APPLICATION MINING: REUSING LEGACY ASSETS

Paul Paetz

INTRODUCTION

Legacy COBOL systems are a tremendous source to improve productivity in building new applications, if the business rules can be cost effectively and quickly identified and “mined” for reuse. The term “application mining” has gained currency in marketing publications to describe this process of extracting the valuable business logic from old code, but little formal methodology or mining practice is employed. It sounds intuitively simple to identify the business rules and reuse them: “cut, paste, wrap.” Unfortunately, however, a simple, straightforward approach is impossible, due to years of haphazard maintenance; a lack of consistent, structured programming techniques and standards being used when the original development was done; and the intertwining of business rule logic with infrastructure and architecture code.

However, the term “mining” offers more than just a marketing label to the process; it suggests a methodology and some insight into how to formalize the techniques, improve productivity, and maximize the returns. This article examines what studying the age-old technology and techniques of mining can teach us, and reinterprets these lessons to the process of mining legacy software applications. The result is a set of tools and methods that improves the process of rule discovery, refinement, and reuse by more than ten times, when compared with using code analysis, editing, and slicing tools alone.

PAYOFF IDEA

Because legacy functionality is a valuable business asset, business functionality from existing systems should be reused and adapted to implement powerful Internet applications. Application mining is a useful metaphor to guide the development of techniques for mining legacy applications. Salvaging the business value embedded in legacy applications is necessary to move forward, and is the low-cost, low-risk way to do so.
that legacy functionality is a valuable business asset that \textit{should} be reused and adapted to implement powerful Internet applications. In addition, the article illustrates that the process of salvaging the business value embedded in legacy systems is not only necessary to move forward, but is now the low-cost, low-risk way to do so.

**DEMAND FOR NEW CAPABILITY**

Software cannot be built fast enough for current needs, and the situation is getting worse. The impact of information technology (IT) on business methods and processes has never been as all-encompassing as it is today. The Internet alone is changing the nature of an organization’s business transactions with other businesses or end consumers, as well as the expectations of ubiquity (everywhere), availability (anytime), reliability (absolute), and cost (nearly free). These factors are compounded with other externalities such as the completion of Year 2000 (Y2K) remediation and euro currency implementation, which have both extracted or are extracting tremendous resources for no functional improvement. As a result, organizations are faced with an unprecedented demand for new capability, and order-of-magnitude improvements in how quickly existing systems functionality can be adapted to meet rapidly changing requirements.

Given the conflicting scenarios of a diminishing supply of talent and time, and an increasing demand for new functionality, organizations have a clear and urgent need to find alternative means to accomplish the productivity and quality goals before them. Programmer tools such as advanced (and portable) languages, fourth-generation languages (4GLs), interactive development environments (IDEs), easy-to-use graphical interfaces, and application generators have all been tried with limited success. At best, these tools offer 25 to 50 percent gains in productivity because they accelerate the way things are done now, without any real changes in methodology.

It has been known for some time that software reuse (i.e., component-based development), when accompanied by dramatic overhauls in development methodology, can offer order-of-magnitude improvements. However, the success of such efforts has been spotty and limited, due to such factors as:

- limited availability of components to reuse
- few standards for creating interoperable, sharable component libraries
- the lack of a crisis situation to force the needed methodological and organizational changes
- a natural programmer bias to rewrite or write new applications from scratch
- incompatibility between various generations of application technology, which inhibits reuse
• the huge amount of functionality that is already implemented but not easily reused because of:
  – the difficulty in separating what is useful from what is not
  – the speed at which technology changes, and makes the previous generation of applications obsolete

Nevertheless, no other method even comes close to reuse in meeting the current need for new capability.

TOWARD REUSE SUCCESS
Many of the reasons why software reuse has not been as successful as expected have recently been addressed, or are in a state of imminent change. For example, component-architecture initiatives by major vendors and industry consortiums are creating standards (including COM, CORBA, and EJB), and are making available a ready supply of components to reuse. And the ability to “wrap” old application code using these standards does enable a limited kind of reuse and intergenerational compatibility.

