End-User Systems, Reusability, and High-Level Design

Glenn S. Fowler (gsf@research.att.com)
John J. Snyder (jjs@research.att.com)
Kiem-Phong Vo (kpv@research.att.com)

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974, USA

During the past ten years, the number of computer users has grown by orders of magnitude. This has been brought about by dramatic increase in computing power combined with equally dramatic decrease in hardware costs. Beyond "stand-alone" user applications like word processing and spreadsheets, new classes of business applications arise where competitive advantage is created by empowering "end-users" with instant access to relevant information. In many cases, code already exists to access and process the desired information; the challenge is finding a way to couple such processing capabilities to individual user requests in a timely, specific, and friendly fashion. The keys to such end-user systems lie in high-level design and software reusability. This paper describes a language tool EASEL (End-User Application System Encoding Language) for building end-user systems and experiences in its development and deployment.

1. Introduction

An end-user is a user who needs to perform some computing functions but may not be well-acquainted with the underlying theories and mechanics. In this sense, even an expert computer user can be an end-user in some application domain. An end-user application is a software system designed to be used by end-users. As such, the primary characteristic of an end-user system is an interface that is easy to use, directly addresses users' needs, and hides any complexity in the solution methods. As computing techniques become more diverse and complex while the cost of computing machinery becomes cheaper, the market for end-user applications gets larger. This is particularly true in business computing where combinations of techniques from networking, databases, statistical analyses, equation solving, and graphics are routinely required.

Though computing techniques can be complex, there are a multitude of high quality software tools readily available to solve such problems. From a system construction point of view, it is desirable to take advantage of such tools in building a new application. Coupled with the requirement of a good user interface, the main challenges in building an end-user application are:

- Designing a system architecture that maps closely to users' needs
- Building an interface that reflects that architecture

- Leveraging as much as possible from existing software tools in implementation.

As successful applications often live long and beget others, additional challenges are to ensure that a system is easily evolvable, and to grow the pool of reusable software tools as systems are built. A sign of a successful software development organization is its ability to build families of applications quickly and still provide good maintenance support. This is possible only if:

- The system construction method encourages building reusable tools
- Enough characteristics of an application family can be abstracted into reusable templates
- A means exists to help the coding and maintenance of such templates.

Over the past 10 years, we have been working with a language and system to build end-user applications called EASEL (End-User Application System Encoding Language). EASEL applications use interactive constructs such as windows, forms, menus, and hypertexts to provide easy to use interface and build or reuse computational tools for other application-specific tasks. Within AT&T, EASEL has been used to build hundreds of applications ranging from simple educational systems to sophisticated applications used to plan and manage the telephone network. Some of these applications are used world-wide and support hundreds of international users on a daily basis. Early experience with EASEL was reported in [Vo90]. This paper describes the current state of the EASEL language, how its approach to system construction encourages software reuse, how language macros help build a higher abstraction levels, and relates some experience in building families of end-user applications with EASEL.

2. End-user system architecture

To see how EASEL helps to build an end-user system, we need to understand the components that comprise such a system. The logical structure of an end-user system can be divided into four layers [Vo90] as shown in Figure 1. The top two layers represent *Design Programming*, i.e., programming activities focusing on the user interface and high-level tasks as seen from the user's point of view. The lower two layers represent *Computation Programming*, i.e., activities focusing on how the high level tasks are to be implemented and with what data structures.

The *User Interface* layer allows users to manipulate the systems using well-defined and easy to use steps. A major function of this layer is to hide all differences and idiosyncrasies in the interfaces of underlying computing tools and techniques. An EASEL application builds this layer using menus, forms, and hypertexts to provide an active interface that guides users in taking appropriate computational steps.

The *System Structure* defines a task partition and relationships among tasks. In the EASEL framework, the design process of an end-user system begins with identification of tasks and subtasks as seen from the user's perspective. The high level tasks are mapped to objects known as *frames* which are interconnected in a *frame network*. Each frame defines all user

Design Programming	User Interface	Forms, Menus, Text
	System Structure	Tasks and Subtasks
Computation Programming	Computational Functions	Application Specific Code
	Data Architecture	Application Specific Data Types

Figure 1: Interactive End-User Systems Architecture

dialogs appropriate to the task and prescriptions for computing methods necessary to carry out that task. Dialogs can cause transitions to other frames in the frame network or induce certain low level computations. For example, a simple electronic mail application might consist of the following tasks:

- Getting a list of users,
- Picking addressees from the user list,
- Filling out a mail form, and
- Sending the mail.

Each of these tasks maps to a frame that defines:

- Its own set of relevant parameters (e.g., login ids, mail headers, etc.),
- Mechanisms to obtain the parameters (e.g., from the user, from login database, etc.), and
- Appropriate commands to drive lower-level computational processes (e.g., running the `mail` command).

Thus, frames focus on high level activities required to perform a given task. Actual computations are performed at a lower level by invoking applications code, utilities, and other packages.

