ACE: Zen and the Art of Application Building

Craig L. Zarmer, Bonnie A. Nardi, Jeff Johnson and James R. Miller

Hewlett-Packard Laboratories
Human-Computer Interaction Department
1501 Page Mill Rd., Palo Alto, CA 94304-1126
415-857-8783
zarmer@hplabs.hpl.hp.com

Abstract
Task-specific application development environments enable end users to create their own applications. This is advantageous in two ways: users can draw on their own rich task knowledge to create the applications they really want, and reliance on the scarce, expensive expertise of professional programmers is greatly reduced. Extensible systems such as spreadsheets and statistical packages provide a good model for application construction as they allow end users to create complete applications. Such environments eliminate the need for separate user interface builders; the interface is seamlessly created as the application is developed. In this "Zen" process, there is little difference between application development and user interface development. Further barriers are broken down by creating application development components that can continually be edited and refined, so that distinctions among "editing," "building," "application construction," and "finished application" begin to disappear. In this paper we describe ACE, an architecture for building task-specific applications, and the software libraries we have developed to implement this architecture. We show how ACE supports the building of task-specific applications via a range of extension mechanisms from interactive editing by end users to programmer-defined subclassing.

1: Introduction

There is a tremendous demand for the ability to create, customize, specialize, and extend applications. In particular, end users want these capabilities; they have the detailed task knowledge necessary for creating the knowledge-rich applications they want, and it makes sense for them to have a central role in the development process. We know that end users can be empowered to build their own applications — often surprisingly sophisticated applications — because of more than a decade of widespread use of extensible systems such as spreadsheets, statistical packages, and CAD systems. However, only a handful of such systems currently exist. What we need now is to create many more such extensible task-specific application environments that serve smaller, but nevertheless highly critical domains; for example, a chemist's workbench, or a physician's workstation, or a research library search-and-request system, to name but a very few.

Our approach is to view application development as a multi-level process of specialization, where end users can participate in the higher levels, closest to the application itself. In fact, the process begins with end users, who do the bulk of the application development, calling in more skilled developers to make sophisticated specializations if and when they are needed, in the context of the basic application already developed by the end user. Professional programmers, in turn, are called on to create the extensible environments that make end user development possible.

There are several examples of this type of application development already, most notably, spreadsheet development [9,10]. End users start with an empty spreadsheet, and by adding formulas, text and numbers, build a variety of task-specific applications from budget forecasts, to tax planners, to engineering simulations. When the requirements of a particular application go beyond formula editing, a more experienced spreadsheet developer may be called in to write small bits of more sophisticated code: for example, macros, or fancy graphs, or a new format for showing cell values. The work of the more experienced user is an adjunct to the basic application developed by the end user, unlike traditional development in which a professional programmer does the bulk of the development.

Although spreadsheets have a great deal of extensibility, they also have limitations. The spreadsheet itself is not extensible. It is impossible, for example, to create new types of objects (e.g., an application-specific object such as a transistor) for use in spreadsheet cells, or to create spanning rows and columns, or to change the cell layout policy so that cells expand to fit their contents automatically.
Another limitation of spreadsheets is that they are only appropriate for a subset of all possible applications. We seek an underlying architecture that can support a number of domains and provide for a wide range of specializations, to be made by a range of users working together.

This paper presents ACE, an “Application Construction Environment.” The ACE architecture supports the process of developing applications through multi-level specialization. ACE consists of 3 layers:

- **ACEKit**: a class library that provides enough information about objects to make them manipulable in editors. It is used by professional programmers to create specialized application construction environments for end users.

- **Visual formalisms**: frameworks for visual editing that are used interactively by end users and programmatically by professional programmers.

- **Extension languages**: task-specific textual languages with operations mapped directly to users’ tasks.

ACEKit (“Application Construction Environment Toolkit”) makes it possible for any object to be “builder-readable” — that is, to have the necessary information about its operations available at run-time so that a builder can use the information. A builder can then determine how to instantiate an object, which operations exist for it, the types of arguments required for each operation, and how to test potential argument values for suitability.

Visual formalisms provide the frameworks needed to create visual editing environments. Visual formalisms are diagrammatic notations with well-defined semantics for expressing relations. They include visual/semantic objects such as tables, graphs, plots, panels, and maps. Visual formalisms provide an excellent basis around which to organize whole applications, or large pieces of an application. We have noticed that many real-world applications (computer and non-computer) are organized not around pushbuttons and menus, but around the more complex visual formalisms that provide effective displays of relational information. For example, a spreadsheet is organized as a table, while a circuit design tool might be a graph, and an airplane cockpit (real or simulated) a panel.

ACE visual formalisms, themselves written using ACEKit, are application building blocks. Each type of visualization has its own uses, its own rules, its own semantics. A table makes sense for some applications, while a graph is best for others, and still other applications require maps or plots or other visual formalisms. Users begin application development by selecting the specific visual formalism that seems most suitable for their application, and then they interactively edit and specialize it until it meets their needs.

