its data.

2.3 Disadvantages of objects

Applications implemented with object-oriented techniques have typically used more computer resources than those implemented with other techniques. This extra cost stems from two main sources: the inherent run-time implementation overhead associated with any use of objects, and the replicated structures that arise from separate data and view objects.

Earlier object-oriented languages used expensive run-time method dispatch techniques, which made object method calls many times more expensive than simple function calls. However, the method dispatch mechanism used by current C++ compilers adds little cost. In our experience, the performance difference is not noticeable for interactive applications.

The memory overhead for method dispatch is more critical. A C++ object, for example, typically requires an extra word of memory to store a pointer to its method lookup table (more if it uses multiple inheritance). For large objects, this overhead is not significant, but for small objects, an extra word of storage per object can result in a substantial overall memory size increase.

Many interactive applications follow a data-view structural model, where the application data and its views are represented by separate objects. This model, first popularised in the Smalltalk MVC (model/view/controller) paradigm (Krasner & Pope 1988), is attractive because it allows applications to define several different views (or even several different *kinds* of views) for the same application data. Figure 3 shows the idea. The application data objects keep a record of their dependent views; when a data object changes, it notifies its views, which update their appearance to reflect the new state of the data.

The data-view model works well if the data and its views are composed from only a few, relatively heavyweight objects. However, when applications build hierarchies containing many thousands of objects, the overhead of storing dependency information and the cost of the notification and update can become excessive. For example, if a document editor represented its data as a hierarchy of character objects and its views as a hierarchy of dependent character view objects, then the total memory cost of the application and view objects could be many times the memory cost of the data itself.

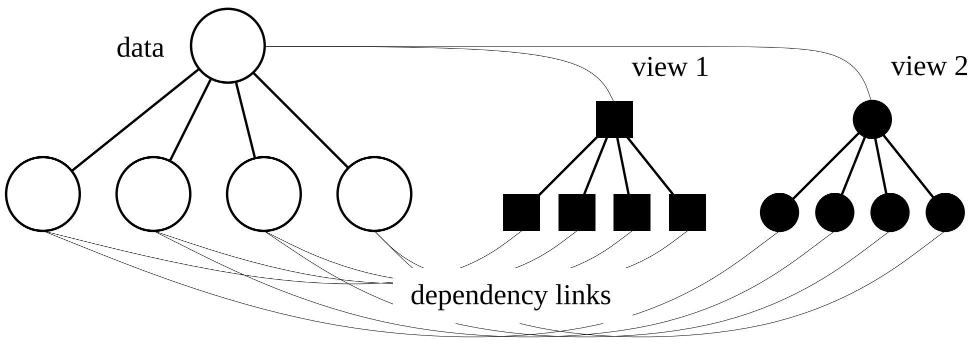

Figure 3: Data-view object structure with dependency links

3 Sharing: reducing the cost

Our approach to making objects cheap to use is to make them sharable. This approach has two main benefits: it reduces the number of object instances needed, since a single object instance can be used in all the places that object is needed; and it eliminates much of the structural redundancy in a data-view model since a single object can play the roles of both data and views.

3.1 Reducing object counts

For many applications, a user manipulates only a few kinds of objects. For textual data, for example, users manipulate characters; even if a document is large, the total number of different character objects will remain small.

For such applications, a single sharable object can represent a primitive data object each time it appears. For example, the same object can represent a particular character each time it appears in a document. This technique reduces the total cost of the objects by reducing the number of object instances needed. In effect, the cost of an object is amortised over the number of shared uses. For applications that exhibit a high degree of sharing, the reduction in memory size can be significant. Section 5 describes an experiment that used shared objects to represent characters in a document editor.

3.2 Eliminating replicated structures

One way to reduce the overhead of replicated data and view hierarchies is to use objects that can serve as both data and view; then a single object hierarchy will suffice.

This technique will result in fewer objects (since a single object replaces a data object and all of its views) and smaller objects (since the data object no longer needs to record information about its dependents). However, it will only succeed if the structures of the data and view hierarchies are identical and if a single protocol can be devised that suits the needs of both data and view.