The *Computational Functions* layer consists of utilities and application-specific code embodying the computational methods to access, transform, and generate the data necessary to accomplish the end-user's tasks as requested. The Easel language provides for string manipulation, mathematical operations, file input/output, and event handling. In addition, there are several language mechanisms to execute application specific code that may be in C or some interpretive languages such as the UNIX shell [Bou78, BK89] or AWK [AKW88].

The *Data Architecture* layer defines the types of data to be manipulated along with their storage and access methods. Specific data types depend on the application and the tools used to perform the computational functions. Data types may be tuned to support, facilitate, and optimize execution of algorithms and methods at the computational level. Examples include EASEL variables, C and C++ data types and structures, data for relational and object-oriented databases, as well as data types required by specific packages, such as statistical or queuing packages.

The four-layer architecture points out some useful insights on reusability in some broad categories of tools. Traditional UNIX tools are often designed to do single tasks that, in many cases, embody powerful data structures and algorithms. Since their interfaces are geared toward the specific computing methods being implemented, they are not easily usable by end-users although they are immensely reusable. At the other end of the spectrum, many screen-oriented applications based on screen libraries like *curses* [Arn84, Vo85] or *X* [Nye90] and various spreadsheets or database packages are easy to use, but offer little reuse because the computational methods are too intertwined with the implementation of other parts of the system. EASEL is an attempt at bridging this *reuse gap* between general purpose but hard to use tools and application specific but easy to use software.

EASEL's approach to system construction can be summed up as that of *separating Design Programming from Computational Programming.* The EASEL language focuses on expressing the high level design of a system including its user interface and task partition. Most computational details in the lower two layers are left out at this level but enough of the interface to appropriate software components can be specified. In this way, tool reuse is naturally a component of the EASEL's system construction method. As we shall see later, by bringing in a language tool for program design, EASEL also makes possible the reuse of certain high level tasks in much the same way that software tools traditionally improve computational reuse.

3. The EASEL language

The EASEL language is block structured where the block types map to certain high level activities. Some blocks define user interface components and tasks such as:

- `{frame` – a high level task
- `{context` – grouping of related activities
- `{menu` – selections to be decided by users
- `{question` – a group of questions is a form
- `{write` – display text to screen

Other blocks and statements define computations such as:

- `{action` – runs the enclosed shell script

- `{process` – runs named cooperating UNIX process and sends enclosed text to its standard input
- `~Ccall` – calls the named C function
- `~call ~goto ~overlay ~return` – transition among frames
- `~evglobal ~evlocal` – event handling
- file input and output
- string handling
- mathematical operations and functions

Other language statements provide for managing the scope of variables, manipulating the run-time UNIX process environment and keyboard bindings and macros. Another group of statements is used to modify default display attributes, such as window locations, border styles, colors, etc.

EASEL variables need not be declared and are initialized as empty strings. Most operations including those requiring communicating with external processes or subroutines result in string values. However, in cases where numerical values are required, the mathematical assignment statement `:=` can be used to do such computations.

EASEL applications are constructed in a network of frames. Each frame is defined in a frame block:

```
{frame  FrameID  ~arg1 ~arg2 ...
    ....
}f
```

Each frame has its own name or frame id, may accept arguments, may contain statements and other nested blocks, and may return values to its caller. EASEL variable names (and keywords) begin with the ~ (tilde) character.

Composite blocks provide grouping of statements and nested blocks:

```
{context  (entry):(exit)
    ....
}c
```

Control flow as exemplified above is directed via `(`*entry*`)` and `(`*exit*`)` conditions. A block or statement is executed only if its `(`*entry*`)` condition is true; control loops through the block or statement until its `(`*exit*`)` condition is satisfied. An omitted, or null, condition is taken as true.

Rather than examining the EASEL programming language in detail, we will illustrate how to build a system with EASEL using a small example.

3.1. An Easel email application

An electronic mail application serves as a small but realistic example. The main tasks of the application consist of:

- Getting a list of users,
- Asking the user to select addressees,
- Filling out a mail form, and
- Sending the mail.

The design of the frame network for such an *email* application might then look like Figure 2.