Extension languages, such as the spreadsheet formula language, allow end users to add behavior to an application. A loose definition of an extension language is a language that can be used without access to the core application sources, and without needing compilation. A successful extension language will go beyond that by coupling to the structural components in a visual formalism, and by providing control constructs appropriate for end users. An extension language framework can provide a common language infrastructure while allowing each visual formalism to add its own primitives.

The alert reader will have noticed that we have not yet used the term user interface. This is intentional, for two reasons.

First, despite all of the current efforts aimed at making user interfaces easier to build, we believe that an extreme focus on an application’s low-level, widget-oriented user interface is not the best way to get a good application. In a task-specific application, what the application does is at least as important as the user interface. Consider, for example, the clumsy user interfaces of early and even current spreadsheets — which have in no way deterred users from using spreadsheets.

Second, ACE does address user interface development, but in a more integrated fashion than do tools such as UIMSs, UI toolkits, or builders. We will discuss each of these UI tools and show how ACE integrates their functions into a complete application development environment. In particular, we will show how the need for dedicated UI builders can be eliminated. This is because we completely merge the notion of interactive application editors, such as spreadsheets, with UI builders, yielding an extensible application builder/editor. ACE makes builders a state of mind in which application development, building, and editing meld into a common “Zen builder” process.

We have built prototypes of the first two ACE layers, ACEKit and visual formalisms, in C++ on Hewlett-Packard workstations. Together, these prototypes will allow programmers to develop task-specific applications with less effort and with more built-in flexibility than other tools and toolkits. We have also done background work on the desired qualities of an extension language.

In this paper we present a justification for building task-specific applications. We then review the current techniques used to build these applications, focusing on the user interface aspects of application development. Next, we present our key architectural layers, ACEKit and visual formalisms. We show how they are used to implement a generalized, extensible tabular application. Finally, we describe our findings on extension languages and show how they relate to visual formalisms and ACEKit.
2: The Need for Task-Specific Applications

A carpenter will have several different types of hammers in his or her shop. A cook will likewise have several different knives. Each is dedicated to a specific task. Generic hammers exist — one can always pound things with a brick or the back of one’s shoe — yet given the chance we’d prefer to select a tool that is designed specifically for the task.

The same is true of computer applications. A generic computer tool supports only the least common denominator of the objects and operations a user might want to work with. For example, a text editor understands only characters and provides operations to manipulate only characters. It can be used to edit what is really an outline, or a slide for a presentation, or a piece of C++ code, but in each case the extra semantic mapping is made by the user to the primitives offered by the text editor. The desired indentation and numbering in an outline, for example, must be handled by manually entering spaces or tabs, and manually editing the numbering as the outline changes.

An outline processor would clearly be more useful for the task of editing an outline. What may not be as obvious is that it should also be easier to learn and easier to use. The objects and operations an outline processor provides to the user should be closer to those expected for the task of outlining. Being closer to the task, such operations require fewer manipulations, and are also more familiar to the user.

A few objections might be raised to the idea of a multitude of task-specific applications. How can we maintain consistency over applications, especially consistency in the user interface? And if users routinely customize all of their applications as much as they customize Unix, how will anyone ever be able to use anyone else’s applications? Who said users were any good at application design anyway?

Let’s first look at consistency. If users are going to use task-specific applications, won’t this imply that they will have to learn hundreds of different applications, each with a task-specific and therefore non-standard user interface? Our answer is that consistency, like application development itself, can be addressed at several levels. At the lowest level, it would be a good idea if every tool had a similar way of invoking common operations like “quit.” The pushbutton, menu item, or whatever the world agrees to use, should be the same for almost every application. At higher levels, the answer has two parts. First, naturalness and task-matching are more important than blind imposition of consistency rules [4]. Outlines and slides are different; the manner of selecting a component in each might well be different. Second, there will be consistency within related application families. For example, all of the spreadsheet-based applications share a basic user interface, standardizing the basic tabular format. The creation of a framework for building these editing environments will foster consistency across families as well.

In terms of application sharing and quality of design, often what users want to do is to create simple throwaway applications just for themselves, such as a mortgage analyzer built using a spreadsheet. We want to make it possible for users to create such personal “scratch paper” applications, as this kind of experimentation is a tremendous aid to problem solving, and one of the most useful things one can do with a computer. When an application goes beyond personal use, then sharing is important, and the overall design of an application is critical. Applications that take on importance to an organization will need support, just like any conventional application. A study of CAD users [3] showed that many organizations already provide formal support for CAD solutions. Local developers collect and disseminate, as well as develop, extensions. They maintain the quality and standardization of these new CAD-based applications.

It is important to note that these local developers are generally not professional programmers — they have grown from within the ranks of CAD users, and bring with them essential domain knowledge.

