Explore, Excogitate, Exploit: Component Mining

Diomidis Spinellis, University of the Aegean

How do we obtain software components? In a panel at TOOLS USA 1994, Eric Aranow nicely outlined the basic question: "Is it nature or nurture?" In other words, are components born—devised from the start as components—or are components made? That is to say, have they evolved from program elements that may have been originally built for other purposes? Although some panel members argued for the nature view, it was clear to everyone that the nurture process of maturing candidate components until they are fully satisfactory can be invaluable.

In this column, Diomidis Spinellis describes a process of component mining. His source of candidate components—his "mine"—is a set of highly respected components from an earlier generation: the famous Unix utilities. He shows how to identify the most promising of these candidates and re-encapsulate them in components that satisfy today's views of component-based development, based on the principles of object technology.

—Bertrand Meyer

The wide adoption of the OO paradigm and recent technology advances like Enterprise JavaBeans and ActiveX controls has generated renewed interest in component-based software engineering.

An increasingly important issue in the development process is the reliability and availability of the source that supplies components.

But an increasingly important issue in the development process is the reliability and availability of the source that supplies these components.

As a relatively new field, component mining is the process of extracting reusable components from an existing component-rich software base. The most effective component-mining technique is one in which the process of mining is clearly defined. This column outlines a method I use for mining components from applications that are typically executed as Unix processes.

Editor: Bertrand Meyer, EiffelSoft, ISE Bldg., 2nd Fl., 270 Storke Rd., Goleta, CA 93117; voice (805) 685-6869; ot-column@eiffel.com

THE UNIX MINING FIELD

Unix developers have created a large collection of applications that provide a single service—such as comparing two files, searching for a pattern, or delivering e-mail—without requiring user interaction. Many of those programs have been implemented using state-of-the-art algorithms, have been stress-tested for decades, and have had their interfaces standardized. Furthermore, many of these programs are freely available in source code form through open-source initiatives like GNU and BSD.

You could argue that these programs have always been used as components, which you basically glue together using a Unix shell. Although this might be true in one sense, current trends call for a component model that is much richer than the one provided by the Unix shell.

Systems using ActiveX or JavaBeans components

- are based on OO languages,
- can provide a variety of efficient component-composition approaches,
- are often integrated with GUI environments, and
- are supported by modern development environments.

In contrast, the Unix shells largely

- lack facilities for programming in the large,
- support only the serial pipe composition model,
- are designed for character-based terminals,
- do not provide compilation support, and
- offer only rudimentary debugging facilities.

A process for packaging existing programs as object components can elevate

the individual reuse of specific algorithms or implementations to an organized component-mining operation.

THE COMPONENT MINING PROCESS

The process of mining components and subsequently using them within an application domain can be divided into the three phases illustrated in Figure 1:

- exploration,
- excogitation, and
- exploitation.

These phases roughly correspond to the selection, specialization, and integration dimensions of typical software reuse methodologies.

During the exploration phase, you elicit component requirements and—on the basis of component abstractions—select components. In this phase, the selected components and the system architecture determine your corresponding interface requirements.

The excogitation phase deals with the encapsulation of the components that have to be mined and the implementation of suitable interfacing glue for connecting components with the rest of the system. The abstract nature of packaged components and interfaces means that many of them can be stored in a repository for future reuse or retrieved from this repository for direct reuse.

Finally, during the exploitation phase, you use the reused and newly encapsulated components and corresponding interfaces to create a functioning system.

The excogitation and exploitation phases are composed of three basic activities:

- component encapsulation, where an existing stand-alone program is converted into a component object;
- component glue implementation, where special-purpose components provide a uniform and reusable interfacing mechanism between the mined components and the rest of the system; and
- component use and composition, where component objects are combined to form new structures and components.

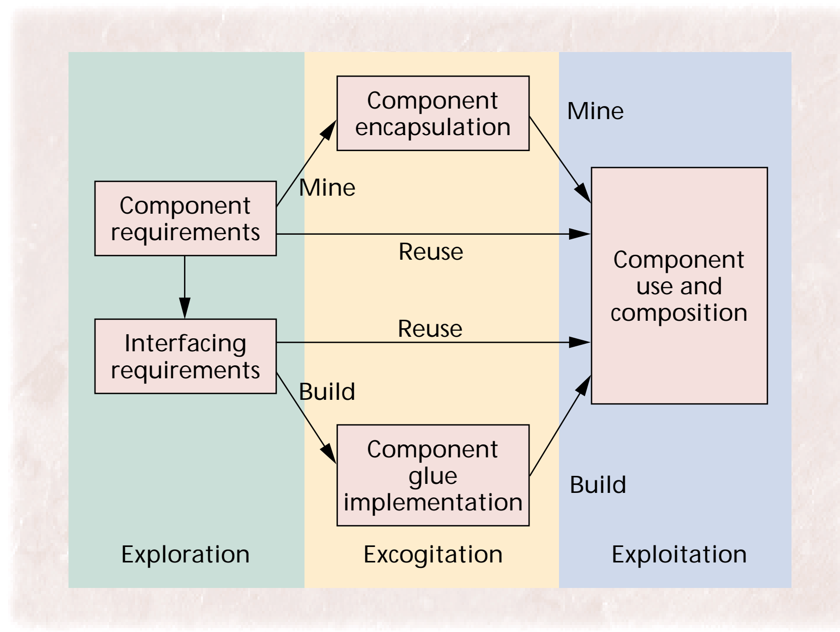

Figure 1. The component mining and exploitation process.