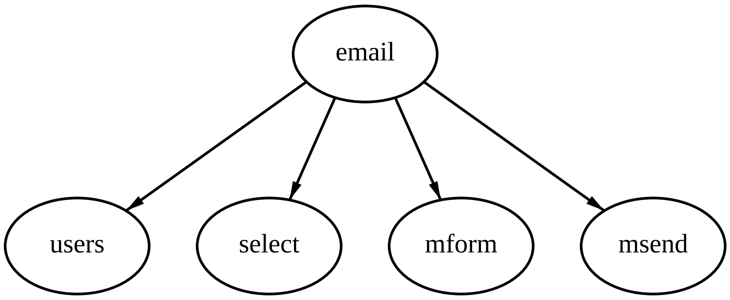


Figure 2: The *email* frame network

Below is the Easel code for the top level `email` frame. Line 1 begins a frame block for frame `email`. Line 2 calls frame `users` to get a list of users that is assigned to variable `~users`. Line 3 calls frame `select` with argument `~users` to ask user to make selections which will be assigned to variable `~to`. Line 4 calls frame `mform` with argument `~to`. This frame will return three values to be assigned to `~to`, `~subj` and `~mesg`. Line 5 begins with an entry condition that is true only when both `~to` and `~mesg` are not null; if that is the case, then frame `msend` is called to send the message to the selected people. Line 6 denotes the end of the frame block.

```
1: {frame  email
2:     ~call users  > ~users
3:     ~call select ~users  > ~to
4:     ~call mform ~to  > ~to ~subj ~mesg
5:     (~to!=~null && ~mesg!=~null): ~call msend ~to ~subj ~mesg
6: }f
```

The first task or frame called by `email` is `users`. Below is the code for `users`. Line 2 runs `/bin/ksh` as a cooperating process or coprocess. Values returned from the coprocess will be assigned to variable `~logins`. Line 3 specifies the "end-of-response" and "end-of-commands" protocol delimiters to be used between Easel and the coprocess. When Easel is done sending data to the coprocess (the text on line 5), it sends the end-of-commands text (`echo ShIsDone\n`). Easel then starts reading text returned from the coprocess until it sees the end-of-response text (`ShIsDone\n`). Line 4 says that the screen will not be disturbed during this execution so Easel will not refresh the screen when it gets back control from the coprocess. Line 5 is the complete body of text to be sent to the coprocess; on Suns running NIS (formerly Yellow Pages), this shell script will generate the current list of logins. Line 6 ends the process block. Line 7 causes control to return to the calling frame and returns the value contained in the variable `~logins`.

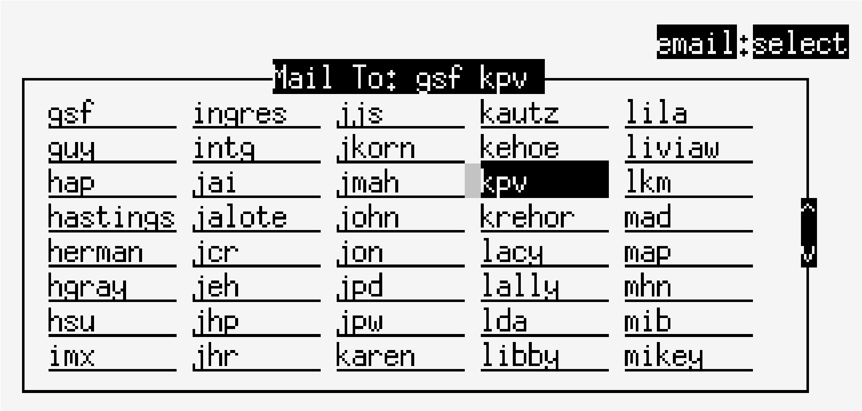
```
1: {frame  users
2:     {process  "/bin/ksh"  > ~logins
3:         ~!"ShIsDone\n","echo ShIsDone\n"
4:         ~$
5:         ypcat passwd | cut -d: -f1 | sort | uniq | xargs
6:     }p
7:     ~return ~logins
8: }f
```

The next task is to ask the user to select the desired addressees from the available list of users. Below is the frame `select` that performs this task. Line 1 begins frame `select`, which has argument `~list`. Lines 2–8 present the `~list` of logins as a menu to the end-user, builds a `~to` list adding each user selection, and loops until a null choice is entered. The `.window` statement on line 3 defines a window for the menu. If this is not defined Easel will construct some default window whose size and placement are designed to make good use of screen real estate. Line 5 provides the list of options to be shown, namely those stored in variable `~list`. Delimiters for the list are specified as space, tab, or newline characters. Line 6 builds a list of the logins selected by the user and stores the result in the variable `~to`. This variable is used in the menu's title given in line 4 to give immediate feedback to user on the current set of selections. Line 9 returns the value of `~to` to the calling frame.

```
 1: {frame  select ~list
 2:     {menu  :(~pers == ~null)  ~pers
 3:         .window( x=5, xlen=50, y=0, ylen=10 )
 4:         Mail To: ~to
 5:         {option  ~list  " \t\n"
 6:             ~to  = ~to * ~pers * " "
 7:         }o
 8:     }m
 9:     ~return ~to
10: }f
```

A screen snapshot of `select` is shown in Figure 3. Note that the frame stack `email:select` shows the traversal sequence in the frame network to get to the frame `select`. If allowed by system designers, experienced users of an Easel application can use the commands at the bottom of the screen to directly access any frame in the network.