For applications we have considered to date, these conditions are often satisfied. For example, drawing editors that allow multiple views of the data usually support only one type of view. Each view is structurally identical, although it may differ in size or view a different part of the document. For such applications a single shared hierarchy of objects can represent each view. Differences in the appearance of views can be accommodated as differences in the graphical viewing parameters of the view.

The ability to share objects does not completely replace the need for an explicit data view model, and it does not preclude such a model where it is appropriate. For example, an application that builds views that are structurally different from the data will still require a mechanism to maintain the view when the data changes. We are investigating ways of representing such views by making some of the structural information dependent on characteristics of those views.

4 Making objects sharable

To see how objects can be made sharable, it is important to recognise the difference between information that defines an object instance and information that defines its use. Instance-specific information distinguishes one object from another similar object; use-specific information distinguishes one use of an object from another use of the same object. The key to making an object sharable is to eliminate the need to store use-specific information in the object.

4.1 Dynamic object protocols

Information that objects need to define their behaviour can come from either of two sources: it can be stored within the object itself, or it can be passed to the object as part of the information provided in its method calls. Information that is stored within the object is static: it lives as long as the object lives. Information that is passed along with a method call is dynamic: it lives only for the duration of the call.

The primary technique that we use to make objects sharable is to make the object protocols dynamic. That is, we design object protocols that pass information to objects as needed, rather than requiring the objects to store the information themselves. If two objects are identical in every respect, that is if they each contain exactly the same information, then a single instance can be used for both. A dynamic protocol promotes sharing because it reduces the amount of static data; more of the information that distinguishes one object from another is passed through the protocol, and less is stored within the object.

Although the concepts of object sharing and dynamic protocols are not new, we have not previously seen them applied to the area of interactive application interfaces. For example, common toolkits such as those based on the X Toolkit use protocols that require objects to store all the information that they need to define their appearance. Consequently, such objects cannot be shared.

4.2 Managing use-specific data

Some object operations involve significant amounts of processing. For example, operations on composite objects that depend on information obtained from their components may require a complete traversal of the object hierarchy. One way to reduce the effective cost of this computation is to store the result in the object so that subsequent calls can avoid the cost of recomputing it. However, this solution is inappropriate if the information is dependent on the way the object is used, since the use-dependent information would prevent the object from being shared. Our solution to this problem is to store different information for each use of the object, then retrieve the particular information as needed. For example, a composite graphical object that calculates the sizes and shapes of its components could store several sets of information, one for each different use.

Another example of the problem and our solution is provided by graphical objects that draw images. Under the X11 window system (the most common graphical platform for current workstations), drawing large images is expensive because it requires that large amounts of data must be sent to the X server. If the image is drawn frequently (perhaps because it is part of an interactive application), then the application will run slowly. Instead, X allows clients to create server-side objects (pixmaps) for image objects. The client creates a pixmap when the image is first drawn, then stores an object identifier for the pixmap. On subsequent redraws, it can refer to the pixmap by its identifier. The X server can quickly and cheaply redraw the image because the data is already stored in the server.

Consider, now, the effect of object sharing. If a graphical object is used in views that draw on several different X displays, then it must store pixmap identifiers for several X servers. Rather than storing such use-specific data in the object (which would severely restrict its sharability), our solution is to store the needed information with the objects that characterise the use; image glyphs, for example, store pixmap identifiers with the Canvas object that represents the drawing surface. When a glyph is asked to draw itself on a particular Canvas, it first checks to see if an appropriate pixmap has already been created for that Canvas. Since the same pixmap can be used for each use of the glyph on a Canvas, only one pixmap identifier need be stored for the image for each Canvas. The effect is to retain both the sharability of the glyph and the efficiency achieved by storing per-use information. The only cost is the small overhead needed to locate the appropriate information.