Having argued that task-specific applications are desirable, we must now show that they are practical to develop. The next section follows two threads of application development, one based on programmers and one based on end users.

3: Current Approaches to Application Development

Who will build task-specific applications, and how? We observe two trends in application development environments. On the one side, various tools have been developed to speed the process of traditional programmer-based development. An important class of tools has focused on the user interface, which is regarded as the major portion of an application development effort. The other trend is towards giving end users more abilities to develop their own applications. In this section we will compare these two trends, and argue that a merger of the two is both possible and desirable.

Programmer-oriented tools for user interface development come in three varieties: user interface management systems (UIMSs), UI toolkits, and builders.

UIMS provide complete separation between an application and its interface, and provide a narrow communication channel between the two sides. By separating the user interface from the rest of an application, a special, and presumably easier-to-use, interface description language can be used to construct the user interface. In theory, interface description languages can be employed by user interface designers who are not trained as programmers. A problem with UIMSs is that in making the interface language easier to use, they necessarily restrict the types of interfaces that
can be built [15]. The narrow communication channel between application and interface components also restricts the type of semantic feedback that an application can give. Hix [6] provides an overview of a number of current research and commercial UIMS systems.

UI toolkits such as InterViews [7] and Motif [12] provide more flexibility in constructing task-specific interfaces, but they are complex, require programming, and address only the lowest levels of an application’s user interface — where the pushbuttons go and which function to call when one is pressed. Many important interface decisions, such as which operations in an interface are noun-verb and which are verb-noun, are left to be part of the application code, if they are thought about at all. Garnet [8] makes it easier to create task-specific interface components but the overall interface building process still requires significant programming skills.

More recently, a number of interactive applications called interface builders have been developed. They exist in connection with either a UIMS language or a UI toolkit. They allow interface components to be interactively composed, and may even allow some behavior to be specified (e.g., when this menu item is selected, display this dialog box). Their output is either the set of UIMS instructions needed to rebuild the interface, or, for a toolkit-based builder, the source code that can be compiled to produce the interface. UIM/X1 from Visual Edge and Interface Builder2 from NeXT are two commercially available examples.

Builders can make UIMSs and UI toolkits easier to use, and they can improve the turn-around time for making a change to the interface. Builders with advanced abilities to support interface behavior can serve as vehicles for rapid prototyping, allowing some aspects of an application’s interface to be tested before the application itself is written. Despite these features, builders cannot eliminate the basic problems of their underlying UIMSs and UI toolkits. UIMSs are too restrictive in the application-interface protocol to provide good feedback, and UI toolkits remain at too low a level. If the UIMS or UI toolkit allows new interface components to be added as needed, their builders have to be modified to work with these new components. Most importantly, builders remain a programmer’s tool. Builders alone will not lead to the proliferation of task-specific applications created by end users that we hope to see.

At the same time that programmers’ tools have been moving towards interactive environments, end user applications are becoming increasingly specializeable. Mechanisms for specialization range from relatively simple interface customizations, such as switching the meanings of the mouse buttons to better accommodate left-handed users, to complete embedded Lisp interpreters. Many systems have demonstrated that significant specialization can be done without resorting to a full-blown programming language. The formula languages in spreadsheets and the paragraph style sheets in document editors give end users the ability to make extensions without knowledge of a general programming language, or even awareness that they are programming.

4: Builders vs. Editors

What exactly is the difference between a programmer developing an application with a UI builder, and an end user developing an application in an environment such as a spreadsheet? Both use the interactive abilities of their tools to select the types of at least some of the objects and specify their placement. The programmer adds behavior by writing C code (for example) and compiling it; the end user adds behavior by writing formulas within an interactive environment. The programmer has more flexibility, but the resulting compiled program is frozen in terms of abilities. The spreadsheet developer has less flexibility, but for tasks where spreadsheet operations do the job, has far less code to write, because the spreadsheet has built-in support for common spreadsheet-based tasks. The process of developing the user interface to a spreadsheet is particularly well supported, as much of the required user interface is subsumed by that of the spreadsheet itself. Other interface decisions, such as component layout, are naturally integrated with the process of composing the semantic structure of a spreadsheet. Most importantly, the spreadsheet application runs inside the spreadsheet, and thus the end users of the application continue to have all of the development tools available to them. For the programmer to provide that service to the application’s end users, the programmer would have to ship the builder and the C compiler with the program (and end users would have to learn to use them!).

