Legacy Software Reengineering

Frank Weil, Ph.D.
UniqueSoft, LLC
UniqueSoft is a provider of next-generation software development tools and services specializing in modernizing legacy software using highly automated tools and techniques to achieve superior results. UniqueSoft helps its clients through the various phases of a major transformation of their legacy systems, including analyzing the full system, formulating a low-risk and high-value modernization strategy, and executing on the elements of that strategy. In this white paper, we describe the UniqueSoft process and toll for reengineering client software systems to create a modern, easy-to-maintain, and high-quality system in a cost-effective manner.

Contents
1. Overview .......................................................................................................................... 3
   Types of Reengineering Projects .................................................................................. 3
   UniqueSoft Legacy Reengineering Capabilities ......................................................... 5
   Competitive Comparison .............................................................................................. 7
   Example Reengineering Projects .................................................................................. 8
2. The UniqueSoft Reengineering Process ......................................................................... 9
   Transformation .............................................................................................................. 15
   Instantiation ..................................................................................................................... 17
3. Scalability ....................................................................................................................... 20
4. Summary ......................................................................................................................... 20
1. Overview

UniqueSoft is a provider of next-generation software development tools and services specializing in modernizing legacy software using highly automated tools and techniques to achieve superior results. UniqueSoft helps its clients through the various phases of a major transformation of their legacy systems, including analyzing the full system, formulating a low-risk and high-value modernization strategy, and executing on the elements of that strategy.

In a general sense, reengineering legacy software is the process of improving the nonfunctional attributes of legacy software without changing its external behavior. That is, the legacy is made better or more suitable in some way. However, one desired improvement should not come at the expense of making other software attributes markedly worse. For example, migrating from a mainframe to Linux should not make the resulting code unmaintainable, require a mainframe emulation engine, or leave scattered remnants of the code’s mainframe heritage. At best, such an approach is a temporary fix that still leaves the deeper problem: the resulting code is not what one would want to maintain.

An effective reengineering process must go beyond indiscriminately applying a limited and fixed set of changes to legacy code. Instead, the process must allow the full goals of the project to be achieved. For example, translating COBOL to Java should result in code that is not merely correct, but is also maintainable, uses Java language features (inheritance, encapsulation ...), and perhaps more importantly, does not use COBOL idioms (flat memory space, manipulation of data as strings, GOTOs ...). Refactoring code to improve maintainability and reduce defects should result in code that is well structured in terms of its features, has dead code eliminated, minimizes replicated code, has a low overall cyclomatic complexity, etc. Extracting business rules from code should result in rules that are logically grouped and that use the business terminology familiar to the business stakeholders as opposed to the often-impenetrable names used in the source code.

This white paper discusses how the UniqueSoft modernization process and tool make these goals possible.

Types of Reengineering Projects

Reengineering projects for legacy software systems typically fall into three broad categories: migration from one platform and/or language to another, correction of defects or deficiencies in the code (e.g., high complexity), or extraction of information such as business rules from the code. Projects often span more than one of these categories, and every legacy-reengineering project has its own unique goals and needs. Categories of projects are summarized in Table 1.

<table>
<thead>
<tr>
<th>Type</th>
<th>Project Goal</th>
<th>Example Projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Migration</td>
<td>Platform migration</td>
<td>Client-server to cloud; mainframe to Linux</td>
</tr>
<tr>
<td></td>
<td>Database migration</td>
<td>MySQL to Oracle</td>
</tr>
<tr>
<td></td>
<td>Language translation</td>
<td>COBOL to Java; C to C#</td>
</tr>
<tr>
<td>Correction</td>
<td>Design improvement</td>
<td>Modularize application to improve maintainability</td>
</tr>
<tr>
<td></td>
<td>Code improvement</td>
<td>Remove defects and dead code</td>
</tr>
<tr>
<td>Extraction</td>
<td>Business Rule extraction</td>
<td>Document insurance application rules as currently implemented</td>
</tr>
</tbody>
</table>

Table 1: Reengineering Project Types and Examples
The overall goal of migration projects is to move the behavior of the software from one enabling technology to another. For example, the legacy system may have a client-server architecture and the goal is to make it cloud enabled, or may be currently on a mainframe and the goal is to move it to a Linux-based server, or may have originally been built with a MySQL database and the goal is to move it to a high-performance Oracle implementation, or may be in an older language such as COBOL and the goal is to move it to more modern language such as Java. The common theme is that the goal is not to change the application behavior, but rather to change how the application is written and deployed.

