

## The Andrew Toolkit - An Overview

*Andrew J. Palay  
Wilfred J. Hansen  
Michael L. Kazar  
Mark Sherman  
Maria G. Wadlow  
Thomas P. Neuendorffer  
Zalman Stern  
Miles Bader  
Thom Peters*

Information Technology Center  
Carnegie Mellon University  
Pittsburgh, PA 15213

### ABSTRACT

The Andrew Toolkit is an object-oriented system designed to provide a foundation on which a large number of diverse user-interface applications can be developed. With the toolkit the programmer can piece together components such as text, buttons, and scroll bars, to form more complex components. It also allows for the embedding of components inside of other components, such as a table inside of text or a drawing inside of a table. Some of the components included in the toolkit are multi-font text, tables, spreadsheets, drawings, equations, rasters, and simple animations. Using these components we have built a multi-media editor, mail system, and help system. The toolkit is written in C, using a simple preprocessor to provide an object-oriented environment. That environment also provides for the dynamic loading/linking of code. The dynamic loading facility provides a powerful extension mechanism and allows the set of components used by an application to be virtually unlimited. The Andrew Toolkit has been designed to be window system independent. It currently runs on two window systems, including X.11, and can be ported easily to others.

### 1. Introduction

During the past five years a number of window systems have been developed for high-resolution bit-mapped graphics displays (Macintosh[1], SunWindows [2], Andrew system[3,4], X Windows[5], NeWS[6]). Each of those systems have included a programmer interface for developing applications. These have been low-level interfaces that have provided a simple graphics abstraction, a method for receiving input events and perhaps some simple components of the system (menus, scroll bars, dialog boxes). In writing to the lower-level interface, application programmers continually replicate the same functional body of code. Further, since the lower-level interface provides few guidelines for the developer, it becomes difficult to build components that can be used by other developers. This again results in replication of large amounts of code that should be reused. It also results in the development of inconsistent applications. Since each window system does provide a full user interface system, application programmers have built functionally equivalent but divergent user interfaces. This leads to chaos for a user community. Because of these problems, higher-level interfaces have been developed (MacApp[7], SunView[8] the Andrew Base Editor[9], X Toolkit[10]).

The Andrew Toolkit (formerly known as Base Environment 2 or BE2) is a new high-level environment for the development of user interface applications. Built upon the lessons learned over the

past four years during the development of a prototype system built at the Information Technology Center (ITC), the toolkit provides a general framework for building and combining components. It is based on a minimal protocol that allows components to communicate with each other about user interface policies, while allowing the developer maximum freedom to determine the actual interactions between components.

The Andrew Toolkit has been built using an object-oriented system, the Andrew Class System, that also provides the ability to dynamically load and link code. This ability provides a powerful extension facility for applications. We have already used this feature to build a generic multi-media editor (EZ) that can edit a wide variety of components by loading the appropriate code when needed. Further, the dynamic loading facility can be used to add additional components to the basic toolkit without having to rebuild the applications.

The toolkit provides the usual set of simple components (menu, scroll bars, etc) and a number of higher-level editable components including multi-font text, tables/spreadsheets, drawings, equations, rasters and simple animations. The text and table components are multi-media components, in that they allow the embedding other components within their description. The drawing component will soon support this feature.

In addition to the editor, we have developed a number of basic applications including a mail system[11], a help system, a typescript facility that provides an enhanced interface to the C-shell, a *ditroff* previewer, and a system monitor (console) that displays status information such as the time, date, CPU load and file system information. Since both the mail and help applications use the text component for the display of information, they automatically inherit the multi-media functionality of the text component. Examples snapshots of these applications can be found at the end of this article.

We have also developed a number of extension packages. These include a C-language programming component, a compile package, a tags package, a spelling checker, a style editor and a filter mechanism<sup>1</sup>.

The original design of the Andrew Toolkit arose from discussions about the development of a text editor that would allow the user to embed other components, such as tables, drawings, rasters, etc. Further we wanted those components to be editable in place. Some of the problems that must be addressed in building such a system include:

- how to resolve the handling of input events between components.
- how to arbitrate the display of menus.
- how to arbitrate the display of the mouse cursor.
- how and when to display components.
- how to determine the size and placement of embedded components.
- how to store the external representation of components.

In examining these problems, we developed a general architecture that allows the inclusion of one component inside another. This allows the developer to build components that can embed arbitrary components without detailed knowledge about the embedded object. Further, new components can be easily included in already existing components without any additional work.

For example, users of the Andrew System can currently compose papers that contain tables, equations, drawings, rasters and animations. The text component uses a generic mechanism to include other components. If a new component is developed, it can be included in the text component using that same mechanism. This is an important feature of the system, especially within a university environment. Given our limited resources we knew from the outset that we could not write all the components that were required by the university. For example, members of the music department will want to include musical scores inside of text just as easily as others include tables. Members of the electrical engineering department will want to include circuit diagrams inside of text. The list is essentially

<sup>1</sup> the filter mechanism gives the user the ability to use standard tools on regions of text contained in a file being edited.

limitless.

