Converting Command-Line Applications into Binary Components

Peter M. Maurer

Department of Computer Science

Baylor University

One Bear Place #97356

Waco, TX 76798-7356

Peter_Maurer@Baylor.edu

Summary

A binary component is a separately compiled program that can be used as a part of a larger program. Binary components generally conform to an accepted technology such as JavaBeans or ActiveX, and generally support a rich program interface containing properties methods and events. Binary components are generally used in a graphical user interface (GUI) environment. There are a number of benefits to be realized by converting command-line software into binary components. The most important of these is that GUI environments are more popular and more familiar to most people than command-line environments. Using binary components can greatly simplify a GUI implementation, to the point where it is only slightly more complicated than a typical command-line implementation. However there are benefits that go beyond mere convenience. Binary components have much richer interfaces than command-line programs. Binary components are service-oriented rather than task-oriented. A task-oriented program has a main routine that is devoted to accomplishing a single task. A service-oriented component has no main routine or main function, but instead provides a variety of services to its clients. Binary components can be easily integrated with one another, which permits a design where each major feature of an application is implemented in a different component. Such a design encourages software reuse at the component level and facilitates low-impact feature upgrades. We first delineate a design-pattern-based methodology for converting command-line programs into components. We then illustrate these principles using two projects, a simulation system for digital circuits, and a data generation system for software and hardware testing.

Key Words: Component-Level Design, Object Oriented Design, Command Line Software.

Introduction

Over the last few years, we have converted a number of command-line applications for UNIX and DOS into binary software components, primarily ActiveX controls. The motivation for the conversion was to make several command-line packages available in the Windows environment. During the course of these conversions we developed a reasonably precise methodology that can be applied to most such conversions. Our methodology applies only to *filters*, but surprisingly enough, this includes the vast majority of all command-line programs. This is primarily due to the rich programmer's interface that is available with binary software components.

In the command-line environment a filter is a program that reads text from its standard input and produces text on its standard output. In the binary software component environment, a filter is any program that accepts input data and transforms it in some way. The input and output data can be simple data types, memory-based objects, files, or collections of such items. Based on this definition we would consider the LINUX *rm* command (remove files) to be a filter, because it accepts a list of unwanted files and a current directory, and produces a transformed directory minus the unwanted files. We would consider the LINUX *df* command (display free space) to be a filter, because it transforms the file */etc/mtab* into human readable form. We would consider the LINUX *ed* command (command-line editor) to be a filter because it transforms a sequence of user commands into a set of human-readable responses. We exclude programs with visual interfaces such as *vi* (visual text editor), daemons and servers such as the *Apache Httpd* server, and commands like *poweroff* that alter the system state without producing any output.

Despite the more general definition, object-based filters can be strung together into larger applications in much the same way as command-line filters. Of course, in component-level programs, the intermediary between components is not shell coding, but glue logic in a language like Java, DELPHI or Visual Basic. The sophistication of these languages permits the interfaces between components to be more complex than those between command-line programs.

Before proceeding it is necessary to define what we mean by the term “component.” In this paper we use the term exclusively to denote binary objects such as ActiveX controls, JavaBeans, COM/DCOM components, VBX components, DELPHI components, and others like them. We will often use the term “binary component” to emphasize the fact that we are focusing on independently compiled binary objects. Although binary components are essentially independent programs, they cannot be loaded and run like an ordinary program. To be useful they must be incorporated into a larger application. Communication between components is accomplished through the use of “glue logic” which is often written in a different language than the component itself. For example, many ActiveX controls are written in C++, but are combined together using Visual Basic. (On the other hand, JavaBeans are written in Java and glued together using Java.)

The component interface must include properties, methods, and events. In some respects a component resembles an object with the properties taking the place of data items and the methods taking the place of functions. However a component is not an object. One important difference is that accesses to properties are monitored. Any time an external entity reads or writes a property the component can observe the access and

perform complex operations in response. As with objects, a method is a function contained within the component that can be called by some external entity. An event, which has no counterpart in objects, is a function implemented by an external entity that can be called from inside the component.

