Component Mining: A Process and its Pattern Language [*†]

Diomidis Spinellis and Konstantinos Raptis
Department of Information and Communication Systems
University of the Aegean
Greece
email: dspin@aegean.gr

Abstract

An important issue in a component-based software development process is the supply source of mature, reliable, adaptable, and maintainable components. We define as *component mining* the deliberate, organised, and automated process of extracting reusable components from an existing rich software base and present a pattern language used for mining components from programs that are typically executed as non-interactive autonomous processes. We describe the patters in terms of intent, motivation, applicability, structure, participants, consequences, and implementation. Based on the pattern language, we describe the implementation of a set of COM components that encapsulate the Unix filters and an exemplar application that uses them.

Keywords

Component mining, pattern language, Unix tools, reuse.

1 Introduction

The increasing adoption of the object-oriented paradigm in conjunction with recognised shortfalls of "pure" object-oriented development [6, p. 30] and recent technology advances such as Enterprise JavaBeans and ActiveX have generated renewed interest in component-based software engineering [4]. Object-oriented design and implementation allows the composition of systems using pre-packaged software components [15] while technologies such as CORBA, ActiveX, and JavaBeans provide the necessary framework for constructing such systems.

A component can be defined as "a physical and replaceable part of a system that conforms to and provides the realisation of a set of interfaces" [2, p. 20]. More concretely, a software component can be defined as a unit of composition with contractually specified interfaces and explicit context dependencies only; it can be deployed independently and is subject to third-party composition [22]. Components, in common with objects, encapsulate state, allow access to it through separately described interfaces, and support modular design based on separation of concerns. However, components differ from objects in a number of ways: they can be implemented in different languages, they are often packaged in binary containers, they can encapsulate multiple objects, and are typically more robustly packaged than objects [24].

An important issue in a component-based software development process is the supply-source of mature, reliable, adaptable, and maintainable components. We define as *component mining* the deliberate, organised, and automated process of extracting reusable components from an existing component-rich software base. Component mining is a product *and* process reuse activity [14] that relies on the exploration and exploitation of large pre-existing component-rich fields [20]. Effective component mining is supported by a clearly defined, and possibly automated, *process* for identifying and packaging the software components.

The remainder of this paper is structured as follows: in section 2 we describe our mining field which consists of mature, filter-style programs available as open source in many Unix implementations; in section 3 we present a pattern language [1] used for mining components from programs that are typically executed as non-interactive autonomous processes, and in section 4 we provide a case study on how the components were implemented using a partly automated process. Section 5 concludes the paper with a brief evaluation of our approach and our plans for further work. The pipe and filter model and component-based software engineering is not a new idea; see for example [19, 13] (discussing pipe and filter architectures), [15, 6, 22] (discussing component-based development) and the references therein. The main contributions of this paper are the description of

[*] *Information and Software Technology*, 42(9):609–617, June 2000.

[†] This is a machine-readable rendering of a working paper draft that led to a publication. The publication should always be cited in preference to this draft using the reference in the previous footnote. This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

component mining using a pattern language, the proposal to repackage Unix filter-style programs as components, and the presentation of a partly-automated mining process based on a domain-specific language.

2 The Mining Field

The mining field and at the same time our motivation source for defining the patterns we describe consists of the numerous user and system programs available under the Unix operating system implementations. Based on the Unix tool-centred philosophy software developers have created a large collection of programs that provide a single service (e.g. compare two files, search for a pattern, deliver mail) without requiring user interaction. Many of those programs are implemented using state-of-the-art algorithms, have been stress-tested in many diverse applications for decades, and have their interface and operation standardised under efforts such as POSIX[10]. In addition, many of these programs are freely available in source code form through Open Source initiatives such as GNU and BSD. In fact, many of the social processes that have contributed to the success of mathematical theorems as a scientific communication vehicle [8] apply to this class of programs. Many of these programs have been:

- documented, published, and reviewed in source code form,
- discussed, internalised, generalised, and paraphrased, and
- used for solving real problems often in conjunction with other programs.

The above factors are responsible for the creation of a rich base of mature, reliable, standardised, and maintainable component candidates.

One can argue that these programs have always been used as components connected together using one of the Unix shells. Although this statement is in a weak sense true, current technological trends call for a component model a lot richer than the one provided by the Unix shell. Systems using ActiveX or JavaBeans components are based on object-oriented programming languages, can provide a variety of efficient component composition approaches [25], are often integrated with GUI environments, and are supported by modern program development environments. In contrast, the Unix shells 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. Therefore, a *process* for packaging existing programs as object components can elevate the individual reuse of specific algorithms or implementations into an organised component mining operation.