The dynamic loading feature is essential in extending the toolkit's functionality. If a member of the music department creates a music component and embeds that component into a text component (or any other component that allows embedding of components), the code for the music component will be dynamically loaded into the application. Except for a slight delay to load the code, the user of the editor is unaware that the music component was not statically loaded. The user is also unaware that the music component was not part of the original system. The editor did not have to be recompiled, relinked, or otherwise modified to use the new music component. Further, all users of the text component automatically acquire the ability to use the music component: it can be sent in a mail messages easily as edited in a document.

The Andrew Toolkit has been designed to be window system independent (and to a great extent operating system independent). It currently runs under both the original ITC/Andrew Window System and under X.11. Within the university community, and we believe the more general community as well, there exists a wide range of machines and thus window systems that need to be supported. Within the UNIX<sup>†</sup> operating system community, the X Window System is developing into a de facto standard, although other window systems may eventually rival it. In the lower-end personal computer market there exist the PC and the Macintosh. Even though the price of computers will continue to drop those machines will remain useful environments for many users. Finally there is the development of newer operating systems, such as OS/2, for machines between the low-end personal computers and the high-function UNIX workstations.

Since we could not see the development of a window system standard over that range of machines, we chose to make Andrew Toolkit window system independent. This will allow us potential to support a consistent set of applications over a diverse set of window systems. Clearly the personal computer versions would be more limited, but we consider it to be a great advantage for a user to be able to use a low-cost machine for the normal simple tasks, and then easily move to the more powerful machines when required.

## 2. Basic Toolkit Objects: Data Objects and Views

The Andrew Toolkit is based on the development of components that can be used as building blocks for either applications or other more complex components. *Data objects* and *views* are two closely related basic object types within the toolkit. A toolkit component is normally composed of a view/data object pair. The data object contains the information that is to be displayed, while the view contains the information about how the data is to be displayed and how the user is to manipulate the data object (the user interface). For example, the text data object contains the actual characters, style information and pointers to embedded data objects. It also provides ways to alter the data, such as inserting characters and deleting characters. The text view contains information such as the current selected piece of text, the portion of the text that is currently visible, and the location of the text. The text view provides methods for drawing the text, handling various input events (mouse, keyboard, menus), and manipulating the visual representation of the text.

The contents of a data object can be saved in a file, but the contents of the view cannot. The information associated with the view is transient and is valid only during the running of an application. When the application terminates that information has no further meaning. On the other hand, views provide the facility for printing within the Andrew Toolkit.

While it is often the case that a view has an underlying data object, there are many cases when a view will be used to solely provide a user interface function. In such a case there is no underlying data object. The scroll bar is one such example. It only adjusts the information contained in another view.

We have made the view/data object disjunction to provide a system where multiple views can simultaneously display the information contained in a single data object. Our design is similar to the Model-View-Controller design used in Smalltalk[12] systems. By comparison, our data objects serve as the models, our views are views, and the controller is distributed between the interaction manager

<sup>†</sup> UNIX is a trademark of Bell Laboratories.

(global decisions) and individual views (decisions between children and parent views).

This separation of concerns has brought us many advantages. For example, in building an editor, we wanted to provide the user with the ability to edit the same information in more than one window. Further, we wanted changes made in one window to be reflected in the other. This case is handled by having two views of the same type, one in each window, displaying information from the same data object. Similarly, we might want to have multiple views of the same type on a single data object in one window. A system like Aldus' PageMaker(TM) could be built under the Andrew Toolkit by allowing the user to specify a set of views and their placement on a page. Some of those views (for example, the text views) would be examining different sections of the same data object.

It is also possible to have two different types of views displaying information contained in the one data object. Currently the text view is a display-based text processing system. It can be characterized as a semi-WYSIWYG<sup>2</sup> or a WYSLRN<sup>3</sup> view. It displays text with multiple fonts, indentations, etc. but makes no attempt to display the information as it would appear on a piece of paper. This view has been used for the basic text editor as well as the mail and help systems. It has been successful, except in the case the user wants to format the text for printing. In this case we plan on providing a full WYSIWYG text view. This paper-based text view will be designed to use the same text data object. The user of the system will be able to choose to use either view or perhaps have one window using the normal text view and the other using the WYSIWYG text view. Again changes made in one window will automatically be reflected in the other window.

Just as it is possible to have two different views on the same data object in two windows, it is also possible to have two different views on the same data object within the same window. A text component could have two embedded views on the same data object. For example, the user might want to display a table of numbers and a pie chart representing the table. This could be done by having one table data object and two views, a normal table view and a pie chart view.

Despite its advantages, there is a cost to separating information into data objects and views. Two particular areas of difficulties we have discovered are coordinating data objects and views, and the maintaining stable view state. Each is briefly discussed below.