In our conversion methodology, we insist that the interface of a component include properties, methods, and events, but beyond this, we make no recommendation as to which technology is most appropriate. Virtually all existing component technologies are sufficiently powerful to implement the software described in this paper, and all (at least in our opinion) leave something to be desired. The reader is encouraged to choose whichever technology is most appropriate to his or her project.

The Conversion Methodology

The conversion methodology is based on an extensive component-level design methodology that uses the concept of *component categories*. These categories, each of which has its own design methodology, can be used as a guideline for subdividing an application and as a tool for creating the components themselves. The two categories that are most important in command-line conversions are the *filter* and the *serializer*. A short description of each category is given in **Error! Reference source not found.** See Reference 1 for more details. A filter is a component that accepts input data from one or more sources and produces one or more items of output data. A serializer is a specialized filter that is used to transform internal objects into files and vice versa.

Category	Description
Model	A component wrapper for an object, usually a real-world object with a visual interface to show changes in state.
Editor	An interactive editor with an extensive user interface for creating and editing textual or graphical objects.
Background Editor	An editor without a user interface, but with an extensive interface for creating and editing objects programmatically.
Display	Permits a visual display of an object without the capability for changing the object. A PDF viewer is an example.
Accessor	Provides access to an object or simplifies access to an object. An example is a component that enumerates the HTML tags in a page.
Cache	Storage for data. Examples include Open-File-List managers and database access components.
Filter	A background component that reads data, transforms it, and outputs it. Most command-line programs fit in this category.
Serializer	A specialized filter that converts internal objects into files and vice-versa. Can be used to provide access to many different types of files.
UI Widget	Buttons, scroll-bars, tool bars, drop-down lists, and other user interface components.
Decoration	Labels and other visible objects that do not respond to manipulation.
Function Library	Very rare, because standard function libraries generally work better. An example is the standard dialog interface component.
Service Wrapper	Provides access or simplifies access to an operating system service like dialing a modem, or setting the state of the caps-lock key.
Container	A component that can contain other components. Usually used to group user interface items.
Other	Anything that doesn't fit in any of the above categories. Such components are extremely rare.

Figure I. List of Component Categories.

For command-line conversions, the design of a filter component is essentially the design of the input and output mechanisms. Although these mechanisms could be arbitrarily complex, only a few mechanisms are used in practice and we have developed a number of elementary patterns to handle them. These patterns are on the scale of the *iterator* pattern from Reference 2. This pattern includes four functions *First()*, *Next()*, *IsDone()*, and *CurrentItem()*. These functions are typically embedded in an iterator object, but can be embedded in larger objects as well. The *First* function initiates a list

traversal, the *Next* function proceeds to the next object and the *IsDone* function tests for the end of the list. The *CurrentItem* function returns the item currently selected by the *First* and *Next* functions. We use the iterator pattern and a number of other patterns to construct our filter interfaces. The body of the filter is taken from the command-line application.

The input-usage pattern of the command-line application can be arbitrarily complex, but it generally falls into two broad categories which we call *demand-for-data* and *respond-to-data*. The demand-for-data pattern, which is illustrated in Figure II can be found in the LINUX *cat* and *cp* commands. These applications use the *read* system call to demand data from files when it is needed. The LINUX *rm* command, on the other hand, exhibits the respond-to-data pattern. A list of unwanted files is passed to a subroutine. This subroutine responds to the data in the list by deleting the files contained in it. The LINUX *df* command contains both patterns. The application first constructs a list of mounted devices using the *setmntent* and *getmntent* system calls, and then passes the list of mounted file systems to a processing routine for translation and display.

```
myfile = fopen(fileName,"r");
InputChar = getc(myfile);
while (!eof(myfile))
{
    // process data
    InputChar = getc(myfile);
}
fclose(myfile);
```

Figure II. The Demand-for-Data Pattern.