The overall goal of correction projects is to correct (or improve) some aspect of the software. This improvement generally falls into one of two broad groups. The first group covers legacy systems that are functionally correct but need design improvements, such as when the software is difficult to maintain or enhance. Typical improvements are to restructure the code to reduce its complexity, to improve its modularity (e.g., to group the code implementing a feature into a single module), to reduce the coupling between modules, and to replace replicated code with calls to a single function.

The second group of corrections covers legacy systems that require quality improvements. While some improvements in this category are straightforward, such as the removal of dead code, others require complex analysis of the software to discover locations in the code that are potentially incorrect. This is not a solvable problem in general because a tool can't know what the software is supposed to do. A tool can, however, point out two type of suspicious coding. First, analysis can locate “anti-patterns” in the code. For example, potentially incorrect code may include decision statements in which some values are not covered by branches, calls to external library functions which do not check the return value, or use of unsafe functions such as the C strcat routine. Even the presence of dead code may in itself indicate an error in the code. Second, a tool can statically examine the control and data flows of the system to locate potentially incorrect behavior. For example, analysis can locate use of a variable before it is assigned a value, data that is created and then discarded, or places where an index variable has a value outside of the array bounds.

The overall goal of extraction projects involves uncovering information in the legacy system. Two kinds of information can be extracted: intrinsic and extrinsic. Intrinsic information about legacy software relates to what can be extracted purely from the code itself. That is, it is independent of what the code is supposed to do or why. Examples of intrinsic information about legacy software include number of lines of code, number of files, the directory structure, what calls are made to external libraries, what databases are accessed, etc. Extrinsic information about legacy software relates to what the code does in the terminology of the stakeholders. For example, one may want to understand what features are implemented in a system, where in the code those features are implemented, what business rules are realized, and how those business rules relate to each other.

All three types of reengineering project are inherently difficult, and that difficulty grows with the size of the legacy software. For example, consider migrating a legacy application to the cloud. Cloud migration poses challenges because the cloud applications must be stateless, must not depend on local storage, should scale dynamically, require monitoring and management tools that are cloud-capable, must be in a supported language, and must have additional cybersecurity considerations. Properly migrating local file access, for instance, requires not just knowing where in the code the file accesses are (which is simple to find), but also why each local file access is being done: to store temporary transaction information, to store permanent
information, to log the transaction, or for some other purpose. Knowing the *intent* of the file access provides the information on how the access should be transformed.

Similar knowledge is also needed for the other types of reengineering projects. For example, one cannot create feature-based modules unless one understands what features are implemented in the code and where they are implemented, and one cannot extract business rules that are meaningful to the business analysts unless the extracted rules use the analysts’ terminology.

Once this software intent is uncovered, how that information is used is equally important. One use is for the system analyst to quickly and interactively build an understanding of the important aspects of the system. For example, software maps can be used to graphically represent the information in a compact manner, allowing the analyst to see how features are distributed across the code files, which parts of the code have high complexity, which external resources are used by which applications, etc. The analyst should be able to navigate through these maps to see the information at various levels of abstraction from the application level down to the relevant individual lines of code.

Another use for the software intent is to drive the transformation of the system. Ideally, one would know exactly where and how to make changes to the system to achieve the project goals. The changes that are made to the system should be driven by program intent, quantitative metrics, and knowledge of the source and target environments. For example, a naïve translation from COBOL to Java typically produces code that is inefficient, bloated, littered with COBOL-isms, and completely unmaintainable. Even though the code is in Java syntax, the code will look unnatural to a Java programmer and therefore will be difficult to maintain. Instead, the reengineering process should translate the intent in the COBOL code so that the resulting Java contains appropriate class hierarchies, uses encapsulation, does not rely on manipulation of a flat COBOL-like data space, and uses appropriate native Java libraries.

This discovery and use of program intent requires human expertise and has traditionally been available only through manual rewriting. For large code bases, this option is prohibitively expensive and error prone. As a better solution, a human expert should have automated tool support act as a knowledgeable assistant and provide the scalability required to handle very large legacy systems. The tool should be able to do both mundane tasks such as computing the code complexity as well as complex tasks such as suggesting places in the code that may relate to a feature and applying a selected transformation across relevant parts of the system.