The third `email` task is to display a mail form and let the user fill it in. Below is the frame `mform`. The context block beginning on line 2 is used to group the enclosed question blocks so they will be displayed together as a unit or form. The `.form` statement on lines 3–8 specifies that answer fields should be shown in color 2 (typically underlining on black and white terminals) with `attributes` set to `printable` characters. The `.form` statement also includes a template for the form and indicates which question variables correspond to which answer fields. For example, line 4 indicates that the answer field is the `~To` variable, which, in this case, was passed as an argument to the frame. Thus any menu selections made by



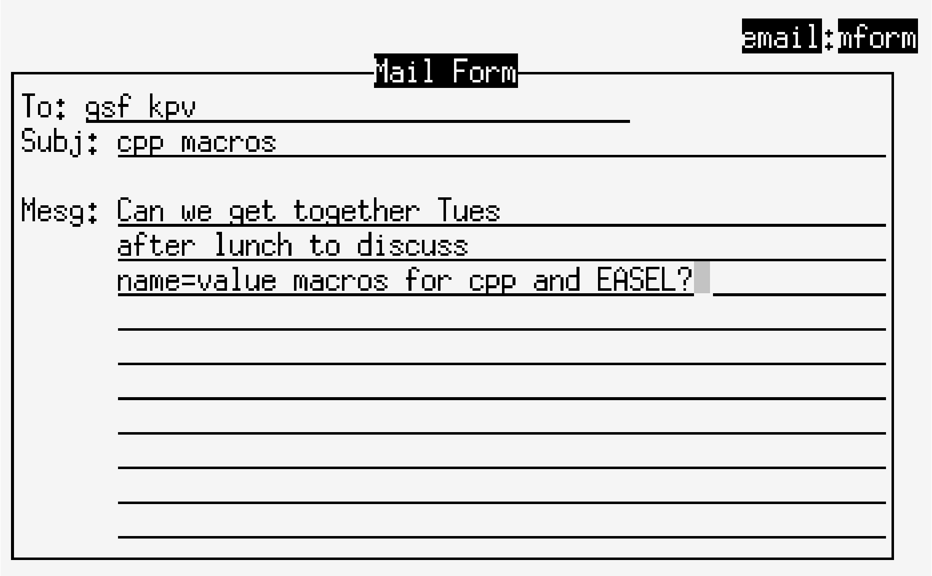
Mail To: gsf kpv

Figure 3: The `select` screen

the user in the `select` frame will be shown on the form as soon as it is displayed. The user can edit the answer field on the form or move on to the next field. Lines 10–12 provide help text for this form should the user request help. Although not shown here, help text may include EASEL variables as well as hypertext links. Lines 13–22 are the question blocks used to build up the form; each provides a variable to collect the user's response to that question. The `.field` statement in line 20 overrides the suggested template and specifies that the answer `~Mesg` field be `r`epeated 10 times for a ten-line answer. If the user types beyond the 10^{th} line, the answer field will scroll (unless the answer has been restricted to the length specified). Although the syntax for specifying a form is a bit verbose with its `.form` statement and group of question blocks, it saves frame designers from counting row and column coordinates and allows entry conditions on question blocks to tailor field access dynamically. Line 24 returns the values in `~To`, `~Subj`, and `~Mesg` back to the calling frame. Figure 4 shows a screen snapshot of the form.

```
 1: {frame  mform  ~To
 2:     {context
 3:         .form( c=2, a=p,
 4:         To: <_____________________ $ ~To
 5:         Subj: <___________________________ $ ~Subj
 6:
 7:         Mesg: <___________________________ $ ~Mesg
 8:         )
 9:         Mail Form
10:         {descript
11:             Some HELP for the form
12:         }d
13:         {question  ~To
14:             To:
15:         }q
16:         {question  ~Subj
17:             Subj:
18:         }q
19:         {question  ~Mesg
20:             .field( r=10 )
21:             Mesg:
22:         }q
23:     }c
```

```
24:     ~return ~To ~Subj ~Mesg
25: }f
```



email:mform

Mail Form

To: gsf kpv

Subj: cpp macros

Mesg: Can we get together Tues
after lunch to discuss
name=value macros for cpp and EASEL?

Figure 4: The `mform` screen

The final task, if the user has provided non-null text for the addressees and the message, is to send the user's e-mail, as coded in frame `msend` below. Line 2 gets the current value of the environment variable `$LOGNAME` and stores it in the EASEL variable `~LOGNAME` (for use in the mail message body). Lines 3–14 communicate with the same coprocess `/bin/ksh` as was set up in the frame `users`. This time, what is being sent to the shell is a multi-line mail command that is parameterized by several EASEL variables. The variable `~efsdate` in line 8 is an EASEL read-only variable containing the current date as a string, e.g., `Sat May 21 01:40:10 EDT 1994`.

```
 1: {frame  msend  ~To ~Subj ~Mesg
 2:     ~getenv ~LOGNAME
 3:     {process  "/bin/ksh"
 4:         ~!"ShIsDone\n","echo ShIsDone\n"
 5:         ~$
 6:         mail ~To <<!!
 7:         Subj: ~Subj
 8:         Date: ~efsdate
 9:
10:         ~Mesg
11:
12:         Thanks! ~LOGNAME
13:         !!
14:     }p
15: }f
```