In some respects, our approach to building shared views is similar to that taken by systems that allow a single X client window to appear on multiple X displays. These systems use a special version of the X server that acts as an X resource multiplexor. For each X resource allocated by the client, the multiplexor allocates one resource for each controlled display. Client requests to use a particular X resource cause the multiplexor to relay the request to each display in turn, substituting the appropriate resource identifier for that display.

Both models achieve similar results; they allow a single client object to be transparently shared across several displays. The protocol multiplexor model is simpler for clients since the client does not even have to know that its windows are shared. However our model is more powerful since it allows the per-use information to include more than just X resource information.

4.3 Updating dependents

When objects store information that depends on other objects, the application must provide a way to update the stored information following changes to the object on which it depends. For example, if a graphical object calculates its size from the sizes of its components, it must recalculate its size whenever the size of one of its components changes.

Objects that are arranged in simple hierarchies can automatically propagate such changes by defining a protocol for change notification and storing a reference to their parent in the hierarchy. When the object changes, it calls its parent's change method. However, objects that are shared cannot use this approach since storing parent information would make the object unsharable. Instead, the application must explicitly propagate the change to all places where the object is used.

Our solution to this problem is based on the observation that many object hierarchies contain key objects beyond which certain changes are known not to propagate. To continue the graphical object size example, most applications contain objects whose size does not depend on their component's sizes. For example, the size of a graphical editor's drawing window does not depend on the sizes of the components in the drawing. Therefore, a change in the size of a drawing component need not be propagated beyond the window object.

To support this common situation, the application can define special objects that act as stable anchor points in the object hierarchy. If necessary, these objects can store additional per-use information that allows them to propagate changes automatically. Since the number of these stable points is likely to be low in relation to the total number of objects, the overhead of this extra information is small.

5 An example: views of text

This section reports on the use of shared objects to build a document editor. Although the discussion focuses on textual data, many of the results will also apply to other applications. Indeed, a document editor represents a microcosm of common workstation applications: the editor includes simple graphics and tabular capabilities that show how objects can be used to build graphics-based and table-based applications typical of drawing editors and spreadsheets.

5.1 Glyphs: sharable graphical objects

The InterViews user interface toolkit defines simple sharable objects, called glyphs, that produce graphical output on a display (Linton & Calder n.d.). InterViews application programmers can build views of application data by choosing primitive glyphs that represent the components in the data, and composite glyphs that arrange their elements to suggest the structure of the data. As we showed in section 2, a document editor could represent the characters in the document by glyphs that display the appearance of a single character. It would arrange the character glyphs into lines and columns using composite glyphs that tile the characters left-to-right and top-to-bottom.

Glyphs are sharable because the glyph protocol relies on dynamic information; the information that a glyph needs to define its behaviour is passed to the object as part of its method calls. For example, the glyph draw method is passed a geometric allocation, which specifies its size and position, and a canvas, which specifies the surface on which it should draw. The same glyph can simultaneously appear at many different places, and on several different canvases, because the information that distinguishes one use from another is not stored in the glyph. Figure 4 shows the differences between the dynamic glyph draw protocol and a hypothetical static draw protocol. The

```
/* static protocol */

class Glyph {
    ...
    draw_with (Style*);
    draw_on (Canvas*);
    draw_at (Allocation&);
    draw ();
    ....
    Style* style;
    Canvas* canvas;
    Allocation allocation;
};

/* dynamic protocol */

class Glyph {
    ...
    draw (
      Style*,
      Canvas*,
      Allocation&
    );
    ...
};
```

Figure 4: Hypothetical static and dynamic Glyph draw protocols

static protocol would require that each glyph store the allocation, canvas, and style that it needs to draw itself.

Because glyphs are sharable, an application need only create a single glyph instance for each symbol in the document; it can use that instance to represent the symbol each time it appears. For example, a document editor might create a character glyph that represents the letter 'e', then use that instance for each 'e' in the text. This sharing substantially reduces the number of character instances needed for many documents. For example, when editing a typical document such as a technical paper or a chapter in a book, an editor might only need a few hundred glyphs to represent the tens of thousands of characters in the document.