UniqueSoft’s legacy reengineering tool acts as this knowledgeable assistant. The capabilities of this tool are described in the rest of this paper.

**UniqueSoft Legacy Reengineering Capabilities**

UniqueSoft applies a high degree of automation throughout the legacy modernization process. The flexibility of the UniqueSoft tool enables the customization of the results to the needs of the project instead of forcing the solution to fit the tool. The tool is scalable in terms of the size of the legacy system to be modernized and the variation in languages and environments.

The UniqueSoft reengineering process described here separates reengineering into three phases: Discovery, Transformation, and Integration (See Figure 1 on the next page).
Discovery provides the stakeholders with the information they need about a legacy system in order to make informed transformation decisions. Transformation refactors the code to meet the project goals. Finally, the transformed system is instantiated for the new operating environment, cloud framework, client middleware libraries, language, etc. The UniqueSoft tool suite has at its core a powerful and customizable rules engine built to automate this process.

The UniqueSoft reengineering process and tool is applicable across a spectrum of legacy reengineering projects spanning both embedded and IT systems:

- Transform COBOL (or other languages such as SQL, PL/I, KShell,) to maintainable code in a modern language such as Java without being locked into the impenetrable and inefficient code that uses proprietary libraries produced by other translation tools, or to create code that is easy to understand and maintain from that originally translated code.
- Extract business rules from legacy code (for documentation or for modernizing to a new system) in a way that reflects the project’s actual business processes and terminology.
- Migrate a system to a new technology platform (cloud, JEE, Hadoop, Linux, etc.) while taking full native advantage of the target platform’s capabilities.
- Significantly reduce maintenance costs through modularization, clone elimination, dead code elimination, etc.
- Produce test suites that can be used to demonstrate functional equivalence of the legacy software and the modernized software.
- Visualize and explore the legacy software from multiple perspectives (features, metrics, dependencies, data flow, etc.) in terms of the concepts natural to the stakeholders.
- Create a modernization plan and drive the reengineering process based on quantitative metrics instead of basing the work on guesses as to what should be done.
- Reconcile out-of-date documentation with the true functionality of the legacy code.
- Merge code bases to determine what functionality is duplicated, unique, or modified.
- Translate from or to proprietary languages, middleware layers, or libraries for which no off-the-shelf tools exist.

Figure 1: High-level overview of the UniqueSoft reengineering process
Competitive Comparison
There are multiple ways to approach sustaining and modernizing legacy software systems. The most common options, and the issues they address, are summarized in Table 2, organized from left to right roughly in the order of increasing cost. The issues range from those more typically associated with mainframe software (e.g., obsolete languages) to more universal ones (e.g., costly maintenance). For example, if the only issue is the interoperability with external systems, then wrapping mainframe data APIs can be an option. Hardware virtualization emulates the current environment but does not meet any other sustainment goals. Tools for direct translation of the legacy code exist, but use of these tools typically results in unmaintainable software still containing the idioms and style of the legacy language. The system can also be manually rewritten, but that option is almost always forbiddingly expensive and is the start of a new cycle of legacy software. Table 3 lists additional disadvantages of these options.

Table 2: Comparison of Legacy System Options

<table>
<thead>
<tr>
<th>Legacy Issue</th>
<th>Hardware Virtualization</th>
<th>Scrapping / Wrapping</th>
<th>Rehosting</th>
<th>Direct Translation</th>
<th>UniqueSoft Reengineering</th>
<th>Manual Rewrite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obsolete hardware</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Obsolete language</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Lack of external access (web/mobile)</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Lack of features</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Obsolete architecture</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Obsolete platform</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Lack of documentation</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Lack of test cases</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Costly maintenance</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Lack of Software Assurance</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