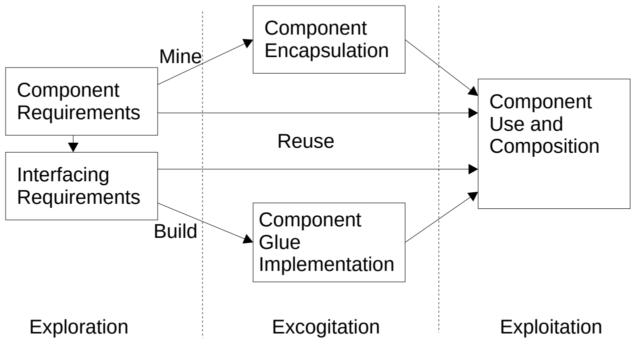


Figure 1: The component mining and exploitation process.

3 The Process and its Pattern Language

Given a set of component requirements and their resulting interfacing requirements, the process of component mining and subsequent use within an application domain is functionally and temporally divided into the three phases illustrated in Figure 1. These phases roughly correspond to the *selection*, *specialisation*, and *integration* dimensions of typical software reuse methodologies [12]. During the *exploration* phase component requirements are elicited, and components are selected based on the existing component abstractions of the mining field. In addition, the selected components and the system architecture determine the corresponding interfacing requirements. The *excogitation* phase deals with the encapsulation of the components that have to be mined, and the implementation of suitable *connectors* for joining the components together and *glue* for interfacing 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 the reused and newly encapsulated components and corresponding interfaces are used to create a functioning system.

The three activities that form the excogitation and exploitation phases concern the design and implementation of concrete artefacts. These activities are:

Component encapsulation An existing standalone program is converted into a component object.

Glue and connector implementation Special-purpose glue components provide a uniform and reusable interfacing mechanism between the mined components and the rest of the system. In addition, connectors may need to be built to interface the mined components with each other [6, p. 454].

Component use and composition Component objects are combined to form new structures and components.

These activities are essentially solutions to problems occurring repeatedly in the context of component mining. By giving each activity a concrete name, describing the problem it addresses and its context, outlining the solution's elements and relationships, and analysing the activity's consequences we are creating a *pattern language* of the component mining process. It is therefore appropriate to describe these activities by means of the three corresponding design patterns. These patterns do not depend on the underlying component framework or the mined programs and can therefore be used to integrate arbitrary tool-type programs to object frameworks such as ActiveX, JavaBeans, and CORBA. In the following paragraphs we describe each pattern by roughly following the format used Gamma et al [9]. Thus, for every pattern we:

- provide the name that will be used to describe it in our component mining process vocabulary and classify it as creational, behavioural, or structural,
- illustrate the design problem that provides our motivation to use the pattern,
- outline the situations where that pattern can be applied,
- provide a graphic representation of the pattern's classes,
- list the classes and objects participating in the pattern,
- describe how the pattern supports its objectives, and
- provide prescriptive guidelines towards the pattern's implementation.

3.1 Component Encapsulation — Creational

Intent

Component encapsulation creates a component object out of a standalone non-interactive program.

Motivation

Encapsulating the multitude of powerful and mature standalone programs as component objects makes them usable within modern object-based frameworks.

Applicability

All programs lacking mandatory user interaction can be encapsulated using this pattern. Particularly elegant is the encapsulation of *filter* style programs which transform a data stream into another. Programs requiring limited character-based user interaction can be encapsulated using a suitable wrapper, while programs with a graphical front-end are in most cases poor candidates for encapsulation.

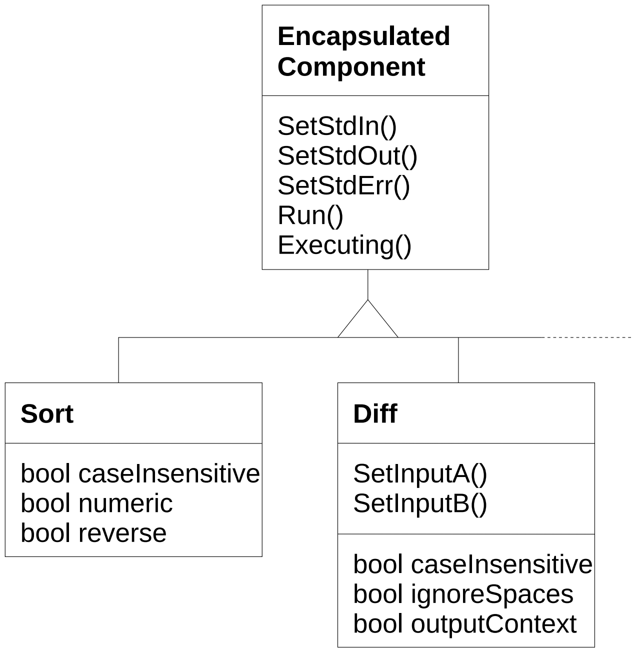


Figure 2: Component encapsulation object diagram.