COMPONENT ENCAPSULATION

Component encapsulation creates a component object—usable within an object-based framework—out of a stand-alone, noninteractive program. Encapsulating filter-style programs that transform one data stream into another is a fairly straightforward process. Programs requiring limited user interaction can be encapsulated using a suitable wrapper, while programs that have a graphical front end are—in most cases—poor candidates for encapsulation.

Encapsulated Unix components typically process input data streams and generate output streams based on parameters that modify their behavior. As an example, the diff component (which I implemented as an encapsulated version of the Unix command by the same name) processes two textual input streams and generates a third stream that contains their differences. Among other things, the diff parameters specify what output format to use, what algorithm to use, and how to handle white space.

Encapsulating a stand-alone program makes it directly usable in object-based frameworks. And specifying a standard filter-type component class allows repetitive aspects of the encapsulation to be reused. You can even specify a class to automate the encapsulation, but doing so means you are bound to a generic component interface.

Component encapsulation allows you to experiment with different component implementations that may vary in terms of performance, cost, licensing restrictions, and resource use.

As an example, thread-based implementations conforming to a framework's structuring conventions offer increased efficiency but at a higher implementation cost. A thread-based implementation can advantageously use wrapper libraries to transform existing OS call primitives into interfaces to the encapsulation code.

COMPONENT GLUE IMPLEMENTATION

Apart from singular options controlling an encapsulated component's operation, the bulk of the data is transferred to and from the component through streams. Typical streams are formed from the stand-alone program's standard input and output as well as any other user-specified files. To use a component effectively, these streams need to be connected to existing data sources such as in-memory data structures, files, relational databases, procedures producing dynamic

data, GUI widgets, and other components. These connections are all handled by the glue components.

A glue component class needs to be designed whenever a new type of existing data source or sink needs to be linked to an encapsulated component. While glue components can be used to provide extra functionality for linking components together, they also allow the integration of encapsulated components within an object-based framework.

Suitable glue components can be used to provide efficient interfaces to system data, which can obviate the cumbersome file-based approaches typically used to interface stand-alone programs. The existence of glue components allows the designer to experiment with different data sources and sinks without having to modify the rest of the system structure.

COMPONENT COMPOSITION

Encapsulated components do not operate in a vacuum. They are composed to create more powerful components and are integrated within an object-based system to provide more specialized services.

The composition of encapsulated components with component glue can be used to provide efficient access to offline data, GUIs, and a multitude of other component-based services. A spelling checker, for example, can be easily constructed by composing the translate, sort, unique, and other components while gluing together the editbox and listbox components to provide the GUI.

Many of the problems solved under the Unix programming environment using shell programming constructs and pipelines can be transformed to component composition structures. Of particular relevance are sequences of filter-type components, where each one receives a data stream, performs some operations on it, and forwards it to another filter to perform some other operations. Examples include pipelines of tools that process text, images, sound, and program code.

The components composed are object instances of either active process components—that are connected to existing data sources and sinks—or glue components that provide such sources and sinks. By using component composition, it is possible to implement sophisticated component interaction topologies. It is also possible to package together existing components to create new, reusable ones.

I have used the component mining process to encapsulate a number of Unix tools using Perl's OO features. I used a simple wrapper approach that required only a modest implementation effort. By using the encapsulated components, my colleagues and I have been able to code applications intuitively and naturally.

The component mining process has proven to be addictive. The ease of encapsulation, the limitless possibilities of object structuring, and the flexibility of using a high-level language to interact with the components have opened new ways to leverage existing tools and applications. I am currently experimenting with more efficient encapsulation techniques using threads to construct image processing applications with encapsulated Unix tools.

I would like to see component mining extended to other mining fields, which would be supported by appropriate domain-specific patterns and languages. ❖

***Diomidis Spinellis** is an assistant professor in the Department of Information and Communication Systems at the University of the Aegean. Contact him at dspin@aegean.gr.*

Set Industry Standards

Our members write important IT standards. Our members wrote IEEE 802.3, the standard for Ethernet, the most widely deployed LAN. But technology networks are not the only kinds developed here.

Grow Your Career • Find Out How @

http://computer.org/standard/index.htm

Did You Know?

Right now, over 200 Working Groups are drafting IEEE standards.