Table 3: Legacy System Option Disadvantages

<table>
<thead>
<tr>
<th>Legacy System Option</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Virtualization</td>
<td>The software is not touched at all, so sustainment and modification costs are not reduced.</td>
</tr>
<tr>
<td>Scrapping / Wrapping</td>
<td>The legacy system is largely untouched, so sustainment costs increase (original system plus new software). Data is replicated.</td>
</tr>
<tr>
<td>Rehosting</td>
<td>The software is not touched at all, so sustainment and modification costs are not reduced.</td>
</tr>
<tr>
<td>Direct Translation</td>
<td>The generated code is typically unmaintainable and do not use the paradigms of the target language. Additional software licenses are often involved for proprietary runtime environments.</td>
</tr>
<tr>
<td>UniqueSoft Reengineering</td>
<td>Requires some involvement from SMEs.</td>
</tr>
<tr>
<td>Manual Rewrite</td>
<td>Labor intensive and expensive. Defects are introduced, and the new code is now the legacy code (incorrect documentation, poor design choices, etc.).</td>
</tr>
</tbody>
</table>
There are no proprietary runtime libraries or other code required for the code produced using the UniqueSoft tool, and there is no inherent reliance on UniqueSoft services after the reengineering is complete. The client retains 100% ownership of all intellectual property resulting from the reengineering.

**Example Reengineering Projects**
The UniqueSoft reengineering approach described here has been deployed on projects that cover a variety of the legacy reengineering purposes described above.

- Reengineer a set of applications that process trouble tickets. The applications totaled approximately 175,000 lines of code written in C++, Visual Basic, and SQL stored procedures. The applications were modernized to use Java, SQL, and rules using the FICO Blaze Advisor Rule Engine. Business rules were automatically extracted from the legacy code, rule logic was simplified, dead was removed, and replicated code segments were consolidated.

- Modernize a financial application from POJO Java to the EMC vFabric framework, including the vFabric Web Server, tc Server, Spring, GemFire, RabbitMQ, SQLFire, PostgreSQL, and Hyperic.

- Automatically migrate the Java-based OpenAM software (Authentication, Authorization, Entitlement and Federation) to use Amazon Web Services S3 (AWS S3) as a Cloud-enabled persistence mechanism.

- Reengineer the code for an insurance suite from Java that was automatically translated from the original COBOL to maintainable Java using the native Java constructs. This suite of applications consisted of over 1 million lines of code. The code was reengineered by refactoring the code to use object orientation, eliminating code clones, translating COBOL-like constructs to more natural Java equivalents, and various other improvements in the code such as removing the execution framework imposed by the original translation tool.

- Reengineer a cellular GPRS Allocator module from “C-like” C++ to Object-Oriented C++. The legacy code made extensive use of preprocessor directives and used almost no OO features. The original code was approximately 60,000 lines of C++. The code was refactored to remove dead code; remove cloned code; improve modularity, coupling, cohesion, and testability; and fully use native C++ constructs.

- Reengineer a networking component for maintainability and performance. The legacy code was approximately 110,000 lines of C code. The code was refactored to remove dead code; remove cloned code; and improve modularity, coupling, cohesion, and testability. This project was successfully delivered in approximately three months.

- Reengineer the ETL (Extract-Transform-Load) code for a back-office customer management system. The legacy system consists of multiple millions of lines of code written in several languages. The code was refactored to reduce code, optimize the processing time, and target a different ETL solution.

The details of the UniqueSoft reengineering process are described in the next section.
2. The UniqueSoft Reengineering Process

The UniqueSoft reengineering process is made up of three main phases: Discovery, Transformation, and Instantiation. Each phase is comprised of a number of steps tailored to the needs of the client. The steps in the overall reengineering process are shown in Figure 2. An overview of the process is provided in this section, and each phase is described in more detail in subsequent sections.

The Discovery phase is comprised of four main steps. A Domain Model is created that decomposes the system into project-centric features and groups of interacting features (also called a feature model). Model Extraction parses the legacy code and converts it into a model that captures its behavior and structure. Many common languages such as COBOL, Java, C++, C, SQL, PL/I, JCL, Assembler, Visual Basic, UML, and Kshell are currently supported, and new or custom languages can be readily added. Architecture Extraction derives a formal architectural representation of the system from the legacy artifacts. Feature Extraction annotates the derived model according to the features identified in the Domain Model. At the end of the Discovery phase, the intent of the software has been uncovered through both the intrinsic and extrinsic properties of the software.

The Transformation phase is comprised of five main steps. Quality Analysis computes quality and modularity metrics such as feature diffusion, complexity, identification of dead or cloned code, and entity coupling to provide quantitative guidance to the modernization. Diagnosis of the Quality Analysis results produces a plan for the refactoring to ensure that the transformations result in the desired metrics improvements. Refactoring transformations perform these changes to the design and architecture, leading to better modularity and alignment with the operating environment. Validation tests the extracted and refactored model. This sub-process is repeated until the system meets the project goals. Test-Case Generation generates a regression test suite from requirements or from the legacy system itself.