Structure

Encapsulated components process input data streams generating output streams typically based on parameters that modify their behaviour. As an example, the *diff* component (an encapsulated version of the Unix command with the same name) processes two textual input streams generating as output a third stream containing their differences. Its parameters specify *inter alia* the handling of white space, the output format, and the algorithm to use.

As shown in Figure 2 every encapsulated component provides a set of standard methods. These methods specify the component's interconnection. The *SetStd*()* methods are used to set the component's input and output streams. In addition, most components will offer component-specific instance methods and variables to specify particular program options such as the recipient of a mail message for a mail transfer agent, or the collating sequence order for a sorting component. In order to maximise the component composition flexibility, parameters that are associated with files in the original programs can be changed to methods used to specify streams in the encapsulated component. These can then be arbitrarily and efficiently connected to a variety of data sources and sinks. Once all component parameters are set the component's *Run()* method is called to specify that its operation can commence. Since most components execute asynchronously, they provide the *Executing()* method to allow a program to wait until a component's operation has completed.

Participants

Encapsulated components are typically connected with the rest of the program and other components using *glue component* objects.

Consequences

The encapsulation of a stand-alone program as a component makes it directly usable in modern object-based frameworks. The specification of a standard component class allows repetitive aspects of the encapsulation to be reused and can even be used to automate the encapsulation; at the cost of having a generic component interface. In addition, the generic specification of data sources and sinks as streams allows the composition of encapsulated components using sophisticated topologies that can not be typically implemented using the Unix shells. Furthermore, the component encapsulation allows the system implementor to experiment with different component implementations which may vary in terms of performance, cost, licensing restrictions, and resource usage.

Implementations

A large number of implementation possibilities spans varying levels of implementation cost and efficiency. Components implemented as separate system processes using a wrapper approach offer a quick way to prototype this approach at a cost of reduced efficiency. Thread-based implementations conforming with a component framework's structuring conventions offer increased efficiency at a higher implementation cost. A thread-based implementation can advantageously use wrapper libraries to transform existing operating system call primitives to interfaces to the encapsulation code. Finally, hypermedia technologies can be used to integrate a component's documentation with its encapsulated implementation [7].

3.2 Glue and Connector Implementation — Behavioural

Intent

Glue is used to interface components with the rest of the system while *Connectors* are used to interface components with each other.

Motivation

The primary data flow mechanism of all components is a *stream*. 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 standalone program's standard input and output as well as any other user-specified files. To

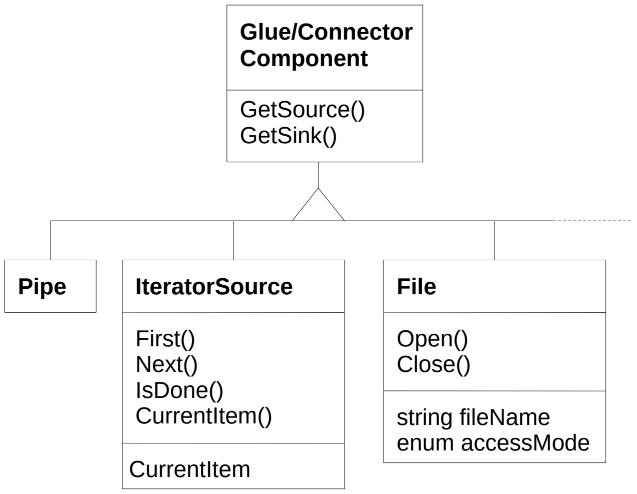


Figure 3: Component glue object diagram.

use a component effectively these streams need to be connected to existing data *sources* and *sinks* such as in-memory data structures, files, relational databases, procedures producing dynamic data, GUI widgets, and other components. These connections are handled by the glue components.

Applicability

A glue component class needs to be designed whenever a new type of an existing data source or sink needs to be linked to an encapsulated component. In addition, component connectors are used to provide functionality for linking components together and for providing additional features such as the scattering or gathering of multiple data streams.