Misperceptions
The natural bias of programmers is really a misperception rather than an accurate assessment. Netron’s experience (in particular, as a purveyor of tools to support software reuse) is actually that programmers prefer to do the least work to accomplish any given task, rather than start from scratch each time. The unfortunate truth that underlies the above perception is that it has usually been easier for programmers to write from scratch than to reuse anything, due to the lack of standards and the difficulty in finding the component they need for their requirement.

Moreover, it would be difficult for anyone to argue that organizations have not reached the desired crisis state. The initial generation of dot.com companies has shown that business can be done effectively and at lower cost on the World Wide Web, and use more efficient paradigms than was thought possible. Now the traditional “brick and mortar” companies, while they do have the advantage of organization, brands, profitability (in most cases), and established functional business methods, are in a position where they must respond quickly or rapidly lose market share to the upstarts. Not only must they respond quickly, but they must be able to maintain a much faster rate of change than in the past — all with expensive and scarce resources.

The only serious remaining issue is the inability to (re)use the wealth of existing applications code. This is divided into two separate and distinct sub-issues.

The first is the (perceived) obsolescence of previous generations of applications based on older technology. Actually, old technology itself is
not necessarily an inhibitor to reuse because it can be wrapped in a modern interface to communicate with new architectural standards. Instead, it is largely old application architectures that make the applications obsolete. Still, in most cases, even the most radical E-business implementation changes less than 20 percent of the existing business processes, and it is often far less than that. This naturally leads to the second issue: that is, how the useful bits can be separated from the architectural and infrastructural noise.

**Separating the Useful Bits**

This has given rise to the concept of application mining as a tool — a supported technique for extracting the useful reusable bits. However, to date, the phrase “application mining” has been just that: a marketing metaphor rather than an analogous process to the modern and efficient means by which valuable raw materials are found and extracted from the earth. The balance of this article examines what organizations can learn and apply methodologically from the mining metaphor to mining existing applications code.

**MINING METHODS**

Mining is the lowest of low-tech industries. It typically operates on very low margins (unless a high demand resource is in short supply), which depends on careful and strict management to minimize input costs and maximize output. It is labor and capital intensive. Yet, for well-managed operations, it continues to be a profitable business because it applies engineering and methodology to the process.

Although mining operations vary a little by the type of material being extracted, they typically consist of the following phases:

- terrain mapping
- prospecting
- extraction
- refinement
- use

The activities that occur in each of these phases are expanded upon below.

**Mapping the Terrain**

The primary purpose of first mapping the terrain is to narrow the scope of where to prospect. This is because, usually, the minerals or other raw materials to be extracted are not easily detected by the five senses. Instead, they are often buried below the ground in non-obvious places. In other words, companies could drill for oil everywhere (the brute-force
method), but because this is highly inefficient, they might spend a lifetime and never find anything.

The scope narrows when the problem is understood. That is, based on previous experience and applying geological knowledge to studied analyses, companies can determine the most likely places to look for the desired resources.

This often starts with an aerial survey, whereby photographs are taken by plane or by satellite. Once the regional characteristics have been identified at a macroscopic level (literally, a 30,000-foot view), a detailed geological survey is prepared. Next, major geological formations are identified and compared with the characteristic attributes of known formations. This is often accompanied by computer-automated analyses, which look for patterns and assess the probability of a strike at any of the possible mining sites.

The findings are classified and categorized, and the best candidate sites for digging are located on the map. A significant amount of time is spent on this analysis before the first shovelful of dirt is turned over. Why? Because even when the geological patterns indicate that there may be something worth further investigation, only a small fraction of the candidate sites usually turn up anything useful. This upfront analysis saves a tremendous amount of time and money by focusing attention on the sites with the highest probability of success right from the start.

**Prospecting**

Once the candidate sites have been narrowed down, there are still no guarantees that anything will be found when the site is dug. However, the probability of a strike has been significantly increased. At the prospecting stage, the first step is to stake a claim to the best candidate sites where the rest of the time will be spent (unless the sites are already secured by deed or mining rights).