The metrics computed in Quality Analysis have the additional benefit of providing guiding information for the automatic refactoring done during the Transformation phase. For example, identifying modules with high complexity or features with low modularity provides information on specific areas where the UniqueSoft tool can apply its refactoring transformations. That is, the transformation efforts can be applied where they will demonstrably have the greatest effect. At
the end of the Transformation phase, the system has been changed to have the characteristics that meet the goals of the project.

The Instantiation phase creates the code for the target environment from the transformed model. An Architecture Model is built to realize non-functional requirements such as reliability and performance. A Deployment Model captures how that code is then deployed across the components of the target system. Finally, code generation produces the target code. At the end of the Instantiation phase, one has maintainable code that meets the goals of the project and can be deployed natively in the target framework.

This full process can be tailored to meet the needs of each project. For example, detailed code metrics would typically not be produced for a project whose goal is to replace the local file storage in the legacy code with cloud APIs, and creating a new architecture model may not be needed for a project whose goal is to transform business rules from one language to another.

To illustrate the UniqueSoft process and its reengineering tool, screen shots from the steps of a small and relatively simple reengineering project will be shown. The purpose of this project is to translate a video rental application written in COBOL for the mainframe CICS environment to Java that uses the Spring framework. The original application is about 5000 lines of COBOL that uses an IBM 3270 “green screen” GUI and has its interface logic intermingled with its business logic. The project goals are to have an application with a modern look and feel using a browser-based front end, persistent storage through the use of Hibernate, and maintainable Java code free from “COBOL-isms”. To achieve the goals, the business logic will be identified, the code for the user interface will be extracted, the code for the data access will be extracted, and the code will be transformed into Java that uses the expected native interfaces and constructs.

**Discovery**

The main purpose of the Discovery phase is to uncover the intent, features, architecture, design, and structure of the legacy system and represent that information in an internal model to be used by the tool. Several steps are performed as part of Discovery. First, the legacy source code is parsed to create an initial model that captures the behavior and structure of the original system. It is this internal model that is transformed by the rest of the reengineering process. This model is also used as the basis for the software maps that are created, such as shown in Figure 3 and Figure 4.

**Figure 3: Example Software Map Showing Code Structure**

The software map is shown in the upper center of the screen, with the code on the right and various metrics below. The outer circle represents the whole application, with the COBOL Copy Books in the left circle and the application code in the right circles. The third-level circles are the directories, and the smallest circles are the individual files in the application. The user can interactively navigate to the code by clicking within the software map.
The meaningful concepts in the legacy code are also captured as a domain model. What is meaningful to capture is dependent on the goals of the project. For example, if one of the goals of the project is to transform a client-server application into a cloud application, then local file storage and the category of stored information are important features. For an order-entry application, inventory control and customer support levels may be important features. An example domain model is shown in Figure 5.

A small initial domain model is typically built based on analysis of requirement documents, application documentation, test cases, developer interviews, user interviews, etc. The tool helps the domain expert expand the domain model, so a minimal domain model is initially created covering only those features that are known to be of interest. Additional features are added to the domain model once the analyst sees what parts of the code are still left uncovered and what the tool suggests as new features. The distribution of the features across the code base is also visualized as a software map, as is shown in Figure 6.

The tool uses data mining and topic analysis to discover concepts that may be used as features. Artificial intelligence techniques such as clustering and proximity analysis extend the initial model for greater coverage of the concepts. This process is iterated until the domain model is sufficiently detailed and the code is sufficiently covered for the purposes of the project. Examples of the use of clustering to provide high-level information are shown in Figure 7 and Figure 8. (Next Page.)

In addition to understanding the features that are in the system, an analyst will want to identify architectural elements by
understanding the relationship between components and modules in the system. Data flow diagrams, control flow diagrams (see Figure 9), and call sequence diagrams (see Figure 10) are created automatically that express the architectural information. These diagrams rely on sophisticated code analysis techniques such as partial evaluation and abstract interpretation to understand the behavior of the code. Actual execution of the code is not required. Once entry points into the system are identified, the tool can also provide use cases across the system.

Figure 7: Software Map showing Clustering of Identifiers

This software map shows the clustering of different identifiers in the system. The size and positioning of the color-coded circles provide a direct view of the symmetries in the system. For example, the two circles representing the copybooks in the upper left of the diagram indicate that they have a large degree of similarity.