Structure

As shown in Figure 3, glue components typically offer *source* and *sink* data streams. The streams they return can be used as arguments to a component's *SetStd** methods to connect a component to a specific stream. The glue component classes offer additional class-specific methods and variables to specify for example the connection attributes and SQL string for a database source or the methods for accessing iterator-based data structures. Most glue components act either as data sources or as data sinks. The *pipe* connector component provides both a source and a sink stream. Data written to the sink stream appears on the source stream; pipes are typically used to link together different components.

Participants

Glue components connect encapsulated components with the rest of the system while connectors are used to interface components of the same family between them.

Consequences

The provision of glue components allows the deep integration of encapsulated components within an object-based framework. Suitable glue components can be used to provide clear and efficient interfaces to system data obviating the cumbersome file-based approaches typically used to interface standalone 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.

Implementations

Although straightforward, the implementation of connectors and glue components is intimately bound to the implementation technique used for the corresponding encapsulated components. Process-based encapsulation techniques dictate the implementation of streams and glue components using file descriptors, pipes, and filesystem-visible file descriptors. On the other hand, thread-based encapsulation implementations direct towards connectors and glue components based on a shared-buffer producer-consumer model.

3.3 Component Composition — Structural

Intent

The component composition pattern identifies the primary methods of encapsulated component composition and integration.

Motivation

Encapsulated components do not operate in a vacuum. They are composed to create more powerful components and integrated within an object-based system to provide specialised services. Moreover, composition of encapsulated components with component glue can be used to provide efficient access to off-line data, graphical user interfaces, and a multitude of other component-based services. As an example a spelling checker can be easily constructed by composing the *translate*, *sort*, *unique*, and *common* components, while the gluing of a *editbox* and *listbox* components can be used to provide a GUI front end.

Applicability

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 object code. Meunier [13] describes a complete pattern language for a “Pipes and Filters Architecture” that can be used as a base to structure applications.

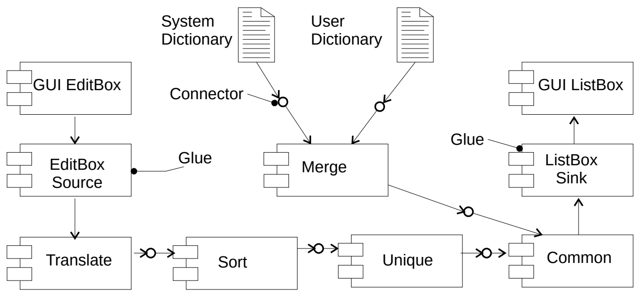


Figure 4: A spell checker with a GUI.

Structure

Figure 4 depicts the component interaction diagram of a filter-based spell checker built from Unix-mined and glue components. The text to be spell-checked is retrieved from the GUI edit box using a data source glue component. It is transformed into a list of words using the *translate* component which is a direct equivalent of the Unix *tr* command. The word list is then transformed into a sorted list of unique words using the *sort* and *unique* components which correspond to the Unix *sort* and *uniq* commands. At the same time, the system dictionary and a user dictionary are passed using appropriate file connectors to the *merge* component which merges two sorted streams; the merge component is a specialisation of *sort* which provides this functionality. Finally, the two sorted streams of words to be spelled and acceptable words are checked by *common* — derived from the Unix *comm* command — which outputs a list of words contained in the first stream and not contained in the second one. This stream of misspelled words is sent using the *ListBoxSink* glue component to a GUI list box. It is important to note that the integration of GUI elements using the same component object paradigm and the merging of two data streams could not be implemented using the standard Unix linear pipeline system.

Participants

The components composed are object instances of either active process components that are connected to existing data sources and sinks, or connector and glue components (pipes and environment interfacing classes) that provide such sources and sinks.

Consequences

Using the component composition pattern it is possible to implement sophisticated component interaction topologies. In addition, it is possible to package together existing components to provide new standard and reusable components.

Implementations

The implementation of the composition pattern is independent of the component-framework used. Most relevant decisions are taken when implementing the encapsulation and the glue patterns. Designs based on the composition pattern should be portable across different component frameworks.

4 Case Study

We used the component mining process pattern language to prototype the encapsulation of a number of Unix tools using the object-oriented features of the Perl programming language [5]. Following this proof of concept demonstration we decided to implement the components in a wider-used platform. We thus implemented a number of key components using Microsoft's OLE/COM technology [3] and used them from within Java and Visual Basic applications. In the following sections we outline the key features of the technology platform we utilised, describe the concrete framework we used for encapsulating components, explain how we automated the repetitive parts of the mining process, present an exemplar application we realised using the encapsulated components, and discuss interoperability issues.