If there are any loose deposits, panning and sifting or its logical equivalent is in order to collect some easy returns. If not, digging or blasting begins, with the objective of finding the veins; that is, the rock rich in deposits. Poor or difficult-to-mine sites are eliminated from further exploration. The objective is to identify the 10 percent of sites that will yield 90 percent of the returns.

Prospecting can also be described as detailed site analysis and exploration. As the first phase of real mining activity, it provides a detailed understanding of where and how to extract the deposits and enables early returns to be achieved.

**Extraction**

This is the phase that most people not in the business associate with mining. It is where the mines are dug, the veins are exploited, and the ores
are collected for refinement and transported to the mill. It is where the majority of drudgework, and therefore the majority of mechanical automation, can be applied to the process. Tools applied are explosives, digging machinery, conveyor belts, train cars, elevators, etc., in addition to significant human labor.

**Refinement**

Ores are melted down to separate the metals from (worthless) rock slag. The molten metal goes through various steps to create a finished product of the desired purity and usability, and ingots or rolled sheets of metal are formed. These are the raw materials that are delivered to factories for use in fashioning finished products. This involves (relatively speaking) that a small amount of human labor input and decisions be made, and is largely automated.

**Use**

Not really part of the mining process, this is why people mine in the first place. While usage is fairly obvious in the physical world, when referring to software development, it may appear to be less obvious and more abstract. That is a fallacy, because no software excavation project ever gets funded without an expected outcome — whether it is to understand and document the requirements for a replacement system, rearchitect as a component-based Web-enabled application, or modularize to improve maintenance cost, while retaining COBOL and the same underpinning architecture. However, the specific objective may direct how and what project activity occurs. Therefore, usage is included here as the final step to illustrate the parallel in software.

**APPLICATION MINING**

**Characterizing Existing Systems**

Existing applications have been constructed over many years, by many different people with different coding styles, using ad hoc methods. They have been patched endlessly with the immediate purpose of keeping things running, rather than preserving architectural integrity and code readability. Moreover, if there ever was documentation, it is almost certainly out-of-date and does not reflect actual business processes implemented in the code. It is these characteristics that force us to define a repeatable, learnable methodology, rather than a bunch of ad hoc extraction processes, and that suggest that there may be something to be learned from the mining analogy.

For example, the code is often (usually) poorly structured “spaghetti.” Business logic is not isolated in any given place in the code, and individual business rules are often distributed throughout the code. It is seldom
even obvious that a particular piece of logic is a business rule or part of one, even when one is looking right at it. Architectural code is freely mixed with business rule code. And, strikingly, most organizations do not even know what business rules exist in their code — at least not completely. That is, the only documentation of what exists and how it performs is often the code itself.

**Needle in a Haystack**

The purpose of application mining is to identify and extract the 5 percent of existing code that defines business processes as they exist today; in other words, to “separate the needles from the haystack.” In a legacy application, particularly if the objective is to re-architect it for the Web, this is the only code that is of value. The remainder defines the application architecture, screen handling, database structure and input/output (I/O), flow of control, and necessary housekeeping infrastructure that every program requires. That is, while it is required to execute the program, it is of no value from a purely business perspective.

This is not dissimilar to the vein of gold running through solid rock. Although one can plainly see the gold when the vein has been exposed, there is much around the vein in lower concentration that is also gold, and none of it is easily separated from the rock.

“Componentizing”

“Componentizing” the mined applications is often a secondary objective. When the application is reconstructed, one can address issues such as cost, ease, and speed of change (i.e., adaptability of the application), in addition to maximizing reusability of the extracted pieces by isolating related functionality into discrete components.

Choosing what to componentize and how to prioritize what to work on first is part of the mining methodology. There are three essential qualities to identify in prioritizing componentization efforts: complexity of the logic, volatility (how often it changes), and how critical the functionality is to the business. In total, these categories add up to 30 to 40 percent of the total business logic. These qualities are roughly the equivalent of market size and stability, estimated mass of minable material, and projected profitability as decision factors regarding whether or not to establish a new mine.