Figure 8: Using a Software Map to Discover Structure in Related Applications

This example shows the scalability of the analysis. The software map on the left shows the relative sizes of a set of related applications spanning over 1500 files and 1 million lines of code. The software map on the right shows the files grouped by applying clustering analysis.

Figure 9: Example Control Flow Diagram

The full Control Flow diagram of a legacy system can be displayed. This diagram is an efficient representation of all possible flows through the system. The diagram is interactive: an analyst can examine all flows related to an entry point and can drill down into varying levels of detail. The relations of the flows to the functions are highlighted in the detailed views. One can also view all points in the flow that read from selected database tables or write to them.

Figure 10: Zoomed View of a Call Graph

A complete call graph can be generated for an application or set of applications. This example shows a portion of the complete call graph. The entry points into the system (that is, the starting points of the call graph) are automatically identified by the tool.
Other representations, such as resource structure matrices, express dependencies between components as well as between input and outputs. The dependencies between external resources and code modules can be explored using software maps, as is shown in Figure 11. Based on the domain model and its mapping to the code, code segments focused on selected concepts (that is, the implementation of features of the system) can be selectively displayed. This allows the analyst to visualize how the features are distributed across the system. If it is appropriate for a project, other types of diagrams can also be displayed. Examples of such diagrams are given below.

- Class and component diagrams present a high-level overview of the data structures and architecture extracted from the legacy code and the allocations of functionality to packages or files (see Figure 12).
- Activity diagrams present application flows, data dependencies, and control dependencies between key components of the legacy code (see Figure 13). For example, the sequence of reads and writes from and to databases in an ETL system allows the detection of anti-patterns and helps in performance optimizations of the legacy code.
- Use case diagrams highlight the identified independent interactions with the legacy system as derived from its entry points.
- Activity diagrams and Use Case Map (UCM) diagrams represent the identified control flow and its abstractions to serve as the starting point of automated test generation for the legacy system.
- File dependencies are shown from the perspective of data types, function calls, and global variables (see Figure 14).
• Design Structure Matrices (DSMs) show the modules of the system on both axes. The value in the cell corresponds to the number of write-read dependencies between modules through shared resources. The DSM can be sorted according to the number of dependencies. The scalability of the DSM representation is enabled by the ability to drill down the dependencies from the top-level modules to low-level dependencies (see Figure 15).

• Sequence diagrams highlight the details of interactions between components of the legacy system or the low-level functionality of a fragment of the legacy system (see Figure 16).

• Call Graphs are computed from all potential entry points.

• Resource utilization charts show how different parts of the system use different external resources such as databases.
Transformation
The main goal of the Transformation phase is to develop a new system that has been improved in some way. What that improvement goal is depends on the individual project, but it could include improved quality attributes (higher modularity, lower complexity, less replicated code), rearchitecting for the cloud, etc. As part of this phase, the quality metrics must be determined both at the beginning of the transformation and then throughout the iterative transformation steps. In this way, it can be verified that the transformations are making the desired improvements. As a whole, metrics improvements typically involves making trade-offs. For example, one may be able to perform additional reductions in code replication at the expense of modularity. The quantitative metrics provide the data needed to make informed decisions.

Having concrete metrics data allows the Transformation phase to be driven by a quantitative, data-centric strategy. The metrics are compared to reference values and to those from previous iterations to identify the main problem areas in the system. There are many types of metrics that can be computed during the quality analysis and diagnosis steps. These metrics provide views of the code base from several perspectives that are relevant to the reengineering project. Which metrics are computed is based on the specifics of each individual project, but they typically include size and complexity metrics (indicators of how hard a module is to understand, test, and maintain – see Error! Reference source not found.), dependency metrics (indicators of how much a module is resilient to external changes (afferent coupling) or how much a change will affect other modules (efferent coupling)), modularity metrics (indicators of the complexity of the implementation and the likelihood of defects in a module), and redundancy metrics (indicators of how hard a module is to maintain).

In addition to the quality metrics, various other detailed information is derived from the model. The UniqueSoft tool visualizes the results obtained from analyzing the legacy code in a variety...
of formats. A number of impact, dependency, and cross-reference views are created based on the specifics of each individual project. Typical views are listed below. For the quantitative metrics data, the raw values are provided in tabular form, and a diagnosis is provided which highlights values that fall outside of target reference values.