4.1 Implementation Environment

The Microsoft Component Object Model (COM), our basis for packaging the components, is a software architecture that allows applications and systems to be built from binary components supplied by different software vendors. COM is the underlying architecture that forms the foundation for higher-level software services, like those provided by OLE — Microsoft's unified environment of object-based services. OLE services span various aspects of component software, including compound documents, custom controls, inter-application scripting, data transfer, and other software interactions. The basic features of COM include the definition of an efficient binary standard for component interoperability, programming language independence, dynamic loading of components, limited multiple platform support, and mechanisms for component communication across process and network boundaries.

COM components are packages of compiled code that conform to the model's conventions. In COM, applications interact with each other and with the system through collections of functions called interfaces. A COM interface is an immutable, strongly-typed contract between software components to provide a small but useful set of semantically related operations (methods). As an example, all OLE services (such as drag and drop) are simply COM interfaces. Clients interact with interfaces through pointers. Access to the component's data (i.e. public object members) is only available through interfaces. All components support a base interface called *IUnknown* which, apart from two reference counting methods, provides the *QueryInterface* mechanism that allows clients to dynamically discover whether or not an interface is supported by a component and get the respective interface pointer.

For languages that do not support pointers COM defines *automation*, an alternative way to access component methods through a standard late-binding interface called *IDispatch*. *Automation*-based component access is easier to program on the client side (it does not require the setup of a C-compatible stack frame) and is therefore widely used by text-based scripting languages such as Visual Basic for Applications, Perl, and TCL/TK. In addition, some languages (e.g. Perl, VBA) and language extensions (e.g. Visual C++ 5.0, Visual Java) provide syntactic sugar for accessing COM component "properties" exposed through a pair of specially tagged *propget*/*propput* interfaces, as public member variables. COM interfaces are described using the Microsoft Interface Definition Language (MIDL), a language loosely based on the OSF DCE IDL.

Implementing COM components from scratch in C++ is not trivial. Every component, in addition to its custom functionality, must support registration, an interface for creating component instances called *IClassFactory*, object creation, reference counting, the *QueryInterface* method, and, possibly, dual interfaces for supporting its use through C++ and automation-based scripting languages. Fortunately, these tasks are supported by the *Microsoft Foundation Classes* (MFC), a large, monolithic application framework for programming in Microsoft Windows, and by the *Active Template Library* (ATL) a leaner set of template-based classes that specifically target the development of COM components. We decided to implement the mined components using ATL. By aggressively utilising C++ templates and multiple inheritance ATL supports the development of COM components with brevity and minimal runtime overhead. A bare-bones ATL-based COM component can be implemented in less than 100 lines; most of them automatically generated by a "wizard"-type tool. We therefore found ATL to be ideal for implementing the large number of Unix-mined components and use as a basis to automate the task.

4.2 Encapsulation Framework

The *Unix Filter Component* (UFC framework we developed for encapsulating the Unix-mined components consists of the following classes:

```
command "uniq"

options {
    NumberOccurences:bool:-c:Prefix lines by the number of occurrences
    PrintDuplicate:bool:-d:Only print duplicate lines
    SkipFields:int:-f:Avoid comparing the N first fields
    SkipChars:int:-s:Avoid comparing the N first characters
    CheckChars:int:-w:Compare no more than N characters in lines
    PrintUnique:bool:-u:Only print unique lines
}
```

Figure 5: Description of the *uniq* command.

UFCFile Implements a connector-type component that is used for connecting Unix filter-type components to disk-based files for input and output. A *UFCFile* component can act as data source or a data sink.

UFCIO Implements a glue-type component that connects filter-type components to Windows edit controls for GUI-based interaction. A *UFCIO* component can act as data source getting input from an edit box or as a data sink sending output to an edit box window.

UFCPipe Implements a connector-type component providing a data-source/data-sink pair for connecting filters among each other.

UFCTee Implements a connector-type component for splitting the output of a filter into two identical data streams to be connected to other filters. *UFCTee* must be connected to appropriate data-sources and sinks.