The tabular model of a spreadsheet is appropriate for only a subset of applications. Other specializable editors, such as CAD systems, are needed to support other domains. One way to integrate the builder and editor application development approaches is to provide a hierarchy of specialized editors that can be further specialized through programmatic means (using general programming languages), task-specific extension languages, and interactive direct manipulation techniques. In this combined approach, the distinction between an application and its builder is blurred. These specialized editors are like builders in that they provide interactive ways of selecting and arranging components, and they are like editors in that they do not require a separate compilation — the final application is the current “contents” of the editor. If a programmer specializes a builder/editor, the resulting application is still a builder/editi-
The use of such builder/editors provides benefits to three different constituencies: those who write builders, those who write applications using builders, and those who use the applications. Builder writers don't need to worry about generating source code that can be compiled; they just need to be able to save state. The hard part for a builder then becomes knowing how to work with all of the component objects that have been written, and being open to working with new component objects that are developed later. For application developers, there are several builders to start from, each of which already has some application-level semantics as well as appropriate component layout techniques built into it.

End users benefit by having a builder/editor come with the application. It means that in many cases, an end user can start development using the builder/editor that most closely matches the task, and then (if necessary) call in a programmer to extend the builder/editor to provide any extra functionality. This reversal of development roles, wherein end users build as much as they can, and then call in programmers to add more, smaller pieces to the application, is frequently seen in the spreadsheet world [10] and can be extended to other types of applications.

There are a few examples besides spreadsheets that have some builder/editor aspects, most notably, environments like HyperCard. Like spreadsheets, HyperCard provides an integrated build/edit environment that is present throughout development and use of an application, and includes an extension language. HyperCard also has the same limitations that spreadsheets have: the HyperCard environment itself cannot be extended, nor can new types of component objects be introduced.

Neither spreadsheets nor HyperCard provide a framework for developing what is really needed: a family of specialized and further specializable builder/editors. We need to support programmers in the development of specialized builders, and then let users develop applications with the builders. We now describe two layers of the ACE architecture that support this vision of application development.

5: ACEKit

ACEKit is designed to support and enhance a C++ application development effort. Developers implement classes representing the objects users will work with. ACEKit captures information about the classes and the C++ member functions on those classes that represent the end user interface. The information is used by some ACEKit classes to create part of the user interface, and it is also available to other objects, such as builders, to allow interactive application construction. A complete description of ACEKit appears in Zarmer [16]. We will focus here on how ACEKit provides information needed by builders.

The minimum requirement for making application components "builder-readable" is to have run-time descriptions of the component classes and their functions. If the builder is going to additionally serve as a run-time execution environment, it will need a way to instantiate an object, verify an object's class, and generically invoke C++ member functions. In a language like Lisp, some of this is easy; C++ cannot do this at all.

We begin by doing what many other C++ toolkit builders do: defining class Class and some facility for getting instances of class Class hooked up with the right objects. A Class object can optionally contain constructor information: a set of Class objects representing the arguments and a pointer to a function that will do the construction. Macros are provided to create both the Class object and the constructor support function, hiding these details from the developer.

A central class manager keeps track of all Class objects and provides searching operations. For example, a builder might want to find all classes that are subclasses of class Presenter, whose first constructor argument is a subclass of class Table. A builder could create a menu, based on the actual classes that were linked into a given executable, that displayed the available table presenter classes. A builder can then make an instance of the chosen class. If a new application class is later linked into the executable, it will become immediately available to the builder without any reprogramming.

ACEKit goes beyond standard run-time typing mechanisms by providing descriptions of two kinds of C++ member functions as well. Operations are member functions that an end user can invoke. These take arguments but do not have a return value. The ACEKit description, held in class OpInfo, includes the name of the member function and a description of each argument, including a name and a pointer to the Class object representing the desired argument class. OpInfo also contains a pointer to the actual C++ member function, so that a builder can invoke the operation. OpInfos can optionally contain pointers to additional member functions that can be used by user interface components to provide feedback about argument choices.

ACEKit can also describe a member function called an Accessor, and its description is held in class AccessorInfo. Accessors are used to obtain values from an object, so that they can be presented or otherwise acted on. AccessorInfos contain a name and a pointer to the Class object for the return value.

Each object can be queried for its set of operations and accessors. A builder can provide a list of available operations and help a developer specify input objects for the

---

1. HyperCard is a registered trademark of Apple Computer.
operation's arguments. Similarly, it can provide a list of accessors and help a developer select an appropriate presenter for a particular accessor. Imagine that a table presentation class has some user-settable state, such as whether to draw borders around the cells or not. While the table presentation class could have been implemented to present this state, this would limit a developer's options. Sometimes we will want to see just the table, without all of its editing state presented. By providing an accessor and using an external component to present this state, developers will be able to combine tables and table presenters with other UI components to build the precise application desired. The use of OpInfos and AccessorInfos helps keep core classes more reusable.

5.1: Interface uses for operation descriptions

Besides supporting builders, we can use an object's OpInfos to build reusable, high-level interface components. ACEKit provides the class Command to control the way arguments are gathered and when feedback information is used (if present), and to settle other gesture-independent UI style issues. For example, a Command can be set either to require explicit confirmation before invocation or to invoke the operation immediately when all arguments are available. A Command can be designed to complain about improper argument values as soon as they are entered, or to wait until confirmation is given. Any command style can be associated with any operation because an OpInfo has all of the information needed by any command.

