1. Overview

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

UniqueSoft has developed the D*Code suite of customizable and modular software analysis and transformation tools that provide an integrated solution to software modernization. D*Code acts as an expert assistant to an analyst or architect engaged in the modernization project. While some capabilities of D*Code such as performing dependency analysis are fully automated, other capabilities such as business rule extraction are semi-automated and are accomplished through interaction with the user.

D*Code is built on a powerful framework for performing code analysis and transformation. D*Code currently supports several languages and source/deployment environments, and one of its strengths is its ability to be customized to a particular project’s needs (input and output languages, new platforms, new code transformations, etc.) with a relatively small amount of lead time.

This paper focuses on using D*Code to transform COBOL code to optimized and maintainable Java. The following sections discuss some of the practical issues and limitations and how D*Code was used to overcome these challenges. Examples of the challenges from a large project are provided.

2. Converting COBOL to Java

When converting COBOL to Java, one must consider more than just the syntax of the languages. For example, it is relatively easy to convert a decision construct in COBOL into one in Java. What is much more difficult is accounting for the other factors that are part of a full system migration: system services that are used, user interface, databases, files, dependencies, coding standards, native data types, compiler options, performance, etc. Using a tool that produces a literal translation that is a maintenance nightmare is typically of limited help in a real project.

Converting a system from one language to another requires an understanding of what code constructs were used because of limitations or features of the language itself or because of the underlying platform. The code must then be translated and migrated to use native language constructs and platform features. The rules engine in D*Code does the translation based on the concepts of the programming languages, not simply line by line. For example, uses in COBOL of ‘T’ and ‘F’ string data can usually be converted to native Boolean primitives in Java. This use of the native idioms of the target language makes the resulting code much more maintainable.

Most COBOL code to be translated to Java is non-trivial and is important to the business. These applications typically have been developed, enhanced, and optimized over many years, and non-standard or customized features have often been incorporated. Standard COTS translation tools cannot take these special features into account, and they cannot produce code that meets the specific needs of the organization. The remainder of this white paper discusses how D*Code can solve these problems by providing a case study of a migration project that used D*Code to convert COBOL to Java where other tools and manual efforts had failed.
3. **COBOL to Java Case Study**

This section summarizes challenges encountered and solutions employed when migrating a large COBOL application operating on a mainframe to a Java-based application operating on a Linux server cluster. The application is the core billing application of a consumer and enterprise-facing fortune 100 company. It comprised roughly 4 million lines of COBOL code, interfaced with a DB2 database and many VSAM files, and was memory limited and performance critical. The application takes input from a variety of data sources upstream and sends output files downstream. Over 4000 JCL programs invoke the various COBOL modules and sorts (e.g., SYNCHSORT) and are scheduled by Control-M. The application has been in operation for over 30 years and is still under active development.

The target system is a Linux-based server cluster with a standard Java environment. The migrated application is comprised of Java programs translated from the original COBOL code, interfacing to the DB2 database and the same data sources upstream, and feeding the same downstream data consumers. The Java programs are invoked from shell scripts and Python programs. Sorting is performed by standard Linux routines.

The application was migrated from COBOL to Java using UniqueSoft’s D*Code COBOL2Java translator. D*Code performed not only the translation from COBOL to Java but also addressed several other critical issues, as discussed below. Without having been able to solve each of the following issues, the migration of this system would not have been successful.

**Full Translation from COBOL to Java**

Because this is the billing application for the client, the most fundamental requirement of this migration was the correct translation of the original application programs from COBOL to Java. Given the size and age of the application, it leveraged nearly all constructs of COBOL except for Objects. The application relied heavily on pointers, primarily for performance reasons, but also to allow deployment in a variety of environments. As is typical for large COBOL applications, redefinitions were used extensively: some core data entities had 80 to 100 redefinitions. Redefinitions were also often nested several layers deep. Because the application received multi-gigabyte input files with very large records, efficient and convenient handling of such records was critical. COBOL provides many features to simplify working with such records, and all these were widely leveraged.

The D*Code translation from COBOL to Java had to correctly capture all the subtle semantics of the COBOL language as well as account for the various compiler options that were used in the original COBOL deployment.

**Creation of Maintainable Java Code**

While application correctness was the primary concern, the plans to actively maintain the Java code put a large emphasis on the style of the resulting Java. The Java needed to be as close as possible to code that a Java programmer might have written by hand. If the code were not understandable to the average Java programmer, it is unlikely that the program could be successfully enhanced and maintained in the future.

The result of this requirement is that, wherever possible, the translated program must rely on native Java control constructs and datatypes and must use Java libraries familiar to the typical Java programmer. For example, the translated program uses enumerations and Boolean types for the 88-level entities in the COBOL code. D*Code automatically analyzes the original COBOL code to infer when and how such translations can be made.

This code analysis is very intricate because the memory model in COBOL is fundamentally different from the one used in Java, and there are operations in COBOL that do not have a natural counterpart in Java. In such situations, the abstractions introduced to realize the COBOL behavior were straightforward to use and understand. For example, where efficient conversions between data items of incompatible types are required, D*Code provides an emulation of the COBOL memory model using an API which is familiar to a Java programmer.

In addition, the client in this engagement has organization-wide coding standards that their Java application programmers are expected to follow. These rules range from widely used formatting and coding conventions to specific limitations on
the number of parameters of methods. The translated code was expected to adhere to the same standards even if the original COBOL code did not have the equivalent structure. D*Code was customized to comply with these coding standards, which were enforced by regular scanning of the Java code with Sonar.