5.2 The object-oriented implementation of a document editor

To show that an object-oriented implementation based on large numbers of objects is practical, we built a WYSIWYG document editor that creates a glyph for each character in the text (Calder & Linton 1992). The editor's performance is comparable to that of similar editors built with conventional techniques, but its implementation is much simpler. The editor is in regular use with documents ranging in size from one page memos to a PhD dissertation of over 150 pages. We used the editor to create and publish this paper.

Like many similar applications, the editor uses a data-view model: the application data is structured as a hierarchy of items; each view is a separate hierarchy of dependent item views. An item can have several views; when it changes, it updates all its views.

The editor defines an abstract base class, Item, that specifies the interface to the data classes, and an abstract base class, ItemView, that specifies the interface to the view classes. Each type of document item is implemented as a subclass of Item; each Item subclass has a corresponding ItemView subclass. Table 1 lists the classes defined in the current implementation.

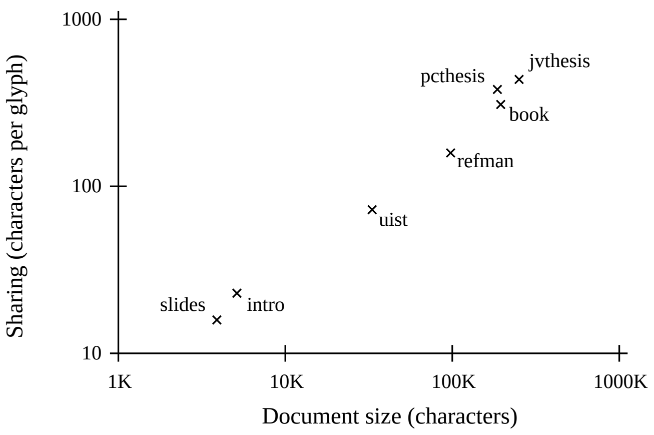

Figure 5: Glyph sharing versus document size

To find out how effective glyph sharing was in reducing the number of glyphs, we instrumented the editor code to record the number of glyph instances created when editing a range of documents. Table 2 shows the results and figure 5 plots the average sharing (expressed as the ratio of characters to glyphs) against document size.

Surprisingly, the total number of character glyphs increased with document size throughout the range of documents tested (we had expected the number of glyphs to be roughly constant for all but the smallest documents). Analysis of the data showed that most of this increase was due to glyphs that represented characters in seldom-used document styles. The editor creates one glyph for each character in each document style. The number of glyphs for commonly-used styles such as the document's main body style rapidly approached an upper limit of about 100 (the 95 printing ASCII characters plus a handful of spacing and special punctuation characters). But the number of glyphs needed for specialised portions of the document such as equations or superscripts remained low even for large documents.

The slow spread of glyphs into seldom-used styles, coupled with the natural range in character frequency for English text, resulted in large differences in the degree of sharing for individual glyphs. The most commonly used glyph for all documents was the interword space glyph in the document's main body style. In the larger documents, for example, this one glyph was used over 20,000 times, while about 50 glyphs were each used only once. Overall, the 20 most widely used glyphs accounted for about 80% of the characters in each of the documents.

5.3 Other applications

Although these results relate specifically to textual data, we believe that similar behaviour would be observed for other common interactive applications. (Vlissides and Linton have described an object-oriented drawing editor implementation (Vlissides & Linton 1988), but they do not explore the possible role of sharing.)

A common factor for many interactive applications is that user interacts with data that exhibits replication at the level of the fine-grained structure. For example, a schematic drawing of an electrical circuit will contain many instances of standard symbols such as transistors and resistors, while a music score will contain many instances of notes and other symbols. Interactive applications that edit such data can use sharing to reduce their object counts. Of course, most applications contain significant amounts of text in ad-

Item	View	Function
TextItem	TextView	text flow
TabularItem	TabularView	simple table
PSFigItem	PSFigView	graphic
FloatItem	FloatView	document float
LabelItem	LabelView	cross-reference target
RefItem	RefView	cross-reference
CounterItem	CounterView	counter increment
PagenumberItem	PagenumberView	page number

Table 1: Document item subclasses and their views

Document	Description	Pages	Words	Characters	Glyphs	Average uses
intro	paper abstract	1	750	5148	220	23
slides	conference talk slides	15	400	3900	249	16
uist	conference paper	10	5000	33218	454	73
pcthesis	complete dissertation	130	27000	185833	486	382
jvthesis	complete dissertation	160	39000	254204	579	439
refman	InterViews reference manual	45	15000	98668	610	162
book	InterViews book	120	32000	199432	639	312

Table 2: Sharing of Character glyphs

dition to other data; the benefits seen here can be applied directly to this component of the application.