Our system does not encourage a close connection between the changing of the information contained in a data object and the update of the visual appearance provided by the view. Since only one view will be causing the data object to change, and multiple views may have to reflect the change, a delayed update mechanism must be used. When the user issues a command to a view to alter the underlying data object, the view firsts request that the data object modify itself and then requests the data object to inform all of its views that it has changed. When a view is informed that the underlying data object has changed, it must determine what the change is and update its visual representation appropriately.

The delayed update mechanism is the trickiest challenge in building a data object/view pair. The developer must develop some mechanism with which the view can determine which portion of the data object has changed. This mechanism is normally provided by a set of methods exported by the data object. It is not considered to be proper behavior for the data object to have detailed knowledge of a specific type of view. This would be one way to handle the delayed update, but would preclude the development of other types of views on the same type of data object.

The second difficulty we faced was retaining the stable (or permanent) state for a view. In the chart example, the underlying data object is a table of values. When a file displaying the chart is saved, only those values (along with the information that a "chart" is viewing the table) is saved. However, the user may have set certain parameters in the chart, such as the way to label the axes. This information is not part of the table data object and would not stored in it. Since a view has no permanent state, information kept in the view, such as a axes labelling, would also not be saved. In its simplest form, there is no place to keep this view specific information.

<sup>2</sup> What You See Is What You Get

<sup>3</sup> What You See Looks Real Neat

Our solution consists of two parts: additional data objects and the idea of an observer. In the example above, the chart view would be viewing not a table data object but an auxiliary chart data object. The chart data object would retain information such as axes labelling. In addition, the chart data object would be an observer of the table data object. As information in the table changed, the chart data object would be notified and it, in turn, would notify the chart view. In fact, we do not have specially defined auxiliary data objects. Rather, our update system is based on the observer mechanism, where a data object may be observed by any number of other data objects and views. We have found that this design has permitted a great deal of flexibility and functionality for combining pieces in the toolkit.

### 3. Event Processing: The View Tree

Views are used to define the user interface for an application. In defining a user interface, the view must handle both the visual display of information on the screen and the handling of input events that might change that display. Views are organized in a tree structure. Visually, each view is a rectangle and is completely contained in its parent view. At the top of the tree is a view called the *interaction manager* which is a window provided by the underlying window system. The interaction manager has the responsibility of translating input events such as key strokes, mouse events, menu events and exposure events from the window system to the rest of the view tree. The interaction manager is also responsible for synchronizing drawing requests between views. By design, it has one child view, of arbitrary type. That view may have any number of children. Child views are always visually contained within the screen space allocated to their parent, but the toolkit does not define any screen relationship between sibling views.

In general, when an event is received by the interaction manager, the event is passed down to the interaction manager's child. That view determines if it is interested in the event or if it should pass it down to one of its children. This process recurs until some view actually handles the event. By passing the event down the view tree, each parent gets the chance to determine the disposition of the event. The view can use the semantic information associated with itself to make that determination.