Similarly, ACEKit provides class Editor that adds a currently-selected-object interface to any application. A developer tells the editor which arguments of which commands should come from the currently selected object, rather than directly from a user.

5.2: Input and output

ACEKit can work with any UI toolkit. The InterViews [7] C++ toolkit fits particularly well, because it makes it easy to create specialized objects that present application state. It also facilitates constructing application-specific input gestures.

Through ACEKit, both input and output objects support the same run-time typing that application objects use. This enables a builder to browse for existing presentation types and instantiate them as needed, without having to rewrite the builder as new presenters are defined. Input objects also contain descriptions of the events they generate. For example, a pushbutton has a single event, a drag recognizer has two, and a menu has one for each menu item. This information enables a builder to know how each input device can be connected.

5.3: Summary: Supporting builders

Through the mechanisms described above, ACEKit enables builders to:

- Browse the class hierarchy
- Instantiate any class
- For any object, determine
  - which operations exist, the arguments they require,
  - feedback hooks that are available
- which data values can be accessed for presentation
- Set style parameters for the interface to a specific operation
- Select and connect input gestures for an operation and its arguments

ACEKit allows any C++ class to become usable by any ACE builder without requiring builder modification. In addition, it means that the job of writing a builder has been greatly simplified, making it feasible to design specialized builders. It allows the builder to operate using the actual application components, not just interface widgets (or pictures of interface widgets), as in some builders). A builder using ACEKit components can easily invoke a component's operations. In short, ACEKit enables a builder to serve as the finished application.

6: Visual Formalisms

We have observed that there are some structures for working with a set of components that keep appearing across many very different applications. These structures are inherently visual, yet they have a clear set of rules governing their editing. These structures are what we call visual formalisms, and they are time-tested, intuitive frameworks for organizing information [5], including, for example, tables, graphs, plots, panels, and maps. In Nardi and Zarmer [11], we present the argument for visual formalisms from a user interface perspective. Here, we will summarize the characteristics of visual formalisms, describe the class structure used to specialize and assemble them in our ACE implementation of visual formalisms, and show how they can serve as both builders and application building blocks.

Architecturally, there are four key features of a visual formalism. A visual formalism must be interactive and editable. These features apply to the visual formalism itself as well as to the content objects inside it. Being interactive and editable means that a visual formalism enables users to make changes and immediately see the results.

A visual formalism must also support task-specific content objects. While many applications can be modeled with simple integers and strings, it is the ability to put more specialized objects inside a visual formalism that provides more power for task-specific applications. Content objects
in a visual formalism are not just drawings, or interface widgets. They are full-fledged application objects with their own semantic operations; for example, transistors and resistors in a circuit design application, or innings and players in a baseball application, or patient records and treatment regimens in a clinical application. Content objects can even be other visual formalisms. An application might start, for example, with a panel on which components are placed, including perhaps a table or graph or plot.

Finally, visual formalisms must provide support for relationships between content objects. These can be implicit, as in the rows and columns of a table, or explicit, as in “C3 = C4 + C5,” where C3, C4, and C5 are mapped back to visual formalism components, e.g. cells in a table or nodes on a graph. These features all work together to make visual formalisms effective visual frameworks.

Is a visual formalism a widget, in the interface toolkit sense? While in principle it could be, architecturally this would be undesirable. Application development is much more successful for end users if they work within an overarching visual framework such as the table in spreadsheets, or graphic structures in CAD systems. To bear the weight of an entire application, or a major piece of one, a fairly sophisticated visual framework is needed. The widgets provided by current UI toolkits are fine for presenting simple state or triggering a single function, but their internal architecture is not designed to support complex relationships typical of end users’ real applications. Furthermore, as will be shown in the following section, the desire to make visual formalisms easy to specialize dictates that we use a more modular architecture than a single widget per visual formalism.

Builders, UIMSs, and UI toolkits start from the premise that because so much time and energy are devoted to user interface development, programmers should have better widget sets. Our premise is quite different: application development time can be reduced by putting application development into the hands of end users. This suggests a radically different solution. Users are not going to be able to create applications by moving widgets around; they need much more help than that. A major piece of what they need is a good visual editing environment capable of showing complex relational semantics, and capable of providing a problem-solving framework (see [11]). That is precisely what frameworks such as tables, graphs, plots, panels, and maps do. Working within these frameworks, end users can create their own applications.

6.1: Specializing visual formalisms

Although objects such as tables and graphs are clearly useful, they are much more useful when they can be specialized to meet the needs of a particular application. Specialization includes (1) changing the structure of a visual formalism itself (such as adding a spanning row to a table, or changing the width of a column, or pruning a branch from a tree) and (2) adding or changing the content objects of a visual formalism (such as numbers and text fields in table cells).