In some cases, it may make sense to end the process of (invasive) componentizing when these parts are finished. In other cases, documentation of all functionality is needed, or the application is being rewritten into another technology (e.g., a business rules engine). Therefore, each rule must be fully defined, whereupon the process may continue until the entire system has been documented and deconstructed.
In the following section, each of the mining phases is reexamined in the context of software to see what can be learned.

MAPPING THE TERRAIN
As in the physical world, the primary purpose of this phase is to narrow the scope of where to prospect. This is because of the difficulty in identifying where business rules are found in the code, and what processes one is looking for in the first place. If it could be assumed that one already has a complete knowledge of what and where the business rules are, (i.e., that the code was well structured and documented), then one could simply use a text editor to cut-and-paste the relevant sections, and simple utilities (or manual coding) to wrap them.

Unfortunately, this is never the case, and the larger the system, the greater the complexity that needs to be unraveled. In a small system, this might actually be done manually; but in an application with just a few million lines of code, it is conceivable that a team of programmers could spend their entire careers trying to gain an understanding of what and where the rules are in sufficient detail to extract them all for reuse without tools. In addition, this assumes that the application need not change any time during the work in progress.

Gaining Understanding
The scope is narrowed by gaining an understanding of the problem. Fortunately, in software, a much larger percentage of this can be automated than in the real world. For example, off-the-shelf parsing tools are readily available that can create diagrams of the code structure and relationships to facilitate initial understanding. However, most available tools rarely go much further than that, and this is where much can be learned from geological engineering.

The task of taking and examining a geological survey is primarily one of creating a detailed map and using it to identify patterns that correlate or are indicative of possible mine sites. These patterns are identified by feeding a picture of the terrain and key attributes about it into sophisticated analysis algorithms. Pattern-based analysis is the sort of thing that computers excel at, but that in an even moderately complex application would be overwhelming for programmers to try to undertake manually as part of application mining. The question becomes how to identify useful patterns in software.

Finding Patterns
This is an important question because to be truly effective, the examination and analysis of these patterns should not be sensitive to the language in which the rules were written. The answer is to create an
abstract representation of the attributes of any given fragment of code. This assigns values to key characteristics of the code, such as code flow, types of statements used, recurrence of certain statements, statements used in combination with other statement types, etc. This abstraction can be called a code fingerprint.

Such fingerprints can then be derived for specific pattern types that one wants to search for. For example, archetype patterns can be defined that represent what business rules typically look like, and for other types of objects that one may want to rapidly identify as well.

An example of a tool that uses this process is HotRod from Netron. This tool starts from the premise that abstracting and filtering patterns in process-oriented code (i.e., looking for business rule patterns) can identify the locations of all the rules in a single classification of the code inventory.

As suggested above, the logic executed by the tool is much more difficult and complex than could be effectively performed by an individual. Thus, while it is theoretically possible for an individual programmer to work this way, an application of even moderate size and complexity renders the task impossible from a time and practicality perspective. Algorithms such as a Bayesian search (used by NASA to identify star patterns) are used to match fingerprints and identify common code. This hunting is all done by the computer automatically, not by a programmer “hunting and pecking” his or her way through each rule one at a time, as is typically the case in logic-tracing, brute-force sorts of tools. That is, it makes the programmer’s job significantly easier.

**Predictive Scoring**

Because one of the purposes of this phase is to focus attention on the 5 percent of code that matters, one also wants to score candidates and rank them in terms of how closely they match archetypal rule patterns, or on the likelihood that they belong in a particular group or classification that can identify all the different places in which a similar rule occurs. If something is known about the characteristics of the code being examined, this ranking can also be used predictively. That is, a score that deviates sufficiently from indicating a rule match can indicate almost immediately that the code chunk is definitely not a rule.

Exhibit 1 shows the results of a pattern-based business rule search in the leftmost window. While it cannot be determined by looking exclusively at the paragraph names in this list whether or not one has found a business rule, the labels certainly indicate a fairly high probability that business rules have indeed percolated to the top.