Preservation of a Custom Architecture Layers
The COBOL application isolated the business logic into a separate architectural layer which accessed environment services (including files, databases and other programs) through a well-defined middleware layer. This architecture had been developed with the goal that the code be maintainable by programmers familiar with the business domain but less so with the complexities of embedding the application into the client’s environment. This middleware layer also provided abstractions that allowed much flexibility in the environment. For example, it was transparent to the application programmer whether the application was using IMS or a sequential database such as DB2. This middleware layer was implemented in COBOL by intricate inclusion of copybooks with elaborate replacement schemes. The COBOL files were also preprocessed to expand application-layer concepts.

It was important that the translated Java code preserve this isolation of the application layer. D*Code was customized to treat the “contracts” that provided the interface between the application layer and the middleware as extensions to COBOL and thus preserve this interface in the resultant Java code.

Use of Direct Access Memory and Overflow Memory
The COBOL application has very high physical memory requirements, with tens of gigabytes of data periodically kept in memory and operated on in a database-like fashion. This amount of data exceeds the physical memory available on the mainframe and therefore must be offloaded into overflow memory. Conceptually, an in-memory database is provided by the middleware through a combination of complex COBOL pointers and Assembly Language programs.

A literal translation of the direct access and overflow memory into Java would have had performance and maintenance issues, so an in-memory database was developed which met the requirements of the direct access and overflow memory as inferred from the source code and documentation. D*Code was customized so that the application was translated to access this in-memory database wherever the COBOL programs accessed direct access and overflow memory.

Meeting Performance Requirements
The application operates on the very large datasets mentioned above yet needs to complete its operation in a set amount of time so that the overall billing process is not late. This requirement imposes strict performance constraints on the application.

To satisfy these performance constraints, the translated Java code was profiled, and D*Code was customized to apply extensive optimizations. In addition, for the implementation of the in-memory database, D*Code was customized to generate data structures which were optimized for the access characteristics of individual time-intensive programs. Different implementations of key data structures of the in-memory database were provided depending on the access characteristics of programs using it.

Integration into the Target Environment
The generated code had to seamlessly integrate into the client-determined target environment, including using the selected database, connecting to upstream data sources and downstream consumers, conforming to scripts and other programs that invoke the application, and utilizing native sorting routines. The customization of the generated code to integrate with the target environment was accomplished through customization of D*Code.

Adaptation to the target environment also included handling differences between the DB2 database interface on the mainframe and on Linux. Another type of adaption was related to file input/output behavior. Linux defines the behavior of
file operations differently than the mainframe does. Custom file input/output behavior was realized in D*Code so that the Java-based application matched its mainframe counterpart, especially in error situations.

Migration of VSAM Files

As is typical for mainframe programs, the COBOL application relied on VSAM files to provide a light-weight mechanism for storing data records that allow quick, keyed access to both records and fields. This facility was realized within the Linux environment by leveraging a light-weight database, SQLite. A database schema was inferred from the record definitions of the data stored in VSAM files, and the file operations were migrated to database operations in the Java code. This also involved the automatic conversion by D*Code of algorithms that relied on looping over the VSAM files for retrieving or inserting data to algorithms that rely on database-oriented lookup.

Support for Custom Data Encoding

Data on the mainframe is typically encoded using EBCDIC, while data in a Linux environment is encoded using ASCII. The original application relied heavily on packed numeric data and the aggregation of various numeric fields into larger fields. These larger fields were used as keys into a database as well as for other purposes. These uses would not have the same results if all data were to be reencoded to the native ASCII data encoding. In addition, the use of pointers and the storage of pointers as ordinary data make conversion even more difficult.

The customer chose to encode data in Linux in a mixture of EBCDIC and ASCII encodings, where straightforward textual fields were encoded in ASCII format while packed numeric fields remained in EBCDIC format. D*Code was customized to translate data differentially depending on its format, to operate on data in different encoding formats, and to maintain the proper encoding.

Support for Error Handling and Logging

The mainframe application made extensive use of ABEND routines to provide information about failure modes the application might have encountered. A mechanism was chosen to provide a convenient Java exception handling mechanism which matched the behavior defined for the mainframe application. The uses of ABEND, and routines leading to ABEND, intended to process errors were translated to methods leveraging exception handling that captured the necessary information and provided appropriate diagnostics when error situations were encountered.

Similarly, flexible logging mechanisms had to be created for monitoring the running application. Part of this logging was defined by the mainframe application, but much more extensive logging information was required to validate that the migrated program behaved correctly. A logging facility was defined based on Log4J that allowed the users to specify the events to be logged and to keep logging minimal or completely disabled during production so as not to incur unnecessary overhead.

Migration Support

Extensive information was required during migration to debug the application when discrepancies between the mainframe behavior and the behavior on Linux were observed. These discrepancies were due to a wide variety of reasons, ranging from translation errors to errors in the data files to errors in the original COBOL code that were uncovered during migration. Similarly, performance tuning required extensive information to determine the sources of performance bottlenecks and areas with speedup potential.

D*Code was customized to automatically instrument the generated code, and this ability was leveraged heavily in this migration. Being able to rapidly pinpoint bottlenecks and sources of discrepancies was essential to keep debugging time to a minimum.
4. **Summary**

D*Code was used to successfully migrate a large, extremely complex, performance-critical application from COBOL on a mainframe to Java on Linux servers. By using D*Code’s extensive analysis capabilities and by customizing D*Code to produce Java that was specific to the needs of this project, all requirements of the project were met in a cost-effective manner.

The UniqueSoft D*Code modernization tool suite makes it possible to transform large and complex legacy systems into modern, maintainable assets. D*Code’s powerful and customizable transformation capabilities enable these projects to succeed where other tools provide either a partial solution or no solution at all.