- Replicated code metrics along with where the code is replicated and the amount of the replicated code in each instance. The UniqueSoft tool identifies code that is a direct clone as well as code that is a parameterized clone of other code. See Figure for examples of metrics for replicated code.

- Modularity metrics from a variety of perspectives, including the diffusion of features across the code and the cohesion of the files and methods with respect to a given feature.

- Dependency metrics showing the coupling dependencies between functions through read-write and write-write access of data structure fields.

- Dead code metrics along with location and size. Two types of dead code can be detected. First, code that can never be executed under any combination of inputs. Second, code that is dead because the system no longer uses certain features, entry points, or input values. For example, code may be dead because a product variant is no longer supported but all related code was not previously removed.

Based on the improvement plan created during diagnosis of the metrics, the tool applies the refactoring transformations selected from the reusable refactoring transformation toolbox. Additional transformations are also created to handle project-specific changes, and the transformations can be combined using a scripting language. An example refactoring is shown in Figure 19. As the transformations are applied, a history of specific transformations performed on the system is recorded. Any change, down to those made to individual lines of code, can be examined in detail, and any change can be rolled back if desired (see Figure 20).
Examples of types of refactoring transformations available in the toolbox include isolating the functionality implementing identified features using partial evaluation, reducing coupling by separating functionality annotated by different features, reducing diffusion by moving code fragments annotated with the same feature close to each other in the program, encapsulating data and behavior implementing the same feature into separate classes, and building class hierarchy by consolidating common behavior. Feature-based refactoring transformations automatically apply to all the locations in the legacy code annotated with the selected features.

Legacy code often does not have an associated test suite or, if it does, the test suite is neither comprehensive nor updated to reflect changes since the program’s creation. A viable solution to this issue is possible by taking a different approach to test creation. UniqueSoft can use its reengineering tool to extract the execution traces from the legacy code. These traces can then be automatically transformed into test scenarios that cover the functionality of the legacy code. While this technique does not guarantee correspondence to requirements (such as may exist), it has significant advantages over creating new test scenarios by hand. First, a comprehensive set of test scenarios is created. Second, it is easier to verify that the scenarios are correct than it is to create them in the first place. In addition, if the goal of the testing is to show that no functionality has been changed, then no further test development needs to be done.

Based on the available test suite, the validation step in transformation is used to ensure that no undesirable changes have been made to the code. For example, transformations can be automatically applied that reduce the cyclomatic complexity of the code. However, it is possible that the transformed code will have different performance characteristics that may be unacceptable in a performance-critical region of the code. In this case, the changes would be backed out. The same transformations could then be applied to a more restricted region of the code.

This cycle of measure-plan-refactor-test gives a high degree of control of the transformation process and allows the user to tailor the results to the specific needs of a project.

**Instantiation**

The main goal of the Instantiation phase is to generate code for a target system which has different design or technology characteristics (e.g., new language or platform). UniqueSoft’s code generation system will output code that properly uses the target environment and language. This code generation scheme can be used for a wide variety of transformation projects, such as generating optimized code for a system from a design model (e.g., UML to C++), transforming legacy code to a new language (e.g., COBOL to Java), or moving code from one environment or paradigm to another (e.g., client-server Java using local file storage to cloud-based Java using Amazon AWS S3 persistent storage APIs).

The UniqueSoft code generation tools provide a much more powerful framework than one would get with direct translation tools. In direct translation, a language such as COBOL is translated to a new language such as Java by applying a template-based process. That is, every construct (statement, data type, etc.) in the legacy language has a set way of being directly translated into some construct in the new language. For example, a MOVE in COBOL is translated into a very complex copy operation in Java. While this produces syntactically correct target code that has the desired functionality, it has many problems which make the new code almost impossible to maintain. Using direct translation to create Java from COBOL, for example, will create code that does not take advantage of OO features such as class hierarchies, encapsulation, interfaces, exception handling, polymorphism, abstract data types, etc. In addition, the code will have
remnants of the original language that are considered poor programming practice in the new language, such as string comparisons of literal “Y” / “N” values instead of using native Booleans.