We are implementing a set of visual formalisms in ACE-Kit. Each visual formalism consists of a small library of classes. ACEKit descriptions of the classes, operations, and accessors available make it possible for visual formalisms to serve as builders as well as components that are placed within other visual formalisms.

For some applications, the specialization process will require only editing content objects: adding and removing components, and connecting them together. This is what spreadsheets and word processors allow users to do now. Depending on how flexible a visual formalism is, editing its structure is possible; for example, in most spreadsheets column widths can easily be specified, though spanning cells cannot be created.

More complex applications will require extending a visual formalism, or the components within it. A family tree editor, for example, would probably be implemented by extending a graph visual formalism. A basic graph visual formalism would provide only generic operations such as add and remove node, while a family tree editor should incorporate the additional semantics of genealogy, e.g. an operation to “add descendant,” and the constraint that you cannot be your own parent. A full programming language is usually required to make this kind of extension.

The structuring of ACE classes that make up each visual formalism has been designed to minimize the programming effort required to specialize a visual formalism. An important aspect of the class structure is the separation between a visual formalism and the objects managed by the visual formalism. Current spreadsheet programs have a limited set of object types that can be used in a cell: a number, a text string, or a formula. Trying to create a new type of object and integrating it with a spreadsheet would be a major undertaking — if the source code to the spreadsheet were available at all.

In our visual formalisms, content objects are C++ objects completely separate from the visual formalism objects. Any type of ACEKit-based object can be managed by a visual formalism, including types developed long after the visual formalism is developed. ACEKit provides all the information a visual formalism needs to work with the new content type, including updating the user interface to make the new content class an option for users to select. This allows visual formalisms to be extended without any source access at all — a new application is created by linking the compiled visual formalism classes with the new content object classes.
Many applications can be built with just this degree of flexibility. However, when the functionality of a visual formalism itself must be extended, visual formalisms can be specialized through subclassing. To make this process easier, visual formalisms are implemented as a set of classes, each with responsibility for a few aspects. Typically, only one or two of the core visual formalism classes would need to be subclassed to add or refine behavior. In addition, we have made careful use of C++ virtual methods so that even within a single class extension is straightforward.

6.2: Visual formalisms as builders

Visual formalisms are the basis for application building blocks, serving as specialized editors for any kind of content objects. In this section, we will show how visual formalisms can also serve as builders, in the traditional UI builder sense, and how they can actually go much further to develop an entire application.

Builders allow users to create objects, place presentation objects on the screen, and make basic connections between interface events and application functions. Visual formalisms do that too. Developers populate a visual formalism by selecting classes to instantiate and adding instances to the visual formalism. The choice of classes includes all ACEKit-based classes, including other visual formalisms and any application-specific classes. Unlike other builders, users work with a combined semantic/presentation pair. For example, when a boolean object is added to the visual formalism, a default presentation class for it is added as well. The choice of presentation class can be changed at any time. For a boolean object, a developer might choose between a textual view and a pushbutton.

The insertion of objects into a visual formalism follows the rules of editing for the particular type of visual formalism. For a panel, users can place component objects anywhere and in any size they like. For a table, users pick the specific cell that will hold the content object, change row and column sizes, and split or join cells together to obtain the desired structure to hold a component object.

To specify the user interface, developers connect operations to objects with input abilities, by way of ACEKit command instances. ACEKit-based objects have full run-time-readable descriptions of their operations and accessors. ACEKit-based input devices, such as pushbuttons, additionally have run-time-readable descriptions of the input events they can generate. A pushbutton can be connected to activate, deactivate, or supply a boolean argument to an operation on any object. These connections can all be made interactively.

The state of a visual formalism, including its interface connections, can be saved. To run the application that was just built, no compilation is required — simply start up the visual formalism and open the saved application.

When developers produce subclasses of visual formalisms, they automatically produce a specialized builder at the same time. The following section describes a particular visual formalism, the Table Visual Formalism, in more detail to show how it can serve as an application building block, editor, and builder, and how it can be readily specialized when needed.

7: An Example Visual Formalism: Tables

To make the expressive power and ease of extensibility of visual formalisms clearer, we describe the Table Visual Formalism class library. The details in this section would ordinarily be of interest only to a programmer who needed to make a significant extension. As will be shown in the following sections, end users of ACE tables will see only the integrated visual formalism.

ACE tables give developers high-level functionality for developing applications for creating, structuring, modifying, editing, and browsing tables. Unlike the work of Beach [1] and Cameron [2] whose programs draw complex tables, we are concerned with providing object-based table structures that can be active application components, not static pictures. Our tables exist not only visually, but also semantically.

An ACE table is defined as a rectangle filled with contiguous, non-overlapping rectangular cells, each of which contains a discrete piece of information (a content entry). A content entry is any object, e.g., an integer, string, bitmap, or instance of an application-specific class. Each content entry has a semantic and presentation component. A table need not be a grid; cells can span multiple rows or columns of other cells.

