advanced
People. Partnership. Solutions.

Automated COBOL Refactoring

Our automated COBOL refactoring philosophy is centered on producing functionally equivalent, human-maintainable, quality-driven code positioned on Cloud ready target frameworks and platforms. COBOL-to-Universal (CTU), our proprietary toolset, powers the automated refactoring process.

Translating a procedural language such as COBOL, to an object-oriented language such as Java, presents a myriad of challenges. Thus, CTU was designed around these absolutes:

- The refactored application must behave exactly the same as the original application and produce the same results
- The refactored application must be human-maintainable and follow object-oriented concepts and paradigms
- The refactored application must perform as well or better than the original application in the target modern environment
- The resulting applications should be Cloud-ready and delivered through an automated code pipeline using a standard DevOps toolchain and best practices

CTU deconstructs the legacy codebase to isolate source code and classify down to the field level. It then refactors the assets using customized rules based on customer requirements and standards. This refactoring process includes flow normalization, code restructuring, data layer extraction, data remodeling, and packaging for reconstruction. Upon reconstruction, new object-oriented code structures are generated and deployed into the target environment and enabled by CTU's native Java framework.

CTU's Native Java Framework

CTU-refactored applications are underpinned by a Java framework containing a collection of libraries supporting common functions (application lifecycle, marshalling, data I/O, logging, MVS utility replacement, support functions, etc.) that exist in the legacy system but are not directly translatable to a modern Java environment. The framework effectively gathers all common functionality into a single place, significantly reducing the amount of generated code while improving the separation of concepts.

CTU's Java Framework is a collection of libraries without native components, designed to work with both standard Java Edition and Enterprise Edition platforms. It is retained by the customer, in their modernized environment, and can be

Automated COBOL Refactoring

used as a Maven repository or optionally licensed with the source code, allowing the client to enhance and extend as necessary to meet ongoing modernization requirements.

The Framework provides support for emulating key COBOL and JCL functionality such as the translation of COBOL functions to equivalent Java functions, support for sequential files, VSAM to relational modelling, and Online Transaction Processing (OLTP) support (e.g. CICS API, TD Queues, TS Queues, TP session, etc.). In addition, the Framework handles configuration settings, data types not supported in Java, marshalling of read/writes for data types, EBDIC to ASCII encoding, and sessions to retain application context on a per thread basis.

The Framework also leverages the Spring Batch runtime, a light-weight open source framework used for supporting batch processing (symbolic replacement, COND evaluation, return code handling, GDG support, SYSOUT redirection, utilities, etc.).

High-Level Architecture

Human maintainability is a complex problem for automation to solve, particularly when taking a procedural codebase to an object-oriented one. Java developers work in a universe of indirection, where applications reference databases without knowledge of the structure or contents therein. Conversely, there is no layering or indirection with COBOL applications running on mainframes. To accommodate for this architectural difference, CTU refactors the source into a three-tier application.

Put simply, each COBOL program (online and batch) is refactored to a Java class with equivalent functionality and exposed as a service (business logic layer), which interacts with the data access layer through the CTU framework. All database and file operations from the original COBOL code are externalized in data access object (DAO) classes and Java Persistence Architecture (JPA) mapping files (the data access layer). The presentation layer is generated by delivering the legacy screens as HTML pages in a web browser using Angular, and batch JCL is refactored to JSR-352 XML, which interacts with a distributed job scheduler of the customer's choice.

These layers can be implemented using a variety of products to accommodate for specific customer architectural requirements while providing the robustness necessary for high volume, high-uptime applications.

The resulting refactored code is readable and maintainable, following Java standards and object-oriented concepts. The COBOL business logic and comments are preserved, and individuals familiar with the original application can easily understand the refactored code.

Basic Mapping

CTU wields a strong mapping component that produces an object-oriented model that best represents the procedural source. The code produced by this mapping component can be further refined after the modernization project to optimize fit and function with the overall framework and architecture of the target environment.

The table below illustrates the basic mapping from procedural COBOL to object-oriented Java.