The actual text for the mail command, as sent to the shell, is shown below. In prototyping and early testing of a frame, the process block for the mail command in lines 3–14 might be enclosed in a write block (to the screen or to a file) to simply write out the mail command as it would be sent to the coprocess to let the command be checked without sending any unnecessary mail.

```
mail gsf kpv <<!!
Subj: cpp macros
Date: Sat Apr 30 01:40:10 EDT 1994

Can we get together Tues
after lunch to discuss
name=value macros for cpp and Easel?

Thanks! jjs
!!
```

That has been a quick look at EASEL as a language and system to build end-user systems. As the *email* application shows, the application design process starts with identifying tasks from user's perspective. These tasks can be quickly prototyped in the EASEL language without lower level computations. In this way, the tasks can be tested out with users to see if they are suitable. In parallel to or with user testing, computational tasks can be implemented and tested. This cycle continues until the system is complete. Figure 5 shows this system construction method.

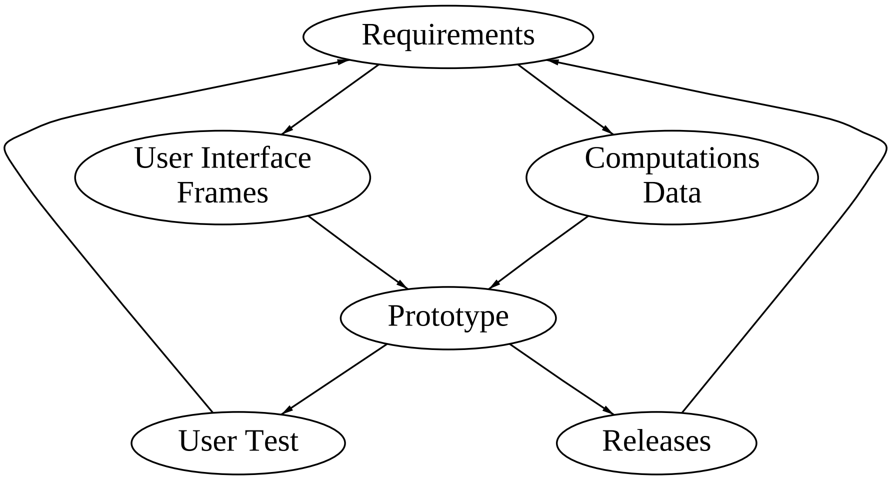


Figure 5: EASEL frame system development cycle

3.2. Building higher abstraction levels with macros

The availability of a language suitable for design programming enables design reuse in the traditional style of building macro templates and code libraries. It is clear how frames parallel traditional library functions and can be reused as such. In many projects where families of applications are built, higher levels of reuse can be achieved by defining macro templates. Such templates are useful because they:

- Reduce programming time of repetitive tasks,
- Standardize the look and feel of the user interface, and

- Standardize the access to lower level computational functions.

For example, consider the task of obtaining a list of logins in frame `users` of the *email* application. The shell communication can be abstracted to the actions of:

- Sending the shell a set of commands,
- Reading responses from the shell, and
- Assigning responses to an appropriate Easel variable.

Below is a macro definition of this task in an extended C preprocessor language [KR78, Fow88]. The macro `KSH_get` takes two arguments `Cmd` and `Var` that define the command to be sent to the shell and the variable to store any responses. The default values for `Cmd` and `Var` are the string `echo ok` and the variable name `ksh_out`.

```
#macdef KSH_get(Cmd="echo ok", Var=ksh_out)
       {process  "/bin/ksh"    > ~Var
          ~!"ShIsDone\n","echo ShIsDone\n"
          ~$
          Cmd
       }p
#endmac
```

The instantiation below of the macro `KSH_get` generates the equivalent code on lines 2–6 of frame `users`:

```
KSH_get(Var=users, Cmd="ypcat passwd | cut -d: -f1 | sort -u | xargs")
```

The communication protocol with the shell as defined in this example is relatively simple. For other tools, it can be much more complex. By using macro templates, frame developers can focus directly on *what* needs to be performed without having to worry about *how* it is performed, which helps avoid inadvertent errors.

The same technique can also be applied to the menu task in the `select` frame. Below is the macro template to do that. Here, macro parameters are used to describe the main components of a menu block. In this example, the menu window is constrained to always appear with the top left corner at location `x=5` and `y=0` and dimensions `xlen=50` and `ylen=10`. This is somewhat contrived (by default, windows are automatically sized and placed by a best-fit rule) but it serves to show how interface constraints can be standardized using macro templates.

```
#macdef Menu_list( Title=Selections, List=list, Sel=sel, Ans=ans)
     {menu  :(~Sel == ~null)  ~Sel
         .window( x=5, xlen=50, y=0, ylen=10 )
         Title: ~Ans
         {option  ~List  " \t\n"
             ~Ans = ~Ans * ~Sel * " "
         }o
     }m
#endmac
```

For completeness, the following instantiation of `Menu_list` generates the same code for the menu block on lines 2–8 of frame `select`:

```
Menu_list(Title=Mail To, Sel=pers, Ans=to)
```

4. Experiences with building an application family

Within AT&T, several projects have successfully used EASEL to build sophisticated network management applications. In particular, a project which we shall called Project D made extensive use of EASEL to generate families of end-user applications for a variety of end-users, including:

- Switch engineers – to monitor, forecast, plan, and equip switches to handle traffic loads efficiently,
- Operations support staff – to monitor and evaluate operator services throughout a network, and
- Service planners – to conduct traffic analyses of specialized services and provide reports.