In contrast, the UniqueSoft code generation system has powerful and extensible analysis and transformation capabilities. The UniqueSoft system is based on a rules engine that applies correctness-preserving rules that transform not just the language syntax, but also the structure of the program from the legacy language to the new language paradigm. For example, in transforming COBOL to Java, the UniqueSoft code generator can move embedded SQL into Hibernate calls, refactor of fixed-length character strings into Java idioms, refactor fixed-length numeric data to account for differing precisions (e.g., for currencies with different base units), enhance character handling to account for internationalization (e.g., i18N characters), eliminate GOTO statements, refactor fixed-position parameter passing to use standard Java mechanisms, and refactor code to make use of external library calls.

The code generated in this way will be natural to an engineer familiar with the target language. For example, when transforming COBOL legacy code to modernized Java, the native Java types and object hierarchies will be created to reflect characteristics of the domain. In contrast, the “JOBOL” created from typical translation tools emulate COBOL types by creating a representation of the COBOL memory space in Java, which is an unnatural representation that is almost completely unmaintainable.

A code generation system must also take into account the underlying hardware, operating system, middleware layers isolating the application from the operating system, inter-process communication mechanisms, process management mechanisms, security frameworks, logging and monitoring frameworks, etc. When targeting a web-based system, the target platform would also contain the app server, databases, persistence layer, data access layer, network communication, messaging, and server and client presentation technologies. Usually, there are a number of choices along each dimension, and often more than one mechanism is required for a given target platform. Other model-based engineering environments typically provide either library calls that connect the application to the underlying platform or predefined platform targets (usually for a vanilla operating system), possibly allowing a user to customize the targeting by providing expansions for predefined macros.

The UniqueSoft transformation system generates code for a target platform using technology and topology mappings. Technology mappings generate code based on specific implementation technologies, while the topology mappings generate code for specific configurations of those implementation technologies. Examples of these types of mappings are shown in Figure 21 and Figure 22. (See next page)
Since the mappings are applied automatically by the UniqueSoft code generator, one is not locked into a specific choice of target platform. The code generator automatically creates the corresponding integration code, deployment descriptors, and build artifacts. The code generator can be specialized as required for custom platforms.
3. **Scalability**

Legacy reengineering and modernization projects tend to be large, often comprised of many millions of lines of code. In addition, the code tends to be in a variety of programming languages or dialects, including the many variations of COBOL, scripting languages, database query languages embedded in different ways in the different languages, and processor-specific assembly languages. These languages often exhibit vendor-specific variations and inconsistencies (e.g., Microfocus vs. IBM PL/I, or Basic SQL vs. Oracle PL/SQL vs. Sybase SQL/isql vs. SQLPlus).

UniqueSoft's tool has been designed with scalability in mind, both with respect to the amount of code to be analyzed and manipulated as well as with respect to the variation in source and/or target languages to be manipulated. The analysis algorithms and refactoring transformations extract the information about the analyzed program into two aspects: global information and local information. Global information affects all parts of the system examined, while local information affects only an individual component of the system. In general, a system is structured into a number of components, and only global information needs to be shared about all components. Even for very large systems, individual components are of limited size. UniqueSoft's tool is structured to allow systems to be analyzed and transformed component by component, typically in a number of iterations, such that each individual component together with the global information about the system can easily be held in memory on modern computer systems. Updating and propagating the global information incrementally from one component to another is efficient. In this manner, the UniqueSoft tool is not limited by the size of the system being analyzed or transformed and therefore can be applied to systems consisting of >100 million lines of code.

Scalability with respect to large systems also applies to the visualization of the system and the results of the analysis performed. It is impossible for users to stare at tables with several thousand rows and columns and see meaningful information. For large systems, it is essential that the results are presented in a manner that is understandable by the user. UniqueSoft has strived to make data visualization scalable by presenting results in an interactive manner that allows users to drill down into the information along various dimensions while maintaining the context of the analysis so that the user will not get lost in the details.

4. **Summary**

The UniqueSoft reengineering process and tool makes it possible to transform a legacy software system into a modern asset. Advanced automation and a powerful transformation system enable the advantages of a manual rewrite at a fraction of the cost while avoiding the pitfalls of one-size-fits-all translation tools. Reengineering can be applied across a wide variety of domains to accomplish project goals, including migration of legacy system, correction of software problems, and extraction of information such as business rules from legacy code. These techniques and tool readily scale to massive projects comprised of many millions of lines of legacy code. The analysis capabilities provide visibility into the legacy system and allow for a low-risk reengineering plan to be created based on quantitative metrics.