Both the table as a whole, and each cell within the table, are components with their own presentation and semantics. The presentation and semantic aspects of the table are implemented through six classes:

- Table represents the semantics of a table as a whole, in particular adjacency information about cells. The operations on class Table deal mainly with changing the structure of a table — the arrangement of rows and columns. The most basic operations and their arguments are SplitRight(Cell), SplitBelow(Cell), and Join(Cell, Cell).

- Cell represents an individual cell of a table. Users will normally be unaware of cell objects — they will only think of the content object (the integer, string, etc.) that is in the cell. A cell can also have names, relational expressions (to express relations within the table), or other attributes associated with it.
- **TableView** is the visual presentation of a table. It translates the semantic adjacency of cells to a visual arrangement of cell views. There can be more than one TableView instance for a given table to provide multiple presentations of the same table. Class TableView provides operations that set a row or column size, filter and unfilter rows/columns, and support scrolling.

- **CellView** presents a cell. The generic CellView class provides visual feedback (e.g. highlighting) for selected cells (for cells selected with mouse or keyboard), and determines how to present the cell’s content object.

- **TableEditor** supports the editing process, providing interface state, such as a currently-selected-cell, which must be maintained on a per-user basis.

- **TableEditorView** is a presentation of a TableEditor object. It provides a graphical interface for a table editor’s editing operations and a presentation of interface state, such as a presentation of the currently selected object.

A table application that does not require interactive editing capabilities can be implemented with the first four classes. A table that will be edited will make use of the last two classes as well. By breaking down the functionality of a table into these six classes, programmers can make small, localized changes to their own subclasses of the basic visual formalisms to get the precise application desired.

Other visual formalisms have a similar class structure. Most of the differences among them are in the equivalents of classes Table and TableView.

8: Extension Languages for Builders

Though our development efforts to date have been devoted to creating ACEKit and the Visual Formalisms libraries, we are planning to provide extension languages for builders. Minimally, an extension language must allow users to add behavior to their systems without having access to core sources, and without compilation. These features make end user development possible, but we believe they are not enough to make it probable. In this section, we discuss those additional characteristics that are important for any end user extension language, and compare some current approaches to end user programming against those characteristics.

Our ideas about extension languages and environments come from our study of spreadsheet users [9,10]. One cannot argue with success: almost every computer user has access to a spreadsheet, and most have written at least a few simple spreadsheet applications. Spreadsheets and the spreadsheet formula language offer users:

- Task-specific semantic primitives
- Simple control structures
- Good visualizations of program state

The spreadsheet formula language contains a number of ready-to-use functions such as SUM. Although modern spreadsheet packages usually contain over a hundred primitives, we found that the average spreadsheet user normally used fewer than ten functions in their formulas [9]. The presence of a well thought out set of primitives helps those without programming experience or interest make rapid progress on their application.

Spreadsheets also avoid complex control structures, such as “for” loops and recursion. This is partially due to the design of the primitive operations, and partly due to the close coupling of the formula language to the tabular presentation. Users do not add columns by writing a “for” loop that iterates over row numbers, they just type SUM and drag out the column to be summed. Spreadsheets remove an even greater programming burden by providing automatic dependency checking. Users are not forced into the linear model of conventional languages. They are able to put small, focused pieces of code in a graphically meaningful arrangement, and the spreadsheet handles the flow of control.

Finally, it is not enough to have a good language. There must be a supportive development environment. Spreadsheets are a naturally good development environment because they help users visualize their program’s state. The key variables are all on display, in a semantically meaningful arrangement, and their values are updated continually. Changes can be made quickly and the results seen just as quickly.

The bottom line is that end user programming is not just a language issue. It requires careful integration of language with a visual framework. Thus, our work on visual formalisms contributes directly towards our vision of end user application development.

Now let’s compare some other approaches to end user development with spreadsheets. One approach is to incorporate a general purpose interpretive language, such as Lisp, into an application. GNU Emacs [13] and AutoCAD [14] are two examples. While these meet the minimum requirements, and many programmers have made significant extensions to these environments, few end users do. Of the three criteria listed above, only one is supported: the inclusion of task-specific primitives. GNU Emacs, for example, has text editing and file handling primitives added to the core Lisp functions. The need to master Lisp, however, is enough to keep most end users from creating the task-specific applications they desire in these environments.

What about HyperCard? HyperCard with HyperTalk is a compromise between a general programming language and
a specialized extension language. HyperTalk doesn’t do everything that a general programming language does, but it does provide a friendlier syntax than, say, C++, and, like spreadsheets, is interactive. But there is very little in the way of commercial stackware. To create stackware, end users are faced with learning HyperTalk, which is almost as complex as a conventional programming language and requires mastering basic computer science concepts. The problem may be that HyperCard-like environments are actually a bad compromise: they have the complexity of conventional programming languages, but lack their speed; and they do not offer the application-specific framework, components, and language found in environments such as spreadsheets, CAD systems and even GNU Emacs Lisp.