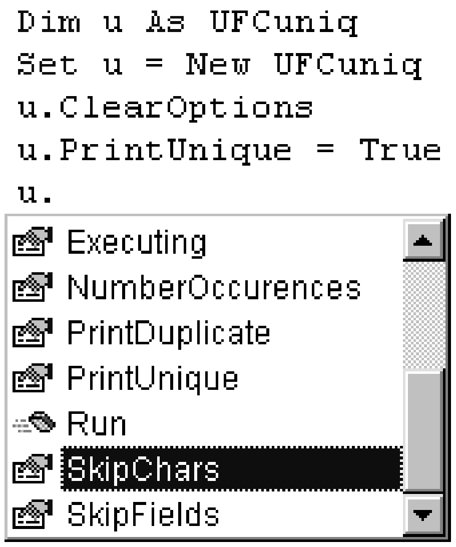
FilterBase A base class (not a component) used to provide basic filter-handling functionality. It implements the filter invocation method, member variables for setting the filter's standard input and output, input/output redirection, and a method for determining if a filter is still executing.

UFCFilter As a subclass of *FilterBase*, *UFCFilter* implements a generic filter-type component that can be used to encapsulate filters for which specific components have not been implemented. The filter executable file and its parameters are all specified using a *CommandLine* property. *UFCFilter* components must be connected to appropriate data-sources and sinks.

Filter components are typically executed as separate processes utilising existing collections of Unix tools ported to Windows such as UWIN [11] and *Cygwin* [16]. The *UFCIO* and *UFCTee* components support asynchronous operation by running in separate threads within the context of the application that uses them.

4.3 Process Automation

Based on the encapsulation framework, and the *FilterBase* class in particular, we defined a process and implemented support tools for automating the mining of Unix filters as

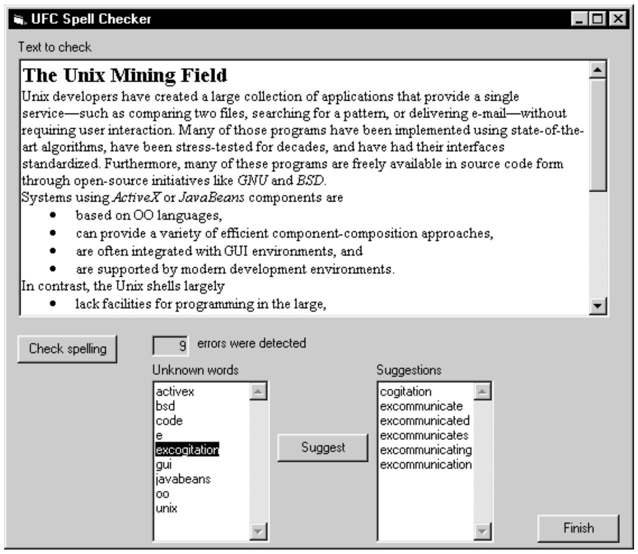


```
Dim u As UFCuniq
Set u = New UFCuniq
u.ClearOptions
u.PrintUnique = True
u.
```

Figure 6: Accessing the *UFCuniq* component within Visual Basic.

components. Specifically, for every filter that is to be converted into a component, one has to define the syntax and semantics of tool's command-line options using a small domain-specific language [21]. An example of this description for the *uniq* filter is depicted in Figure 5. For every filter command line option (e.g. -c) one specifies a meaningful name that is to be used as the respective component property, the option's type, the respective code expected by the filter as a command-line argument, and a descriptive text that appears for the given property in component object browsers.

A small compiler, implemented in Perl [23], compiles the declarative description of the filter interface into a C++ subclass of *FilterBase* that implements the respective component (e.g. *UFCuniq*), the header containing the class declaration, and the associated MIDL interface definition. The class contains a member variable for every filter command-line option, methods for getting and setting the member variable value (thus exposing the command-line option as a "property" of the component), a method for initialising the properties to a known state, and a method for executing the filter with a command line constructed dynamically to match the values of the component's properties. The component also inherits and exposes as properties the methods of *FilterBase*, namely properties for setting the filter's standard input and output, and a property for determining if a filter is still executing. Using the automated component mining process we were able to define new filter components at an average rate of four components an hour. An example of how the methods and properties of the automatically created *UFCuniq* component appear in the Visual Basic environment can be seen in Figure 6. Connector and glue-type



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*.

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,

9 errors were detected

Unknown words	Suggestions
activex	cogitation
bsd	excommunicate
code	excommunicated
e	excommunicates
excogitation	excommunicating
gui	excommunication
javabeans	
oo	
unix	

Figure 7: A GUI-based spelling checker built using UFC.

```
import ufcbase.*;
import java.io.*;

public class UFCSortClient {
  public static void main(String args[]) {
    IUFCFileDefault source = (IUFCFileDe-
fault) new ufcbase.UFCFile();
     IUFCFileDefault sink = (IUFCFileDe-
fault) new ufcbase.UFCFile();
     IUFCsortDefault sort = (IUFCsortDe-
fault) new ufcbase.UFCsort();

     source.Open(args[0] , 0);
     sink.Open(args[1] , 1);
     sort.putDataSource(source.getHandle());
     sort.putDataSink(sink.getHandle());
     sort.Run();
     while(sort.getExecuting() != 0) {
       ;
     }
     source.Close();
     sink.Close();
  }
}
```

Figure 8: Using UFC components in Java.

components still need to be written by hand, but the effort required to implement them is only a small part of the effort that would be required to mine a large number of filter-style programs without an automated process.