Moreover, many business rules were created by cloning existing business rule code and adapting it to fit a different context or additional decision criteria. When extracting rules out of existing systems to
EXHIBIT 1 — Pattern-Based Business Rule Search Results
componentize them, the use of pattern-based searches to identify clones helps to eliminate redundancy, driving out maintenance costs and ensuring a complete rule definition that exists in only one place.

**PROSPECTING**

Prospecting in the real world begins with identifying the 10 percent of candidate sites that yield 90 percent of the return, and staking a claim to those sites. In application mining, the rule candidates were identified using pattern-based analyses. To begin the prospecting phase in application mining, such analyses can then be aggregated, and evaluations can be made of the time and cost to extract rules (i.e., creation of a project plan). Here, however, on average 5 percent of the code will yield 100 percent of the beneficial results.

As in mining for gold where prospectors continue by panning and sifting for the easy returns, in application mining the process would continue by identifying potential components with a quick or immediate payback. This both helps to pay for the project as one proceeds and provides more input into how difficult and costly it will be to complete the entire project.

On further inspection, false positives (poor candidate mines) are eliminated from the candidates list. (Because it was created 100 percent automatically by pattern-based analysis, there are almost certainly some code fragments that exhibit rule characteristics, but are not really rules.) As work continues down the ranked list (from top to bottom), annotations can be added to describe what the rules do and their relationship to other rules. This process continues until the threshold is reached below which no further rules are found.

**EXTRACTION**

As in physical mining, this is where the bulk of manual effort is spent. Each rule candidate must be examined individually, and the appropriate component boundary or entry point decided. Tools can support this process by assisting in the inspection of the “perform tree,” determining, for example, where the identified code fragment is used in the program, what other code paragraphs are part of its neighborhood, and if some or all neighbors are part of the rule definition. A decision tree for scoping rule boundaries is shown in Exhibit 2.

Once the scope of a component definition is assessed, the data used within that scope needs to be classified into input, output, or local categories — because all data in a COBOL program is global. Once the scope is bounded and the data is classified, all that is necessary to create a component has been determined. In the screen image used as an example for mapping the terrain (Exhibit 1), the result of classifying the data is shown in the middle window screen.
EXHIBIT 2 — Scoping a Component

- Finished Upward Search
  - No: Component Always Invoked from the Same Place?
    - Yes: Component Scope
    - No: Is Calling Block of Code a Better Entry Point to Component?
      - Yes: Add Calling Block of Code to Component Scope
      - No: Finished Downward Search
  - Yes: Add Calling Block of Code to Component Scope

- Are There Subfunctions?
  - Yes: Finished Downward Search
  - No: Is Subfunction a Natural Element of the Current Component?
    - Yes: Add Subfunction to List of Components to be Designed
    - No: Is Subfunction a Natural Element of the Current Component?
      - Yes: 
      - No: Add Calling Block of Code to Component Scope
Business Logic Tracing
Alternatively, if one know the key data that is used in determining a result (i.e., inputs to a rule), all the logic that contributes to this result can be traced through to its conclusion. Or, if one knows which data is the result of some transformation (i.e., output data), one can use tools to trace this logic back to its origin. This technique of business logic tracing is also known as code slicing, and, in a solid methodology, this data-oriented search ensures completeness and provides another means of determining the correct component boundary.

Unfortunately, in most available technologies, this method is used exclusively. This is unfortunate because it depends on knowing in advance where the rules are or which data is important to deriving a rule — something that is almost never known except for the obvious rules. Using this technique exclusively is like randomly choosing mine sites to dig, and using a pickaxe as the only tool. When used alone as a technique, it is slow, tedious, and prone to miss some critical rules.

REFINEMENT
As part of extracting a rule, it may be desirable to create a component that includes all variations of a rule that have been cloned and adapted over time as part of a single rule definition. It may also be desirable to clean up the logic, or bring it up to a more modern standard than that for which it was originally written. Like melting down the ore and purifying the metal, this is where one puts the finishing touches on a component and maximizes its reusability. These processes can be aided by tools and sophisticated editors.