The respond-to-data pattern is, by far, the easiest to deal with. The data processing routines can be copied intact from the command-line program, and it is merely necessary

to guarantee that the appropriate data structures are supplied to them. Unfortunately, the demand-for-data pattern is far more common. Generally this takes the form of file input, so it is necessary to go through the code and replace any file-handling statements with generic data handling functions. The purpose of these functions is to convert the data input pattern to a respond-to-data pattern. This is necessary because components are essentially passive entities that respond to method calls and changes in their properties. If a component is to read a file and produce a report (for example), the activity must be triggered externally. The component will respond to the receipt of a file name or to a request for a report. There is no main routine that is responsible for stimulating all program activity.

We use four different patterns for component input. In the *one-to-one* pattern a single data item is passed through the component, directly to an internal processing routine. In the *one-to-many* pattern the data is parsed by the component and is passed piecemeal to an internal processing routine. These are the two most common patterns. The one-to-one pattern is typically used with the respond-to-data internal pattern, while the one-to-many pattern is typically used with the demand-for-data pattern. The two other patterns are the *many-to-many* and the *many-to-one*. In the many-to-many pattern, a single input is supplied to the component in piecemeal fashion with each piece being broken down into smaller units. This pattern is generally used with the demand-for-data internal pattern. In the many-to-one pattern, a single item is passed to the component in piecemeal fashion, but the data items are accumulated into a single data-structure before being passed to the internal routines. This pattern is rarely used.

Each pattern is implemented using a specific set of properties, methods, and events, but before giving the details of the implementations, it is necessary to discuss output patterns. The two most important patterns are the *retained-output* pattern and the *transitory-output* pattern. In the retained-output pattern, the component creates an output object, but retains the object until it is requested. The component is capable of responding to several requests for the same output object, either by copying the object or by providing a pointer to it. In the transitory-output pattern, output data is made available once and not retained by the component. This pattern is further broken down into the *single-item* and *multiple-item* patterns. The multiple-item pattern is seldom used in isolation. It is generally combined with one of the other output patterns.

The one-to-many and many-to-many input patterns require the most work, because they are used with the demand-for-data internal pattern. In most cases, the original algorithm was designed to read characters, lines, or other data from a file. The original code is converted using an abstract object that takes the place of the input file. The object provides virtual functions that replace the functions used in the original code. A separate derived object is provided for each different input source. The most common input sources are character strings and files, but the approach supports a wide variety of sources such as linked-lists, remote objects, or direct keyboard input.

Each input and output pattern requires a different set of interface items, with the exception of the one-to-one and one-to-many patterns which appear identical to the outside user. (The difference between the two is the abstract object that is used inside the component to provide input to the processing routines.) When the one-to-one or one-to-many pattern is combined with the transitory single-item output pattern, a single method

is used. The method arguments are used for input and the return value is used for output. Errors are indicated by a special return value (usually zero or NULL), and error diagnostics are provided through an *ErrorCode* property. A different method is used for each different type of input, so we might have *ConvertString* and *ConvertFile* methods to handle both files and strings.

When the one-to-one and one-to-many patterns are combined with the retained output pattern, properties are used for both input and output. (In some cases we replace the input properties with single-argument methods having *void* return values.) Assigning a value to the input property will trigger the execution of one or more internal routines, and will cause the output value to be made available through the output property. In some cases, the output property can be used as the execution trigger to provide on-demand execution of the internal routines. This technique is most appropriate when the output value optional to the over-all computation and when multiple copies of the output are not needed. In the unusual case that the component is being used as a file-filter, a different pattern is used. (File input and output is generally handled by a separate component, so file-filters are rare.) Two properties are used for the input and output file names, and a method is used to trigger execution of the internal routines. Thus we might have *InputFile* and *OutputFile* properties, along with an *Execute* method.

The many-to-many pattern is implemented in much the same way as the many-to-one pattern, but an additional event is necessary. When the current input-item is exhausted, this event is fired to request more data. This data will be supplied through a special property that is used only for supplying additional input. If all input items are exhausted, this condition will be passed back to the component through a side-effect on a event

argument. To illustrate, consider the code of Figure III. An event, *NeedMoreData*, with one argument, *Done*, has been defined. The default value of *Done* is *False*, but when the input is exhausted, the event handler will set *Done* to *True*.

```
Private Sub MyComp1_NeedMoreData(Done as Boolean)
    If DataIsExhausted Then
        Done = True
    Else
        MyComp1.MoreData = SomeString
    End If
End Sub
```

Figure III. An Event Handler for Many-to-Many Input.