4.4 Exemplar Application

We used UFC and the mined components to implement a simple GUI-based spelling checker following the design outlined in section 3.3. The spelling checker was implemented in less than 100 lines of Visual Basic code. Its user-interface is depicted in Figure 7. The spelling checker utilises the following UFC components: *UFCIO, UFCTee, UFCPipe, UFCtr, UFCsort, UFCuniq, UFCcomm*, and *UFCwc*. Compared to a spelling checker implemented using a linear pipeline in the Unix environment, our component-based implementation offers the following enhancements:

- it provides a graphical user-interface,
- the errors detected can be interactively used to search for suggestions,
- it counts the number of errors detected utilising UFC's ability to implement sophisticated non-linear pipeline topologies, and
- it can check formatted text.

In addition, the application was implemented using a typed and modular language in a rich integrated development environment offering a syntax-aware editor, a sophisticated debug facility, a graphical interface builder, integrated help facilities, and source-code management. Third-party tools also provide support for profiling, automated source code examination, and browsing facilities. This level of support is sadly not existent in Unix-based shell-programming approaches.

4.5 Interoperability

The mined components can be used from any language supporting COM such as Visual C++, Visual Java, Delphi, Visual Basic, Perl, and TCL/TK. As an example, we used the UFC components from Visual Java by having the "Java type library wizard" provided by the environment create a special `.class` file representing the COM object. We were then able to import the UFC methods and use them as specified. An small example that sorts a file outputting the result in another file is listed in Figure 8.

Two important interoperability problems are associated with our approach. First of all, the resulting program violates the write-once, run-everywhere concept of Java as it uses the Unix filter components which are written in C and compiled for a particular processor architecture. In addition, UFC relies on COM, a proprietary technology, that is not universally available. We were able to offer a partial solution to these problems by developing a bridge that maps COM UFC objects into CORBA [17] objects. The bridge exports the UFC components as CORBA objects redirecting requests to the implementation of the respective COM components. Although the bridge is implemented in Visual J++, once it is installed and running, any system, processor architecture, and language supporting CORBA bindings can use UFC functionality. The object request broker (ORB) we used, ORBacus for C++ and Java by Object-Oriented Concepts, currently supports C++ and Java. In addition, OMG defines IDL language bindings for C, Smalltalk, Ada, and COBOL.

5 Concluding Remarks

The component mining process proved 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 opened new ways to leverage existing tools and applications. However, the process for mining and using the components is not yet as smooth and versatile as we would like. In particular, the non-standard semantics of Unix command-line option processing means that a number of programs with an idiosyncratic interface can not be automatically converted into components. In addition, the asynchronous execution of components as separate processes in conjunction with the new possibility to create arbitrary component interaction graphs (and not just linear pipelines) means that component users must carefully think about the issues of synchronisation and deadlocks. Finally, our component implementation — which is based on components executing as separate processes — may not be as efficient as components executing within the context of the application that uses them.

We are currently working on extending our component interface description domain-specific language to describe more sophisticated tool command-line options, experimenting with more efficient encapsulation techniques using threads, and planning to use our approach for constructing image processing applications from encapsulated tools such as the portable bitmap collection [18]. In the future we would like to see component mining extended to other mining fields, probably supported by different pattern languages applicable to the specific domains.

Acknowledgements

We would like to thank the anonymous referees for their insightful comments on the previous version of this paper.

References

[1] Alexander C, Ishikawa S, Silverstein M, Jacobson M, Fiksdahl-King I, and Angel S. *A Pattern Language*. (Oxford University Press, 1977).

[2] Booch G, Rumbaugh J, and Jacobson I. *The Unified Modeling Language User Guide*. (Addison-Wesley, 1999).

[3] Brockschmidt K. *Inside OLE*. (Microsoft Press, second edition, 1995), Redmond, Washington, USA.

[4] Brown A. W and Wallnau K. C. The Current State of CBSE. *IEEE Software*, 15(5):37–46, (September/October 1998).