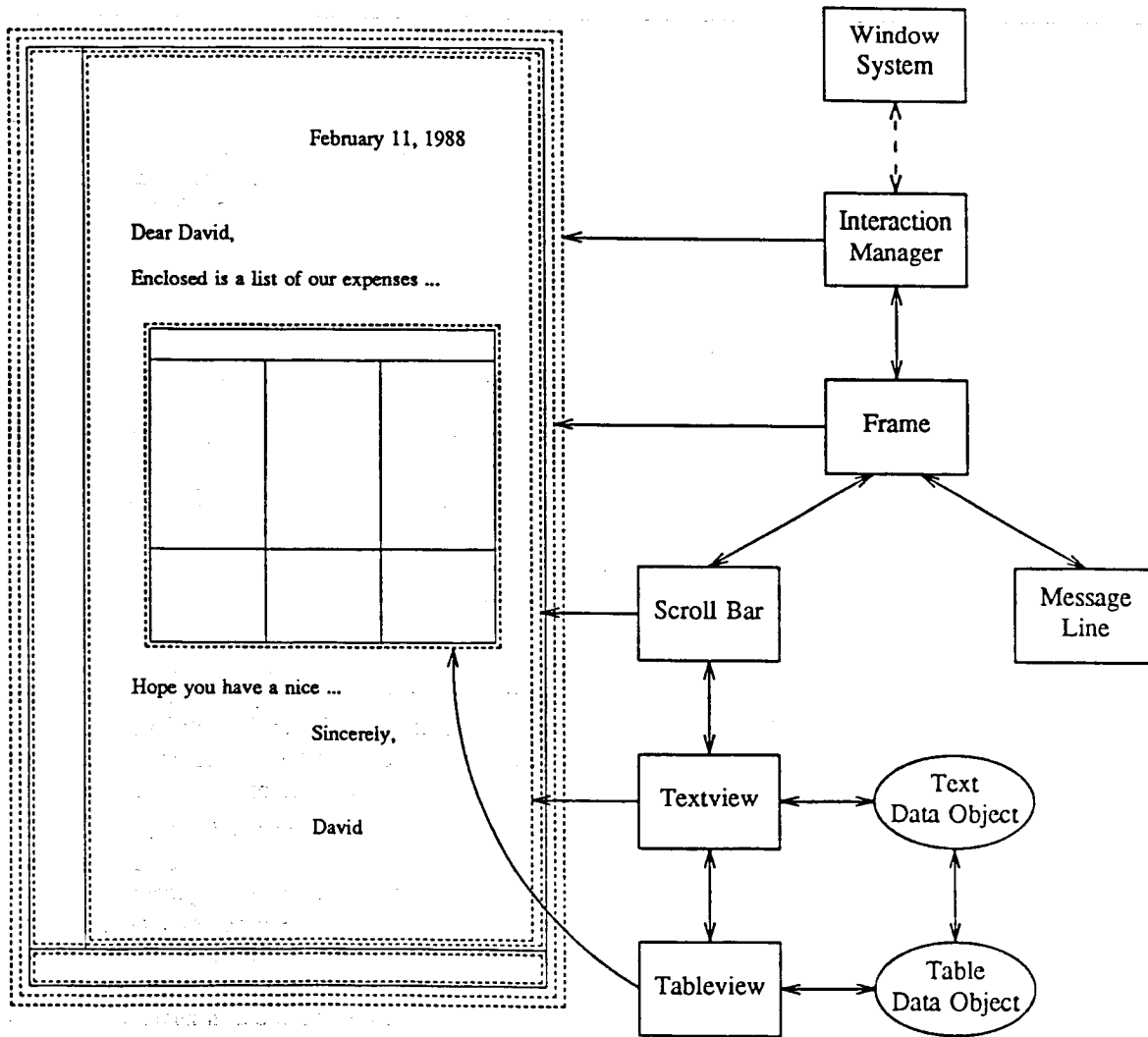
Updates to the visual image of an application are handled in a similar fashion. When a view wants to update its image it makes a request to its parent view. That request is usually passed up to the interaction manager which then sends an update event back down the tree. Since a view might be embedded in another view, it does not have complete control over its allocated screen space. The parent might have overlaid some other image on top of the child's image. By posting an update request up the tree and having the update event come back down, the parent can now update its image and the children's images in the appropriate order.

The following figure presents a view tree for a window that contains a scrollable text view that contains a table view. The text view is surrounded by a scroll bar, which is surrounded by a frame. The frame provides a message line view.<sup>4</sup> The text and table views reference their respective data objects. The lines around the screen image represent the physical area of the image associated with each view.

When a mouse event is received by the interaction manager, it passes the event down to the frame view. The frame determines if it should handle the mouse event directly or if it should be passed down to either of its children. The frame accepts the mouse event directly if it is close to the dividing line between its two children (in this case the user is allowed to adjust the position of the dividing line). If the mouse event is passed down to the scroll bar view, that view will accept the mouse event if it is over the scroll bar or pass it to its child if it is not. The text view accepts the mouse event if it is not in any of its subviews; otherwise it too passes the event down.

As the mouse event works its way down the view tree, the view determining the disposition of the mouse event only needs to be aware of the location of its children and not the child's type. Similarly, the child needs to have no knowledge about the type of its parent, nor its location in the overall view tree.

<sup>4</sup> The frame, in conjunction with the message line also provides a dialog box facility. To simplify the figure, that detail has been omitted.



The controlling relationship between a view and its children is one area in which the Andrew Toolkit differs from other toolkits. Other systems closely tie the handling of events to the physical relationship of components on the screen. If a component is physically on top of another component it will block the transmission of certain events to the lower component<sup>5</sup>. While this is valid in many circumstances, there are times when it is not. Further, many toolkits use a global analysis of all views in order to process and distribute events. The Andrew Toolkit distributes this authority to each view over its children.

Our toolkit was designed to overcome the limitations that we had experienced with a global, physical model. An early prototype toolkit (the Andrew Base Editor) tied the handling of events to the physical relationship of components on the screen. During the early design phase for the Andrew Toolkit, we attempted to build a drawing editor using that prototype. The drawing editor used the text component to display and edit text within the drawings. The text component was a subordinate of the drawing component. The user of the drawing editor might first enter some text and then place a line over the text. When a mouse event occurs near that line only the drawing component could determine whether the user was selecting the line or the underlying text. This was impossible to accomplish since the toolkit maintained strict, global control over the distribution of input events.

<sup>5</sup> X.11 comes very close to handling this correctly except for exposure events which do not propagate to overlapped windows.

A similar case can be seen in the handling of mouse events by the frame view. The frame physically divides its image into two areas separated by a thin line. In order to allow the user to easily drag that line, the frame allocates a slightly larger area to accept mouse events. That area overlaps the space allocated to the frame's children. If the handling of events was dictated by the screen layout, this interaction would be much more difficult to provide and would require more detailed knowledge of the view tree structure to maintain.