The many-to-one pattern is implemented using three methods, *AddItem*, *Clear*, and *Execute*. An example of such a pattern is a Sort component where the list items are supplied one at a time. The *Clear* method is used to erase any accumulated data and start with a new list, the *AddItem* method is used to add items to the list, and the *Execute* method is used to perform the sort operation. This pattern is generally combined with the retained-output pattern. The result of the operation will be available through an output property.

The transitory multiple-output pattern is generally combined with some other output pattern. It is used to provide output data that is both transitory and unpredictable. For example, we use this output pattern to transmit error messages for compiler components. There may be many error messages. Their number and occurrence is unpredictable, and they are a side-effect of another computation whose normal output is something else. This pattern is implemented using an event. The output data is passed to the event handler in the event arguments. In the compiler example, there may be separate arguments for message text, line number, and file name.

In addition to input and output specifications, command-line programs also have flags and other configuration parameters. These parameters translate into properties, one for each independent parameter. Our methodology for creating configuration parameters is quite similar to that of Reference 3. If two flags are mutually exclusive, a single property is used for both. If two parameters are independent, then they are implemented as two separate properties. If a parameter is an on/off type option, such as the *-f* parameter of the LINUX *rm* command, then we represent it as a Boolean property. If it supplies data like the *-f* option of the *tar* command, then we represent it as a string with the null string representing an unspecified parameter. If it supplies one of a collection of mutually exclusive options like the *-O* option of the *gcc* command, then it is represented as an enumerated type with meaningful names for each of the options.

Most filters have some configuration parameters but there are some that cannot perform any function without them. For example we have several circuit-simulators that must be given a circuit description through a configuration parameter. Without this configuration data, the simulator has no default behavior. We call such filters *configurable* filters or *hot* filters. Such filters will have one “essential” configuration parameter, implemented as a property. This parameter is usually a pointer to a complex object.

Two Specific Examples

The methodology described in the previous section was used to convert two major command-line applications to components. The first was a popular data generator called the *Data Generation Language* (DGL) which we have made available on the World Wide Web for many years. The second was a simulation system called the *Functional*

Hardware Design Language (FHDL), which allows for the specification and simulation of circuits at the logic level. The input to both of these systems is a set of specifications coded a high-level language. Parsers are used to convert the high-level languages into data structures, which are then manipulated by other routines.

There were compelling reasons for choosing components as the target technology for our conversion. In the command-line environment it is common practice to use a separate text editor such as *vi* or *emacs* to create high-level language specifications, and then use other programs such as *cc* or *gcc* for the actual compilation process. In the PC graphical environment, it is generally expected that a compiler will provide an integrated text editor. Component-level design is a reasonably easy way to provide such a facility. There are many third-party text editors that can be integrated into a project with only a moderate amount of effort. It is a relatively simple matter to integrate the converted software with these and other third-party components, giving a final product that has a similar look-and-feel to a commercial compiler.

Although our conversion methodology can be used with virtually any software, there are concerns that go beyond the methodology when converting a major application. Command-line programs are generally invoked as separate processes, which simplifies memory management. Lost memory fragments are recovered upon process termination and little attention needs to be given to memory leaks. Components, on the other hand, can run for an extended period of time and their internal functions can be invoked repeatedly during this period. Under these conditions it is necessary to audit the command-line code carefully to locate and correct any memory management problems.

In addition, command line applications may contain legacy code that has existed for decades and does not conform to modern coding practices. In particular, the code may not be object oriented, and may depend heavily on global variables for inter-subroutine communication. Global variables cannot be used in a component environment, and must be eliminated. The best way to do this is to create an object that represents the global environment of the application, and make all functions and global variables members of the object. This process akin to renaming all of the variables of a program and can be quite tedious.