6 Conclusions and further work

If objects are used to represent the fine-grained structure in applications that present interactive views of richly-structured data, the implementation will be straightforward. However, unless the objects are carefully designed, the resulting application can be costly in its use of computer resources.

Sharable objects can dramatically reduce the cost of using objects at this low level by reducing the number of object instances needed and by eliminating the need to build parallel object hierarchies. For a document editor application, sharing reduced the number of object instances for large documents by a factor of several hundred.

The key to making objects sharable is a dynamic object protocol. Dynamic protocols pass needed information to objects rather that requiring them to store the information internally; thus a single instance object can be used with different data.

Currently, our document editor uses a classical data-view model, with a hierarchy of data objects and matching hierarchies of view objects for each view. We are currently developing new components that will allow us to build a version that use a single view object hierarchy for several views. Later we will further refine the objects to eliminate all structural redundancy by using a single object hierarchy for data and views. We expect that these changes will further reduce the editor's resource usage at the same time as simplifying its construction.

So far, we have only experimented with single-user applications. However, we have begun to explore the use of shared objects to simplify applications with multi-user interfaces. For example, a multi-user document editor would allow several users, perhaps geographically separated, to collaboratively edit the same document. Such applications are difficult to implement and are not well supported by current tools and techniques. We believe that object sharing may simplify the construction of such applications by allowing the programmer to represent explicitly the shared objects in the users' views.

References

Calder, P. R. (1993), Building User Interfaces with Lightweight Objects, PhD thesis, Stanford University.

Calder, P. R. & Linton, M. A. (1990), Glyphs: Flyweight objects for user interfaces, *in* 'Proceedings of the ACM SIGGRAPH Third Annual Symposium on User Interface Software and Technology', Snowbird, UT, pp. 92–101.

Calder, P. R. & Linton, M. A. (1992), The object-oriented implementation of a document editor, *in* 'ACM OOPSLA '92 Conference Proceedings', Vancouver, British Columbia, pp. 154–165.

Krasner, G. E. & Pope, S. T. (1988), 'A cookbook for using the model-view-controller user interface paradigm in smalltalk-80', *Journal of Object-Oriented Programming* **1**(3), 26–49.

Linton, M. A. & Calder, P. R. (n.d.), Flyweight objects in InterViews 3.0, *in* 'Fifth Annual X Technical Conference, Technical Papers'.

Linton, M. A., Calder, P. R., Interrante, J. A., Tang, S. & Vlissides, J. M. (1991), *InterViews Reference Manual, Version 3.0*, Stanford University.

Linton, M. A., Vlissides, J. M. & Calder, P. R. (1989), 'Composing user interfaces with InterViews', *Computer* **22**(2), 8–22.

Scheifler, R. W. & Gettys, J. (1986), 'The X window system', *ACM Transactions on Graphics* **5**(2), 79–109.

Swick, R. R. & Weissman, T. (1988), *X Toolkit Widgets—C Language Interface*, Digital Equipment Corporation. Part of the documentation provided with the X Window System.

Vlissides, J. M. & Linton, M. A. (1988), Applying object-oriented design to structured graphics, *in* 'Proceedings of the 1988 USENIX C++ Conference', Denver, CO, pp. 81–94.

Weinand, A., Gamma, E. & Marty, R. (1988), ET++—An object-oriented application framework in C++, *in* 'ACM OOPSLA '88 Conference Proceedings', San Diego, CA, pp. 46–57.