In summary, the lessons learned from the spreadsheet world can be translated to other domains to provide good end user development environments. In the next section, we show how ACEKit, Visual Formalisms, and extension languages based on them work together to support development from end users through programmers.

9: Building Applications with ACE

Once a set of visual formalisms and a basic set of component objects (e.g., integers, strings, dates) have been implemented, the process of building a real application can begin. In the simplest case, a user selects a visual formalism for the outer framework of the application and begins to edit it. The editing may involve both altering the structure of the basic formalism (e.g. adding spanning cells to a table) and putting content objects inside the visual formalism. Users create instances of objects (visual formalisms or widgets or application-specific data types) and arrange them inside the outer visual formalism. A "While You Are Out" memo, for example, could be created by starting with a panel, adding string and date fields, and saving the panel (see Figure 1). Notice that by selecting a semantic type, such as date, proper type checking will automatically become part of the application. As with many applications, the difference between a template and a working document is not significant to ACE. Users just copy the original form and fill it in each time they need a new one.

A more powerful application would involve using a programming language to add behavior. Such a language might be an end user language, e.g. a formula language such as the spreadsheet formula language, or a conventional programming language like C. For example, a tax application might begin with the table visual formalism. The user arranges cells and inserts content objects until the application looks like a tax form and has the appropriate kinds of cells, and then adds in some formulas until the application is a tax form. Note the advantage in starting with a table over a panel (as today’s UI builders generally do): most forms are closer to a table than a freeform canvas. The company that makes MacInTax thinks so too, and it sells the table engine used to generate the tax forms as a general forms design tool.

As another example, consider making an organization chart editor. A user starts with the graph visual formalism and begins adding objects to represent people. Relations are added to capture the organization’s structure. A head count could be computed by adding an integer whose value is computed by the formula “count (walk-tree-from [1]).” Walk-tree-from is a graph-specific function that produces a list of all the subnodes (recursively) of a given node. As with tables, this kind of domain-specific primitive hides control complexity and ties into the visual structure of graphs. Figure 2 shows a prototype of such a chart.

Another level of development involves directly working with ACEKit information. Imagine buying an ACE-based application whose command style was intolerable. A user could open the ACE browser and determine which operations are available, which arguments they require, and how they are currently being invoked. The user could then edit the existing components or add additional ones. If a favorite command is invoked by a third-level menu, a user could create a pushbutton, put it somewhere convenient on a panel, and connect the pushbutton to the same operation.

---

1. MacInTax is a registered trademark of Softview, Inc.
Figure 2. Building an organization chart editor using an ACE graph

Figure 3. Building a baseball scoresheet using an ACE table and a special AtBat object
that the menu item invoked. A command that required confirmation could be changed to execute without confirmation.

Sometimes changes need to be made to the content objects used within a visual formalism. Imagine creating an on-line baseball scoresheet. Baseball scoresheets present an amazing amount of information in a compact tabular form. For such an application, the ACE table classes themselves are adequate, but a new class is required to represent an individual "at-bat" event. Class AtBat captures the number of balls and strikes on the batter, how he progressed around the diamond, whether he made an out, and other information associated with a single chance to bat. A companion class, AtBatView, must be written to present this information graphically. As soon as a programmer implements these two classes, our table visual formalism can immediately begin working with them. Using the official Little League scoresheet as a model, we show a sample scoresheet application in Figure 3.

When the way of organizing, relating, or presenting a set of content objects is inadequate, a new subclass of an existing visual formalism is usually required. Occasionally, an entirely new visual formalism is needed. In other words, a new or enhanced builder is needed. For example, we might want the org chart in Figure 2 to use right-angle lines. A programmer is called in, who takes the graph visual formalism library and header files and creates a new subclass containing the new operation (including an ACEKit description of that operation). Once compiled, either the programmer or the end user can complete the process by defining the user interface components used to invoke the new operation. Completely new visual formalisms can take advantage of high-level ACE classes that are not themselves visual formalisms but offer useful components for beginning the task of writing a new visual formalism.

10: Summary

ACE shows how a more application-oriented notion of builders and rich run-time information can support a variety of application development needs. By blurring the distinction between builders and editors, and supplying a base set of flexible, extensible application components, ACE can:
* Provide a variety of specialized builders
* Make applications inherently customizable with the overall result that end users can take more control over application development

We can learn a lot from the success of today's spreadsheets, word processors, CAD systems, statistical packages, and other extensible applications in use by millions of users. What ACE does is to provide a model for the Zen of editing/building such applications that can extend gracefully to full programming when needed.

Acknowledgments

Thanks to Nancy Kondzielski and our referees for their helpful comments on earlier drafts of this paper.

References