The initial plan for the two conversions was the same. A compiler component was created to parse text into internal data structures, and a second component was created to produce the actual output. For both projects, we took the opportunity to produce a new object-oriented design of the original software before performing the conversion.

The DGL Project

The DGL object design consists of two main classes *CDGLParse* and *CGrammar*. The *CDGLParse* objects create *CGrammar* objects from DGL text specifications, while the *CGrammar* objects provide the data-generation functionality. DGL specifications are modeled after the productions of probabilistic context free grammars[4]. Each input file is assumed to be a single grammar containing several productions and a few other specifications, such as start-symbol name. The *CGrammar* object is the compiled form of a DGL grammar. It contains an array of *CProduction* objects which implement the productions. Unlike ordinary context free grammars, DGL has many different types of productions. The production type is used to specify the manner in which data is

generated, randomly, serially, with replacement, without replacement, and so forth. See References 5 and 6 for details.

In the original implementation of DGL, C code was generated for each production and then compiled to produce a data generator. This was a round-about way to implement polymorphism in a non-object oriented language, and also allowed user-written C code to be integrated with standard productions. On the PC one cannot assume the existence of a C compiler, so this approach had to be abandoned. The generated code was incorporated into the functions of the *CGrammar* and *CProduction* objects. The *CProduction* class was made the base class for a polymorphic type. Derived types were used to implement the various different production types.

Initially the design contained two components, a parser component and a data generator component which were wrappers for the *CDGLParse* and *CGrammar* objects. The structure of the system is shown in Figure IV.

Figure IV. Initial Data Generator Design.

The data generator used the retained output pattern and the one-to-many input pattern with two different sources, files and strings. Two single-argument methods, *parsefile* and *parsestring*, were used to implement the input interface. A property, *GrammarHandle*, was used to implement the output. This property contained a pointer to the most recently

created *CGrammar* object. The transitory multiple-output pattern was used for error messages.

The data generator component was designed as a configurable filter with one configuration parameter *GrammarHandle*. The one-to-one input pattern was combined with the transitory output pattern to create two data-generation methods, *InterpretString* for strings and *Select* for integers. The *Select* method mimics the command-line data generators where the integer indicates the number of selections that are to be made from the grammar start symbol. Unlike the command-line generators, which produced their output on standard out, the component concatenates the selections and places them into the return value of the *Select* method. The *InterpretString* method passes a string to the component, which replaces all non-terminal symbols and returns the result.

In the command line version, user input was limited to a single integer specifying the number of data items to be generated. Data generation always began with the start symbol of the grammar. Great care was needed in preparing test specifications because every single character that was generated needed to be specified in some production. Specifications could not be altered at run time, nor could the behavior of the data generator be changed dynamically.

The richness of the component interface permitted several enhancements to be made. In the component implementation, any specification can be treated as the starting point for data generation. Specifications can be placed in external files, which can be altered at any time. The user can create template libraries and update them without changing the DGL specifications. Templates can be generated dynamically, perhaps in response to

feedback from automatic test software. New productions can be created dynamically, and entire grammars can be constructed on the fly.

Two additional components were created to facilitate these new features. The first was a background editor for creating specifications dynamically, and the second was a serializer for saving dynamic specifications in text format. The final design is shown in Figure V.

Figure V. Final Data Generator Design.

When implementing components it is necessary to consider how they will be used and the complexity of the glue logic that will be used to bind them together. Although implementing each logical component as a separate physical component gives maximum flexibility, it also increases the complexity of the glue logic. Multiple components are more difficult for the developer because it is necessary to keep track of the names of the individual components and remember which features are implemented in which component. Because we wanted our DGL components to be used by as many people as possible we opted for the convenience of a single physical component. The Common DGL component (COMDGL) provides four logical interfaces, one for each of the logical

components contained within it. These logical interfaces are combined into a single physical interface. Communication between the logical components is performed automatically within the COMDGL component. This sort of implementation is typical of third-party commercial components.