Much of the raw data for these systems is generated automatically by the switches themselves in the telephone network. Portions of this data are periodically downloaded onto UNIX servers and stored in relational databases, for use by applications. These applications are often written in C with embedded SQL calls. The C code can be complex, sophisticated, and not easy to use by non-experts. But to better serve telephone customers, it is becoming more and more important to make such capabilities readily available to end-users.

When Project D first started, it recognized the needs to reuse existing C code and to provide friendly user interfaces. They found lots of literature on user interfaces, e.g., [Sch87], and a variety of user interface packages inspired by the early work at Xerox PARC [Gol84], including Apple Macintoshes and PCs with Microsoft Windows. But what was needed was to wrap a friendly user interface around large existing C applications running on UNIX systems. EASEL was chosen for this task.

In building applications, Project D's architects recognized that there were many tasks that are repeated again and again, such as:

- Obtaining information from the database,
- Formatting data for display to end-users,
- Collecting end-user responses via menus and forms,
- Providing hypertext help,
- Storing new information into the database, and
- Time-stamping and posting messages to end-users.

This led Project D to layer their code as follows:

- A library of high-level objects,
- EASEL frame code,
- Application-specific libraries, including database access utilities (C and SQL), and
- A relational database.

The high level objects currently number about 70 and are implemented as macros that expand into EASEL code (using `awk` and `cpp`). Code generated by the macros reference a library of about 90 utilities, written both in EASEL and C, many of which interface with the database. As new reusable objects or object attributes are found and defined, EASEL code and database access utilities can be expanded as needed.

To date, Project D has built nearly a dozen different end-user systems, each consisting of hundreds of EASEL frames. During the initial phase of the project, many new objects are discovered, implemented in macros and, as necessary, supported by new low-level C functions and/or database transactions. Application architects have said that the capability to write new macros and new code at any level avoids hitting "brick walls" and makes this approach based on EASEL even more powerful than a typical 4GL. Recent applications have been built entirely out of predefined macros. The expansion of common macros can be "personalized" to meet particular requirements of particular customer sets.

A typical Project D source code file written with macros averages about 100 to 200 lines long, which after macro expansion averages about 1,000 lines of EASEL source code. Thus, use of the macros reduces code size roughly by a factor of 5 if compared to straight EASEL code. It is not as easy to get a similar comparison between EASEL code and equivalent code in C but it is not hard to conclude that the code size reduction would be much larger than that. Project D developers have estimated that programming in EASEL alone (without macros) offers a 10 to 1 advantage in terms of implementation time over writing the same end-user system in C.

This high level of software reuse has allowed Project D to employ small teams to interact with customer focus groups, design, test, and rapidly refine prototypes using customer feedback to flesh out features for production releases. The end results for Project D are customers who feel that they are getting what they really want. Finally, from an organizational point of view, an important aspect of having high level macro objects is that they are easily learned by developers new to the project. New Project D developers typically can begin to build screen objects within a few days of training.

5. Evolution and reuse experience

EASEL has grown and matured significantly over the years, thanks to two complementary forces. One has been feedback from designers and developers using EASEL to build end-user systems. Many discovered leading-edge ways to use EASEL, pushing at its frontiers and suggesting new features. The other has been continued involvement and interaction with other software engineering researchers. Discussions frequently led to the recognition of common problems, some of which eventually lent themselves to common solutions to the benefit of more than one project.

An example of users' feedback is the recent addition of an interactive frame builder `efb`. For many years, EASEL provided a language and environment to program and execute end-user systems. We have discussed the benefits of having a language suitable for implementing the high level design of such systems. However, an aspect of user interface design that cannot be easily addressed by a textual language is that of experimenting with different screen layouts. The frame builder `efb` addresses this needs by allowing users to write EASEL code and lay out the screen interactively. It is interesting to note that `efb` itself is implemented in the EASEL language as a frame system consisting of about 150 frames and 3 coprocesses written in C.

EASEL is based on a number of standard libraries; the original list included: *curses*, for screen manipulations, *malloc* for memory allocation, *regex* for regular expression matching [KP84], and *stdio*, the standard input/output library. In the course of EASEL's evolution, we found that some of these libraries needed improvement and that new libraries were needed.