The parental authority is a major architectural concept in the Andrew Toolkit. The discussion above described how this authority is exercised for controlling mouse events. The same mechanism is used between children and parents to negotiate the contents of menus, the display of cursors, the mapping of keyboard symbols and the focus of attention.

#### 4. The Graphics Layer

The view tree mechanism provides a general mechanism for handling of events in the Andrew Toolkit. It hides from the developer the specifics of the input model used by the underlying window system. In a similar fashion, the toolkit uses a graphics layer to hide the output model of the specific display medium. The display medium is usually the underlying window system, but can also be a printer.

The graphics layer is built using a third type of object, the *drawable*. A drawable contains information about the underlying graphics medium. For a window system, that information normally includes:

- the window to draw in.
- the location of the drawable in that window.
- a small graphics state (e.g. current point, line thickness, current font).
- the coordinate system for the drawable.

The drawable provides a set of drawing operations similar to those provided by the X.11 window system.

Each view contains a pointer to a drawable, which is used for all drawing operations. The developer of a view rarely accesses a drawable directly. All methods exported by the drawable are also provided as part of the view interface.

Separating the view and the drawable will allow us to provide a simple default printing mechanism. When a view receives a print request for a specific type of printer it can temporarily shift its pointer to a drawable for that printer type and do a redraw of its image. We expect to provide this facility in a later version of the toolkit.

#### 5. External Representation

Most of the problems with embedding components inside other components are solved by the view interface, except for the external representation of the components. As stated earlier, only data object descriptions are written out to files. The toolkit architecture places one requirement on the external representation. When a data object writes out its external representation it is enclosed in a begin/end marker pair. The markers must be properly nested and it must be possible to find all the data associated with an object without actually parsing the data. Those markers provide a tag denoting the type of the object being written and an identification tag that can be used for referencing the data object by other data objects.

Thus the earlier example containing a table embedded in text would have an external representation that looks like:

```
\begindata{text, 1}
text data ...
\begindata{table, 2}
the table data goes here ...
\enddata{table,2}
more text data ...
\view{spread, 2}
rest of text data ...
\enddata{text,1}
```

The `\view` construct is specific to the text object and indicates the exact placement within the text of the view (of type spread) on the table data object.

The use of nested begin/end markers is the only requirement of for the external representation we also strongly encourage developers to follow the following guidelines:

- use only printable 7-bit ascii characters (including tab, space and new-line).
- keep line lengths below 80 characters.
- make the representation understandable.

The first two suggestions make it possible to transport files across almost all networks (especially as mail). The final suggestion is an attempt to provide an easier recovery mechanism in the rare occurrence when files are partially destroyed. This suggestion only makes sense in the context of the first two suggestions. If the file is being stored using only 7-bit codes and with line lengths than 80 characters then the overhead in making it understandable is small. Of course there are some objects, such as rasters, where this requirement is impossible to meet. However, even in those cases it is possible to make it slightly more comprehensible. For example, the raster format could make sure the bits representing a new row always begin on a new line.

## 6. The Object Oriented Environment

The Andrew Toolkit is built using the Andrew Class System (Class). Class provides an object-oriented environment with single-inheritance. The Class language permits the definition of object methods and class procedures. Object methods are similar to C++[13] methods, and a may be overridden in subclasses. Class procedures are similar to Smalltalk's class methods, only they may not be overridden. C procedures for controlling the initialization and disposition of objects are created by the Class preprocessor. Class also provides for the dynamic loading/linking of code.

Class is a C language-based system. It consists of a small run-time library and a simple preprocessor that only preprocesses class header files. The class header files are almost identical to standard C header files except for the inclusion of another type of definition, the class. The preprocessor, generates two include files, an export file (.ch) that is used when defining a class and an import file (.ih) that is used when using a class. The C files written using Class look almost identical to normal C files and are not run through a special preprocessor.

Class is similar in nature to C++. We chose to implement our own system for several reasons:

- We wanted to support dynamic loading/linking. C++ generates code that must be statically linked. Modifying the C++ preprocessor was considered but deemed impractical without getting the changes made in official version. We made some initial inquiries, but could not solve this problem quickly enough for development to continue.
- We wanted to develop a system that could be debugged easily. C++ preprocesses both include and source files. Until a debugger is built that understands the original C++ code, developers would have to understand the transtormations made by the C++ preprocessor. This would pose only small problems for the highly experienced developer, but would cause problems for our developer community.
- We wanted as simple a system as possible. We needed an object-oriented environment but not the other constructs built into C++.



- We wanted to be free of any external dependencies that required yet another licensing agreement. We hoped to make the toolkit widely available. Requiring another licensed product seemed to be a bad idea.

Even though we did not use C++ to implement the toolkit, the Class system used C++ as a model for its object oriented system. If the above problems were solved (which might now be a possibility) converting to C++ would be a relatively easy (but time-consuming) activity.