The FHDL Project

The aim of the FHDL project was to create several user-level programs for specification and simulation of electronic circuits. This is in contrast to the DGL project, where the product was the component itself. Because of the different point of view, the FHDL project resulted in a large collection of different components which could be expanded to provide new services. This is typical of projects that are application oriented. The services of the application are divided up among components in such a way as to facilitate the future development and enhancement of the application. Since the users of the components work closely with the designers of the components (and are often the same people) there is little advantage to combining individual components as was done in the DGL project. The conversion began with the design of the *CNetList* object, which contains two collections of sub objects, a list of gates (*CGate* objects) and a list of nets or wires (*CNet* objects). This structure existed in the command-line version of FHDL, but was cleaned up for the component-level implementation. (See Reference 7 for details of the command-line version.)

The basic simulator consists of two components, a parser component and a simulation component. The many-to-many input pattern is used with an event, *MoreInput*, used to request more input. There are two input sources strings and files. The single-item transitory output pattern is combined with the many-to-many input pattern. The two

methods *ParseFile* and *ParseString* return pointers to *CNetList* objects. The output object is not retained by the component.

The *CNetList* object from the parser must be passed to the simulator to configure it for simulation operations. Once configured the simulator is a Boolean-vector to Boolean-vector filter. Input vectors are passed to the component using the *SimulateVector* method, while outputs are produced in the *OutputVector* property. Delays within the circuit can cause outputs to change values several times before the final output is produced. These intermediate outputs are reported using the multiple transitory output pattern. When an intermediate result is available, the *IntermediateVector* event is fired to pass the intermediate vector to the glue logic.

As time passed many new components were added to the FHDL system. As this occurred, the components themselves became less important, and the *CNetList* object became more important. We began to view the system as a collection of *CNetList* producers and consumers, as pictured in Figure VI. Parsers for new languages were added, schematic-capture tools were created, various macro languages were added, new simulators were written, and a number of circuit-manipulation tools were created.

Many other components have been created for this system, too many to be described here. These have been used to create a wide variety of applications. While these applications could possibly have been created in the command-line environment, the component-level interface has vastly simplified the process of creating new components and gluing them together into applications. Separation of the system into a collection of independent components has permitted virtually unlimited expansion of the system with minimal impact on existing components.

Figure VI. Simulator Configuration.

Conclusion

Binary component technologies have provided us with an efficient and effective means for creating new applications. To use this technology most effectively, the software designer must be able to design and build new reusable components in addition to using existing components. In our methodology we create many of the components we need by adapting existing, well-proven software. During the development of an application, each feature is separated into a separate component. Depending on the target audience for the software, some logical components may be combined into a single physical component.

In addition to converting command-line applications, our methodology can be used to design new applications (see Reference 1). The projects described here demonstrate the effectiveness of our approach. We believe that our methodology will continue to be an effective tool for converting command-line applications and for developing new component-level software.

References

1. P. Maurer, *Component Level Programming*, Prentice-Hall, Upper Saddle River, New Jersey, 2003.
2. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns*, Addison-Wesley, Reading, Mass, 1995.
3. C. Sorzano, J. Carazo and O. Trelles, "Command-line interfaces can be efficiently brought to graphics: COLIMATE (the COMmand Line MATE)," *Software Practice and Experience*, Vol. 32, No. 9, 25 July, 2002, pp. 873-887.
4. T. L. Booth, R. A. Thompson, "Applying Probability Measures to Abstract Languages," *IEEE Trans. Comput.*, **C-22**, 442-450 (1973).
5. P. M. Maurer, "The Design and Implementation of a Grammar-based Data Generator," *Software Practice and Experience*, Vol. 22, No. 3, March 1992, pp. 223-244.
6. P. M. Maurer, "Generating Test Data with Enhanced Context Free Grammars," *IEEE Software*, **7:4**, 50-56 (July 1990).
7. P. M. Maurer, Z. Wang, C. Morency, A. Tokuta and N. Bhate, "The Florida Hardware Design Language," *Proceedings Southeastcon-90*, pp. 430-434.