The earliest version of EASEL, around early 1982, was based on the original *curses* library on the BSD4.1 UNIX System. Aside from a number of bugs, this version of *curses* also suffered from lack of features (such as hardware scrolling) and poor performance. After much consideration, *curses* was rewritten to improve code quality and efficiency. The new library, *screen* [FKV94], remains upwards compatible with *curses* and includes new features such as screen editing, menu display, mouse support. It is also fully internationalized and and supports multiple international multi-byte character sets. Because of *screen*, EASEL is perhaps the only current application construction system that enables applications that run transparently in multiple countries using different character code sets across the Orient and Europe.

EASEL uses the *malloc* package for dynamic memory allocations. Early in its development, it was discovered that standard *malloc* implementations both on System V and BSD UNIX systems had severe deficiencies, either in terms of space fragmentation or time performance or both. This prompted a study by Korn and Vo[KV85] to compare all available *malloc* implementations in 1985. At that time, a new *malloc* package based on the best-fit strategy was also implemented. The study found that this package provides the best trade-off in both time and space. This version of *malloc* is now part of the standard System V UNIX distribution. More recently, we found that certain large EASEL applications may have hundreds to thousands of users running concurrently on the same server. This indicates that there is much to be gained by using shared memory for storing frames on line. Coupled with certain other needs, this prompted a generalization of *malloc* to *vmalloc* [Vo94a, Vo94b], a new library that introduces the idea of allocation from regions each of which may employ a different allocation strategy and a different means to obtain memory.

When event handling was added to EASEL, it was discovered that signals could cause the screen to be only partially updated. The bug was tracked down to the *stdio* library which drops data if a `write` system call is interrupted by a signal. This and other shortcomings of *stdio* led to the writing of *sfio*, a faster, safer I/O library [KV91].

Along the way, it was found that several pieces of software in the department, including EASEL, would benefit from a library for on line dictionary management – for example, to access variables in a symbol table. A flexible dictionary library, *libdict* was written to handle both ordered and unordered objects using binary trees or hash tables [NV93].

The need to distribute low-level libraries that depend on system calls and other environmental parameters for a wide range of hardware and software platforms led to the development of an installation tool to automate the process. The tool, called IFFE [FKSV94], probes and then configures the software automatically without human intervention.

5.1. Other approaches

The first version of EASEL called IFS [Vo90] was first developed in the early part of 1982. At that time, there were few alternative languages for building end-user applications. Within AT&T, the best known tool for form-based applications was FE, a form-entry system [Pri85]. As FE focused on forms, it could not be used for more general applications requiring menus or windows. More recently, there are a number of commercial character-based packages with comparable functionality to EASEL. Most notable are the FMLI package distributed with UNIX System V and the JAM package [Jya94]. FMLI is based on the *curses* library and uses a syntax similar to the shell language enhanced with constructs to write forms and menus. The heavy reliance on separate scripts for forms and menus FMLI cumbersome to use. JAM runs on both PC and UNIX systems and is based on proprietary software for screen handling. JAM relies more on an interactive screen builder to prototype screens than on a language to write applications. Though this makes it easy to build single applications with few screens, it can become cumbersome when families of applications must be built along the line of Project D discussed in Section 4. In fact, Project D developers investigated both JAM and EASEL and decided to use EASEL because its open-ended architecture makes it easy to add new functionality at any number of architectural levels, e.g., high-level macros, EASEL's code, processes, or C routines. On the bit-mapped graphic side, the TCL/TK [Ous94] language and toolkit is compatible to EASEL in the basic approach to tool and application design. In fact, it would be interesting to rewrite the display library of EASEL based on TCL/TK or a similar toolkit. An advantage of doing this is that EASEL-based applications could run transparently on character terminals and graphical workstations.

6. Conclusion

We have described EASEL a language and system for building end user systems. From an application builder's point of view, the success of EASEL derives directly from its philosophy of separating *Design programming* from *Computational programming* by providing a programming language suitable for design implementation. This enables a style of system development that focuses on partitioning tasks as seen from users' perspective and increases the chance of matching users' expectations. Having a high level language for design programming brings in traditional reuse techniques such as code libraries and templates. In addition, by defining precisely a small number of standard interfaces to external code, EASEL encourages the construction of reusable code. We described experiences with a project where a family of end-user applications were built based on these ideas. In this case, the application builders themselves claimed at least a factor of five reduction in code size and corresponding productivity improvement.

We have also touched on our own experiences of software reuse in the construction and evolution of EASEL. The total size for EASEL and associated libraries stands at a little over

60,000 lines of C code. For many years, this body of code was maintained and enhanced by essentially a single person (Vo). This is not normally feasible but for the high level of reusability in the internal code. As the software evolves, we gradually abstract pieces of it into reusable libraries. The construction of such libraries have sometimes led to new and interesting theoretical problems. For example, early in the rewrite of the *curses* library, it is recognized that screen scrolling is best modeled by a string matching problem in which matches have weights. This led to the development of a new heaviest common subsequence algorithm[JV92]. Recently, new algorithms and heuristics for memory allocation were developed in building the *vmalloc* library. Thus, reuse permeates our way of building software and drives the interplay between theory and practice.