## 7. Extending the Toolkit

The Andrew Toolkit has been built to be extendable. This is a major feature of the system. The system has been designed in such a fashion that the creator of a data object or view does not have to take any special action to allow that object to be embedded in another object. The data object and view interfaces have been designed to provide the necessary and sufficient set of methods for two objects to communicate without detailed knowledge of each other. Further, those interfaces make it easy for the author of an object to allow it to include other objects. The text object can include any other type of data object. Authors of new objects are strongly encouraged to handle the inclusion of arbitrary objects instead of special casing the inclusion of specific objects.

The dynamic loading/linking feature also provides a low-level extension language for applications built using the toolkit. Sophisticated users can write code (using the class system) to implement new commands. These commands can be bound either to key sequences or to menus. When invoked, the code is loaded and executed.

This feature has also provided us the ability to run all of our applications from a single base program. We have created a program, called *runapp*, that contains the basic components of the toolkit. The code for each individual application is then dynamically loaded in at run time. Since most UNIX systems do not provide shared libraries, this allows multiple toolkit applications to share a significant portion of code. This leads to performance improvements in a large number of areas:

- paging activity is reduced.
- key portions of the code are almost always paged in thus improving user performance.
- virtual memory use decreases
- file fetch time decreases if running under a distributed file system.
- the file size of an application is reduced.

## 8. Window System Independence

The Andrew Toolkit has been designed to be window system independent. To port the toolkit to another window system, six classes must be written, encompassing approximately 70 routines. Of those routines, about 50 routines are normally simple transformations to the graphics layer of the underlying window system. Once those are written, any toolkit application should run in the new environment<sup>6</sup>. The six classes that must be written are:

- *Window System*: this class exists to allow the toolkit to get a handle on the other window system classes listed below.
- *Interaction Manager*: this class provides the interface to the event processing mechanism from the underlying window system. This includes the handling of keystrokes, mouse events and menus.
- *Cursor*: this class provides an interface to defining cursors on the underlying window system.
- *Graphic*: this class provides the output interface to the underlying window system. All drawing operations are made using this class.

<sup>6</sup> Some applications such as *typescript* are dependent on the underlying operating system, and will not port quite as easily.

- *FontDesc*: this provides an interface to font descriptions.
- *Off Screer. Window*: this provides the facility to draw off screen images that can be later included on screen.

Using this facility we are currently able to run applications on two different window systems without any recompilation. Applications are normally configured for one system. However, using the dynamic loading facility, the modules for the other system can be loaded at run time. The choice of window system to use is currently controlled by the setting of an environment variable. With a little more restructuring of the basic code we believe that it will be possible to actually open windows on two different window systems at the same time.

## 9. Current Status

The basic toolkit applications (editor, mail, help, preview, typescript, console) have been in general use on the Carnegie Mellon campus for the past four months using the original Andrew window system. The system is actively used by approximately 3000 people. Users are beginning to experiment with the multi-media facility which has only been recently advertised. Within the ITC we are starting to convert to X.11. We hope to be using X.11 within the ITC exclusively by the middle of winter. The timetable for converting the campus to X.11 is currently the summer of 1988. This depends on the conversion of various applications to use the toolkit and the performance of available X.11 systems.

One of the challenges associated with building user-interface software is to make it easy to use for the beginning users while making it powerful enough for experienced users. The prototype editor built at the ITC was highly influenced by the goal of making it easy for the novice user. While we were developing that system and until the release to the ITC of EZ, programmers at the ITC used *emacs* to edit programs. Since the release of EZ, use of *emacs* has dramatically decreased. This has been accomplished without sacrificing the usability of the system by our campus user community.

## 10. Conclusion

UNIX, and its software tools approach to computing, provided a new paradigm for building applications. The idea was that portable, general purpose modules could be strung together in different ways to create complex applications without a lot of duplication of effort. The Andrew Toolkit can be seen as an extension of this concept to the graphic workstation environment.

In this way, the Andrew Toolkit is unique among existing toolkits. It provides the usual toolkit functions (text, scroll bars, dialog boxes, etc) but also provides the ability to embed components inside of other components. The architecture for embedding components has been designed to strongly encourage programmers to build new components that can be used in both new and existing applications. The architecture also encourages programmers to develop objects that can include arbitrary components instead of specific ones.

The separation of information into data objects and views provides a highly modularized structure that also supports the building block paradigm. In this way data objects can be used in ways different than originally envisioned by their creator. New views on existing data objects can be created. Existing data objects can also be used as the building blocks for more complex objects. This can be done without using the existing object's companion view.

The Andrew Class System is an essential element in supporting this paradigm. The object oriented nature of the system allows programmers to easily develop new specialize objects out of existing objects such as the C language component. The dynamic loading facility of Class allows the toolkit to be easily extended by a large community of developers.