When the content is right, the last step can be fully automated; that is, the wrapping of a business rule to turn it into a component that conforms to some architectural standard. This is like the rolling of metal or the creation of ingots. These components become the raw materials in an assembly factory to create finished goods. Examples of generated component types are COBOL subroutines, COM, CORBA, EJB, or XML documents.

USAGE
Usage is fairly obvious in the real world, but it can easily be forgotten when talking about the theory and methodology of application mining — that no one componentizes an existing application for its own sake. For this reason, it is worth repeating that the desired outcome is a factor in the specific application mining techniques that can be used, and must be thought of right at the start when constructing a project plan.

In the most trivial reuse of software, the point of application mining is simply to document and create a repository or catalog of all the functionality that exists. This can be used for gap analysis if the system is being
replaced by a package, or as a reference when rewriting the application in another technology.

In a simple E-commerce or E-business application, the objective may be to expose and componentize key order entry functionality so that an organization can start selling over the Internet. In this case, one is not aiming for exhaustive re-architecting, but selective extraction, so the factors of criticality, complexity, and volatility assume prime importance in deciding which pieces of the application to extract.

In a strategic re-architecting project, organizations must consider the long-term usage of the functionality and the appropriateness of the architectural model for supporting transaction throughput, scalability, security, adaptability, and integration with other processes. In this case, a thorough top-to-bottom componentizing project is probably in order, and will include development of new code and infrastructure, including browser-based clients, message-oriented middleware, and application server software.

**SUMMARY OF APPLICATION MINING PROCESS**

Numerous techniques for getting the nuggets of valuable code out of existing applications can be learned and understood by the comparison with mining. In comparing and contrasting mining and application mining, a deeper understanding of how to solve the problem has been gained but the question of why has not been answered. Before concluding, this section reviews the reasons to undertake such a project, and pulls together all the techniques discussed above into a step-by-step process.

In general, there are three categories of reasons to mine applications: namely, understanding the business processes, (re)using business rules, and technical requirements. Each is further evaluated below.

**Why Do It**

**Understanding Business Processes.** As noted near the beginning of this article, software systems are used to automate business processes; yet, in most cases, there is no documentation of either the systems or the processes that they automate. In other words, the only accurate representation of actual processes used in a business is often the code that automates it.

In the case of understanding business processes, the task of application mining can be thought of as reverse engineering the meaning out of the code. Organizations commonly want and need to do this to plan future enhancements to their applications, or to evaluate and customize packages.

A third less obvious use of this detailed level of understanding is to consolidate systems that have a high degree of redundancy. This reuni-
dancy can often creep up over time as packages are bought that have overlapping functionality with existing systems, or as different teams have needs for similar but slightly different functionality in different systems. Redundancy can also result from sudden duplication of systems, such as frequently occurs in mergers and acquisitions (M+A). In many cases, companies that have grown quickly using an M+A strategy have half a dozen or more instances of systems with nearly identical functionality that are being maintained at huge, unnecessary cost. Using an application mining process can quickly identify where the redundancies are, thereby enabling their elimination.

(Re)use of Business Rules. The most compelling reason to reuse existing business rule logic is to accelerate the time to market for new E-business systems. A much quicker, less costly, and less risky alternative to rewriting functionality is to extract a rule and wrap it with a component interface that is able to interoperate with other newly created components, as well as modern languages and application server technology. Incidentally, the latter alternative usually performs better.

Mining rules, but leaving them in their original COBOL form, is a means to significantly improve maintenance costs by modularizing and isolating the key processes together. This is because once the functionality has been isolated, the impact of making a change can also be isolated. The complexity of implementing the change correctly is also greatly reduced. Together, these changes mean less programming time, less testing time, fewer errors, and therefore less debugging. These effects are amplified in systems that are critical to the business, very complex, or that change frequently.