7. References

[AKW88] Al Aho, Brian Kernighan, and Peter Weinberger. *The AWK Programming Language.* Addison Wesley, 1988.

[Arn84] K. C. R. C. Arnold. *Screen Updating and Cursor Movement Optimization: A Library Package, 4.2 BSD UNIX Programmer's Manual Supplementary Documents.* University of California, Berkeley, July 1984.

[BK89] Morris Bolsky and David G. Korn. *The KornShell Command and Programming Language.* Prentice-Hall Inc., 1989.

[Bou78] S. R. Bourne. The Unix Shell. *AT&T Bell Laboratories Technical Journal*, 57(6):1971–1990, July 1978.

[FKSV94] Glenn S. Fowler, David G. Korn, J. J. Snyder, and Kiem-Phong Vo. Feature-Based Portability. In *VHLL Usenix Symposium on Very High Level Languages*, October 1994.

[FKV94] Glenn S. Fowler, David G. Korn, and Kiem-Phong Vo. Principles for Writing Reusable Libraries. 1994. Available from authors.

[Fow88] Glenn S. Fowler. cpp - The C language preprocessor, 1988. UNIX Man Page.

[Gol84] A. Goldberg. *Smalltalk-80, The Interactive Programming Environment.* Addison-Wesley, 1984.

[JV92] Guy Jacobson and Kiem-Phong Vo. Heaviest increasing/common subsequence problems. In *Combinatorial Pattern Matching: Proceedings of the Third Annual Symposium*, volume 644 of *Lecture Notes in Computer Science*, pages 52—65, 1992.

[Jya94] Jyacc. Jam application development guide, 1994.

[KP84] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment.* Prentice Hall, 1984.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Lanugage.* Prentice Hall Software Press, 1978.

[KV85] David G. Korn and Kiem-Phong Vo. In Search of a Better Malloc. In *USENIX 1985 Conference Proceedings*, pages 489–506, 1985.

[KV91] David G. Korn and Kiem-Phong Vo. SFIO: Safe/Fast String/File IO. In *Proceedings of Summer USENIX Conference*, pages 235–256. USENIX, 1991.

[NV93] Stephen C. North and Kiem-Phong Vo. Dictionary and Graph Libraries. In *Proceeding of Winter USENIX Conference*, pages 1–11. USENIX, 1993.

[Nye90] Adrian Nye. *Xlib Programming Manual.* O'Reilly & Associates, Inc., 1990.

[Ous94] John K. Ousterhout. *Tcl and the Tk toolkit.* Addison-Wesley, 1994.

[Pri85] Reuben M. Pritchard. FE - A Multi-Interface Form System. *AT&T Technical Journal*, 64(9):2009–2223, November 1985.

[Sch87] Bruce Schneiderman. *Designing the User Interface.* Addison Wesley, 1987.

[Vo85] Kiem-Phong Vo. screen(3X) - more <curses>: the <screen> library, 1985. UNIX Man Page.

[Vo90] Kiem-Phong Vo. IFS: A Tool to Build Application Systems. *IEEE Software*, 7(4):29–36, July 1990.

[Vo94a] Kiem-Phong Vo. Enhancing library usability with disciplines and methods. 1994. Available from the author.

[Vo94b] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator. 1994. Available from the author.

Biography

Glenn Fowler is a Distinguished Member of Technical Staff in the Software Engineering Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey. He is currently involved with research on configuration management and software portability, and is the author of NMAKE, a configurable ANSI C preprocessor library, and the *coshell* network execution service. Glenn has been with Bell Labs since 1979 and has a B.S.E.E., M.S.E.E., and a Ph.D. in Electrical Engineering, all from Virginia Tech, Blacksburg Virginia.

John J. Snyder is a Member of Technical Staff in the Software Engineering Research Department at AT&T Bell Laboratories, where he has done UNIX systems administration and now works mostly on software for end-user systems. He received a Ph.D. in Econometrics from the University of Colorado in 1979. At that time he worked with FORTRAN on a Cray-1 and UNIX on a DEC PDP 11/70 at the National Center for Atmospheric Research in Boulder. After consulting in Mexico City for a couple of years, he joined AT&T in 1983.

Kiem-Phong Vo is a Distinguished Member of Technical Staff in the Software Engineering Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey. His research interests include aspects of graph theory and discrete algorithms and their applications in reusable and portable software tools. Aside from obscure theoretical works, Phong has worked on a number of popular software tools including the curses and malloc libraries in

UNIX System V, sfio, a safe/fast buffered I/O library and DAG, a program to draw directed graphs. Phong joined Bell Labs in 1981 after receiving a Ph.D. in Mathematics from the University of California at San Diego. He received a Bell Labs Fellow award in 1991.