Finally, the toolkit is unique among other toolkits in its potential to provide a common base of applications across a diverse set of machines and window systems. We have spent considerable effort to make the system window system independent and believe that it will be important in the future to support the same software base on many systems.

existing

## Acknowledgments

The Andrew Toolkit and applications have been designed and developed over the past two and a half years. Many other people have been involved in various phases of the work including Andrew Appel, Nathaniel Borenstein, Mark Chance, Richard Cohn, Curt Galloway, John Howard, Tom Lord, William Lott, Bruce Lucas, David Nichols, Marc Pawliger, and Adam Stoller. The toolkit could not have been built without the earlier work done at the ITC by James Gosling and David Rosenthal. Documentation for the toolkit has been written by Chris Neuwirth and Ayami Orgura.

We are also grateful to our user community, especially our fellow workers at the ITC, who have been forced to be the front-line testing organization for our software over the past four years.

The ITC is a joint project between IBM and CMU. The ITC also receives support from the National Science Foundation. Development and deployment of the toolkit would not have been possible without the support of these organizations.

## References

1. *Inside Macintosh*, Addison-Wesley, 1985.
2. *SunWindows System Programmer's Guide*, Sun Microsystems, Inc..
3. James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, and F. Doleson Smith, "Andrew: A Distributed Personal Computing Environment," *Communications of the ACM*, vol. 29, no. 3, pp. 184-201, March 1986.
4. James Gosling and David S. H. Rosenthal, "A Network Window Manager," in *Proceedings of the 1984 Uniform Conference*, Washington, D.C., January 1984.
5. R.W. Scheifler and J. Gettys, "The X Window System," *ACM Transactions on Graphics*, vol. 5, no. 2, pp. 79-109, April 1986.
6. *NeWS Manual*, Sun Microsystems, Inc., March 1987.
7. K. Shumucker, *Object-Oriented Programming for the Macintosh*, Hayden Book Company, Hasbrouck Heights, NJ, 1986.
8. *SunView System Programmer's Guide*, Sun Microsystems, Inc..
9. James Gosling and David S. H. Rosenthal, "The User Interface Toolkit," in *Proceedings of PRO-TEXT 1 Conference*, 1984.
10. *X Toolkit Library - C Language Interface*, Massachusetts Institute of Technology and Digital Equipment Corporation, 1987.
11. Nathaniel Borenstein, Craig Everhart, Jonathan Rosenberg, and Adam Stoller, "A Multi-media Message System for Andrew," in *Proceedings of the USENIX Technical Conference*, Dallas, TX, February 1988 (this volume).
12. Adele Goldberg and David Robson, *SmallTalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.
13. Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

Snapshots

1. Snapshot of a full screen image containing a *console*, a *typescript*, and two *ez* windows. The two *ez* windows show editing a document and a c file.

The screenshot shows a desktop environment with several windows:

- console**: Shows system messages and a snapshot stored in file `kmp/snapshot-5` at 2:53:16 PM.
- typescript**: Contains a C program for handling messages, including functions for reading, printing, and queuing messages.
- ez -/docs/usenix.d**: Displays "The Andrew Toolkit - An Overview" with a list of authors (Andrew J. Paaly, Wilfred Hansen, Michael L. Kazar, Mark Sherman, Maria G. Wadlow, Thomas P. Neundorffer, Zalman Stern, Miles Bader, Thom Peters) and contact information for the Information Technology Center at Carnegie Mellon University.
- ez -/tc2/basics/init.c**: Shows C code for initializing and finalizing an object, including error handling and key state management.

2. Snapshot of a *help* window. Users can see overview information by clicking in the *Overviews* panel on the right. Similarly they can see specific information about a program by clicking in the *Programs* panel.

The screenshot shows a help window titled "EZ: A Document Editor" with the following content:

**What EZ is**

EZ is an editing program that you can use to create, edit, and format many different types of documents. This help document introduces EZ and explains how you can use it to create and edit "text" documents. It is composed of these parts:

- 1) Related Information about EZ
- 2) Starting EZ
- 3) Selecting Text and using menus
- 4) Previewing and Printing your documents
- 5) Quitting EZ
- 6) Advice
- 7) Pop-up Menu meanings
- 8) Quick Reference
- 9) Related tools

For details about creating and editing other types of documents with EZ, see the *insets* overview file. It is

**Right Panel:**

- Overviews**
  - Andrew Tour
  - Bulletin Boards
  - Customizing Andrew Mail
  - Managing Files and Directories
  - Printing Documents
  - Programming
  - Protecting Files and Directories
  - Using Other
- Programs**
  - About help
  - cat
  - cd
  - console
  - cp
  - ez
  - ezprint
  - fs
  - help
  - login
  - logout

3. Snapshot of *messages* reading window. The panel on the left gives a list of message folders that can be read. It currently contains a list of all the messages folders available on campus. It can also be set to display the folders a user is subscribed to or just the user's personal folders. The panel at the top left contains the list of messages in the selected folder. The message being displayed contains a drawing within the text of the message.