Business rules can also be explicitly and selectively mined for the purpose of creating a repository of components to be used in new development. Similar to the first two examples, the purpose of doing this is to improve development productivity and to reduce the risk of implementing new systems. Risk is reduced because the functionality is drawn from code that is known to work reliably and correctly, rather than by writing new code that always carries the risk of mistakes, or even total failure (70 percent of software projects started are never finished).

In any of these scenarios, reuse of modular components improves the adaptability and therefore the speed at which systems can be changed. This is perhaps the least obvious but most important long-term benefit of componentizing, because one of the most critical drivers of business success is the ability to respond quickly to changing requirements and regulatory and competitive pressures.

Technical Needs. Issues that are more technical in nature can also drive application mining projects. For example, many organizations have made an architectural policy decision to implement systems that are
component based or rules driven. Application mining can be used to identify legacy rules to input to a rules-driven approach, or to extract components that are compatible with or reusable by new component-based initiatives.

A second type of infrastructural change is the re-hosting of applications to a new platform. In this case, all the code that performs screen display, database I/O, flow of control, inter-process communications, etc. is to be discarded, and only the business rules have any value at all.

Another set of needs that larger organizations often face is that of systems that are scalable, reliable, performance oriented, and highly secure. These qualities are typically already embedded in the legacy code, while newer languages or architectures lack these features. Organizations gain the best of both worlds if they wrap legacy components so that they can be used to provide services to E-business applications, yet leave them on the mainframe to continue to perform as they always have, yet in a different context.

Finally, there is the systemic issue of efficient resource utilization — in this case, human resources. There is a wealth of COBOL/legacy programming talent, and an urgent need to deliver more functionality than there are people to build it. Organizations cannot afford to leave useful skills sitting on the bench for political reasons (e.g., the political incorrectness of using COBOL to deliver new applications).

In bulleted form, the business value of mining rules can be summarized as follows.

- understanding business processes to:
  - plan future enhancements
  - consolidate systems
  - evaluate and customize packages
- (Re)using rules for:
  - re-implementing in E-business systems
  - modularizing systems to reduce maintenance
  - speeding time to market
  - improving adaptability and responsiveness to change
- Technical needs:
  - performance, security, reliability, scalability
  - to facilitate re-hosting
  - component-based architecture
  - rules-driven application development

**Application Mining Process**

The key phases of the application mining process are presented in Exhibit 3 in terms more usual in an application design and development
methodology. Terrain mapping is shown as analyze, prospecting as design, and extraction/refinement as generate.

**CONCLUSION**

Application mining is a relatively new approach to leveraging the value in existing systems and reusing functionality in new component-based architectures. It is necessitated by the current crisis of escalating software development costs, resources shortages, competitive pressure, new business models, and new technology infrastructure — all happening concurrently.

Moreover, it is highly unlikely that the pace of change and increasing technology complexity will ever subside. Therefore, it must be assumed that multiple generations of technology will co-exist in every IT environment, and that time and cost constraints will prohibit the rewriting of legacy systems just because they were written with old technology. If such systems continue to meet a current business need, they will be retained first, re-architected second, and rewritten only when there is no other alternative.

Common sense dictates that to maximize productivity, software reuse will be the preferred choice, especially if the fundamental business processes have not changed. However, to reuse software, it must be architected and mined to be reusable. Examining the well-established techniques of mining and applying them to the process holds the promise that effective reuse may finally be upon us.

In summary, then, it is the position of this article that:

- Legacy functionality is a valuable business asset, not garbage to be discarded.
Business functionality from existing systems should be reused and adapted to implement powerful Internet applications.

Application mining is more than a marketing phrase; it is a useful metaphor to guide development of techniques for mining legacy applications.

Salvaging the business value embedded in legacy systems is not only necessary to move forward, it is now the low-cost, low-risk way to do so.

© 2000 Netron, Inc. Printed with permission.

The director of E-business strategies at Netron, Inc., Paul Paetz is an expert in methods for business rule discovery and retrieval, and applying them to E-business. With more than 15 years of experience in component-based development and reuse, he works with system integrators, partners, IT media, and industry analysts to market HotRod, Netron's application mining solution.