[5] Conway D. *Object Oriented Perl*. (Manning Publications Co., 2000), Greenwich Ct, USA.

[6] D' Souza D and Wills A. *Objects, Components, and Frameworks With UML : The Catalysis Approach*. (Addison-Wesley, 1998).

[7] da Silva M. F and Werner C. M. L. Packaging Reusable Components Using Patterns and Hypermedia. In *Proceedings of The Fourth International Conference on Software Reuse (ICSR '96)*. (IEEE, 1996).

[8] DeMillo R, Lipton R, and Perlis A. Social Processes and Proofs of Theorems and Programs. In *Proc. Fourth ACM Symposium on Principles of Programming Languages*, pages 206–214, (Los Angeles, California, Jan. 1977. ACM).

[9] Gamma E, Helm R, Johnson R, and Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. (Addison-Wesley, 1995).

[10] International Organization for Standardization, Geneva, Switzerland. *Information technology — Portable operating system interface (POSIX) — Part 2: Shell and Utilities*, 1993. ISO/IEC 9945-2:1993 (IEEE/ANSI Std 1003.2-1992 & IEEE/ANSI 1003.2a-1992).

[11] Korn D. G. Porting Unix to Windows NT. In *Proceedings of the USENIX 1997 Annual Technical Conference*, (Anaheim, CA, USA, Jan. 1997. Usenix Association).

[12] Krueger C. W. Software Reuse. *ACM Comput. Surv.*, 24(2):131–183, (June 1992).

[13] Meunier R. The Pipes and Filters Architecture. In Coplien J. O and Schmidt D. C, editors, *Pattern Languages of Program Design*, chapter 22, pages 427–440. (Addison-Wesley, 1995).

[14] Mili H, Mili F, and Mili A. Reusing Software: Issues and Research Directions. *IEEE Trans. Softw. Eng.*, 21(6):528–562, (June 1995).

[15] Nierstrasz O, Gibbs S, and Tsichritzis D. Component-Oriented Software Development. *Commun. ACM*, 35(9):160–165, (Sept. 1992).

[16] Noer G. J. Cygwin32: A Free Win32 Porting Layer for UNIX Applications. In *Proceedings of the 2nd USENIX Windows NT Symposium*, (Seattle, WA, USA, Aug. 1998. Usenix Association).

[17] Object Management Group . The Common Object Request Broker: Architecture and Specification, (Oct. 1999), Also available online http://www.omg.org/library. January 2000.

[18] Poskanzer J and others . NETPBM: Extended Portable Bitmap Toolkit. Available online ftp://ftp.x.org/contrib/utilities/, (Dec. 1993), Release 7.

[19] Shaw M and Garlan D. *Software Architecture: Perspectives on an Emerging Discipline*. (Prentice Hall, 1996).

[20] Spinellis D. Explore, Excogitate, Exploit: Component Mining. *IEEE Computer*, 32(9):114–116, (Sept. 1999).

[21] Spinellis D and Guruprasad V. Lightweight Languages as Software Engineering Tools. In Ramming J. C, editor, *USENIX Conference on Domain-Specific Languages*, pages 67–76, (Santa Monica, CA, USA, Oct. 1997. USENIX).

[22] Szyperski C. *Component Software: Behind Object-Oriented Programming*. (Addison-Wesley, 1998).

[23] Wall L and Schwartz R. L. *Programming Perl*. (O'Reilly and Associates, 1990), Sebastopol, CA, USA.

[24] Wills A. Designing Component Kits and Architectures. In Barroca L, Hall J, and Hall P, editors, *Software Architectures: Advances and Applications*. (Springer Verlag, 1999).

[25] Yu H. Using Object-Oriented Techniques to Develop Reusable Components. In *Proceedings of the conference on TRI-Ada '97*, pages 117–124. (ACM, 1997).

Biographical Information

Diomidis Spinellis is an assistant professor at the Department of Information and Communication Systems, University of the Aegean, Greece. He holds an MEng in Software Engineering and a PhD in Computer Science both from Imperial College (University of London, UK). He is the author of more than 40 technical papers and conference presentations. He has contributed software to the 4.4BSD Unix distribution, the X-Windows system, and is the author of a number of public domain software packages, libraries, and tools. His research interests include Software Engineering, Programming Languages, and Information Security. Contact him at dspin@aegean.gr.

Konstantinos Raptis is a PhD student in the Department of Information and Communication Systems at the University of the Aegean. His research interests include distributed applications, software component models and distributed component interoperation technologies. Contact him at krap@aegean.gr.