messages		Version 5.21-N	harmerville
All 1414 Folders		andrew.ms.demo (Local Bboard, 9 of 19 new)	
<input checked="" type="checkbox"/>	andrew.ms.demo.horrible.po2	✓ 23-Oct-87 <i>What it does, how it works</i> - Nathaniel Borenstein (275)	
<input checked="" type="checkbox"/>	andrew.ms.demo.horrible.po3	✓ 23-Oct-87 <i>The big picture</i> - Nathaniel Borenstein (2539)	
<input checked="" type="checkbox"/>	andrew.ms.demo.horrible.po5	✓ 23-Oct-87 <i>The detailed picture</i> - Nathaniel Borenstein (3993)	
<input checked="" type="checkbox"/>	andrew.ms.demo.listoflists		
<input checked="" type="checkbox"/>	andrew.ms.demo.nntpmail		
<input checked="" type="checkbox"/>	andrew.ms.demo.purgesent		
<input checked="" type="checkbox"/>	andrew.ms.demo.reindex		
<input checked="" type="checkbox"/>	andrew.ms.demo.wp		
<input checked="" type="checkbox"/>	andrew.documentation		
<input checked="" type="checkbox"/>	andrew.express		
<input checked="" type="checkbox"/>	andrew.gnu-emacs		
<input checked="" type="checkbox"/>	andrew.gripes		
<input checked="" type="checkbox"/>	andrew.helpsys		
<input checked="" type="checkbox"/>	andrew.hints		
<input checked="" type="checkbox"/>	andrew.informix		
<input checked="" type="checkbox"/>	andrew.kermit		
<input checked="" type="checkbox"/>	andrew.kudos		
<input checked="" type="checkbox"/>	andrew.lost		
<input checked="" type="checkbox"/>	andrew.mac		
<input checked="" type="checkbox"/>	andrew.ms		
<input checked="" type="checkbox"/>	andrew.ms.2d		
<input checked="" type="checkbox"/>	andrew.ms.batmail		
<input checked="" type="checkbox"/>	andrew.ms.cui		
<input checked="" type="checkbox"/>	andrew.ms.demo		
<input checked="" type="checkbox"/>	andrew.ms.dow-jones		
<input checked="" type="checkbox"/>	andrew.ms.pcmgs		
<input checked="" type="checkbox"/>	andrew.ms.stats		
<input checked="" type="checkbox"/>	andrew.ms.tech		
<input checked="" type="checkbox"/>	andrew.ms.tech.evs		
<input checked="" type="checkbox"/>	andrew.ms.tech.mac		
<input checked="" type="checkbox"/>	andrew.ms.version-3x		
<input checked="" type="checkbox"/>	andrew.ms.version-4x		
<input checked="" type="checkbox"/>	andrew.musicians		
<input checked="" type="checkbox"/>	andrew.networks		
<input checked="" type="checkbox"/>	andrew.newboards		
<input checked="" type="checkbox"/>	andrew.notes		
<input checked="" type="checkbox"/>	andrew.opinion		
<input checked="" type="checkbox"/>	andrew.opinion.bar-stories		
<input checked="" type="checkbox"/>	andrew.pserver		
<input checked="" type="checkbox"/>	andrew.picture		
<input checked="" type="checkbox"/>	andrew.picture.animals		
<input checked="" type="checkbox"/>	andrew.picture.cartoons		
<input checked="" type="checkbox"/>	andrew.picture.cupart		

The Andrew message system is, not surprisingly, internally complicated. The drawing below depicts these complications hierarchically. At the top level, it simply shows the five major types of components of the system, which run on five different categories of machines. By using the zip hierarchical drawing editor, you can 'zoom in' on the various parts of the picture to see more detail about how each machine's function is structured internally.


4. Snapshot of *messages composition* window. The message being created contains a raster image.

messages Sent harmarville

To: Andrew Palay <ajp+@andrew.cmu.edu>  
Subject: Big Cat

Won't Keep Copy  
Won't Clear  
Won't Hide  
Reset

Knowing your fondness for big cats, Here's a picture I recently found.



Checkpointing message server state... done.

5. Snapshot of an ez window containing a number of embedded objects (text, equations, and an animation) within a table that is contained inside of text.

~/docs/pascal.txt hammarville

This is an example text component that contains a table. The table contains a number of other components including another text component, an equation and an animation. It also shows off the spreadsheet capabilities of the table.

Pascal's Triangle						
This table contains several descriptions of Pascal's Triangle. It contains a set of equations which defines the values of the triangle. It also contains an animation showing the building of the triangle. Finally there is an implementation of Pascal's Triangle using the spreadsheet facilities of the table object.	$v_{0,j} = v_{i,0} = 0$ $v_{1,1} = 1$ $v_{i,j} = v_{i-1,j} + v_{i,j-1}$					
In order to run the animation, click into the cell and choose the animate item from the menus.	1	1				
	1	2	1			
	1	3	6	1		
	1	4	10	6	1	
	1	5	15	10	6	1

The End