Automated COBOL Refactoring

Existing Type	Target Type
Program	Java class
Paragraph	Java Method
BMS Maps / MFS Screens	Browser based (HTML5) applications via various supported presentation models, e.g. JSF, Angular
File	Data Access Object [DAO]
File Record Definition	Data Transfer Object [DTO]
Table	DAO, DTO
Copybook	Data Classes
Working Storage Data fields Group items 88 levels	Class properties Class methods Boolean variables or classes
CICS or IMS TM statements	Screen constructs transformed to Web client calls System/non screen calls replaced by call to MDSY Java framework
JCL Jobs and Procs	JSR-325 XML "jobs" supported by the Spring Batch Runtime and MDSY Java Framework

The basic mapping from procedural COBOL to object-oriented Java

Mapping decisions also extend to structural considerations around copybooks such as clone detection and handling. Since CTU deconstructs the source, it effectively eliminates copybooks, aggregating cloned code (code that appears in repetition across the estate) into programs and methods as necessary. The result of automated structural decisions such as clone detection and handling is a smaller target codebase footprint and more maintainable Java.

Ultimately, CTU is designed to optimize mapping decisions to use the objects which best fit the overall architecture of the system while prioritizing performance and maintainability.

Data Access Layer

The logic used to handle files and databases is decoupled from the main COBOL structure using a Data Access Object (DAO) design pattern. By using a standard Java design pattern, CTU ensures the refactored code is easier to understand, maintain, and extend. The DAO assures communication with external data sources (files and databases), while the Data Transfer Objects (DTO) are used to transport data between the program and external data source. Thus, modifications can be made to the DAO implementation without altering decoupled modules of the application.

Java Persistence Architecture (JPA) is used to facilitate multiple database engines, where database DTOs are created from SQL table definitions and all queries in a DAO are externalized to a yml mapping file, further simplifying the maintenance and extension of queries.

Clone detection and handling applies to the data access layer in much the same way that it does in the program structure. If the COBOL code contains more than one SQL statement of the same type (even across different programs), only one Java method will be created inside the DAO. In this instance, the DTOs contain all information related to a specific SQL table, and the DAOs represent a collation of different SQL statements related to the table. In sequential file access scenarios, clones are detected and handled in such a way that the DAOs use framework calls to IFileOperations to emulate the original COBOL, and file DTOs are created from COBOL variables associated with each descriptor.

Automated COBOL Refactoring

Business Logic Layer

During the restructuring of code to create the business logic layer, CTU refactors each COBOL program to a Java class. This class encapsulates all working storage fields and paragraphs, and exposed as a service to the main entry point in the run() method.

Each paragraph becomes a private method in the Java program class where program structure and comments are preserved to simplify maintenance. Program flow is normalized and GOTOs, EXIT paragraphs, and dead code (flow and data) are removed. Normalization replaces GOTOs and fall through behavior with a statically determined code, resulting in minimal change to the program structure, while retaining the original conditional statements and constructs.

Each copybook that contains only data will be translated to a single class which can be reused across the entire application.

COBOL data structures are analyzed and remodeled to maximize encapsulation, reuse, and readability, while minimizing memory footprint for better utilization. Deeply nested data definitions are collapsed to primary data fields. When group access is required, CTU will generate a method capable of reading and writing information to and from these fields.

In the interest of maintainability, all primitive data types are translated to native Java types, with one exception, the decimal type is translated to the CTU Framework class AfDecimal. The following table shows how some COBOL data types are translated to Java native counterparts:

PIC 9 - PIC 9(4) USAGE IS COMP	short
PIC 9(5) - PIC 9(9) USAGE IS COMP	int
PIC 9(10) - PIC 9(18) USAGE IS COMP	long
PIC X	char
PIC X(2) - PIC X(n)	java.lang.String
Numeric-edited	java.lang.String
Alpha-numeric-edited	java.lang.String
9(n)V9(m)	bphx.ctu.lang.datatype.AfDecimal
USAGE IS PACKED-DECIMAL	short/int/long for integer values / bphx.ctu.lang.datatype.AfDecimal for decimal values
USAGE IS COMP-1 / COMP-2	float / double
USAGE IS INDEX	int

How COBOL data types are translated to Java native counterparts

Conditional names (88 level) are a bit more complicated and are either refactored to Java booleans or to classes. Conditional names with multiple values and 88 levels with only two values that do not contain boolean values (Y/N, YES/NO, TRUE/FALSE) are refactored to a set of named constants in an enum-like class. 88 levels with only two values that are represented with boolean values are refactored to Java boolean types.

Presentation Layer

The primary aim of the presentation layer transformation is to maintain functional equivalence. To achieve this, CTU essentially refactors 3270 screens to HTML5 equivalents that look and feel the same as their legacy counterparts. Screen layouts are maintained, PF and return keys are fully supported using JavaScript event handling routines, and arrow key navigation between fields is retained as-is, removing the burden of re-training end users.

CTU's automated refactoring process takes legacy screens to Angular using fixed-width fonts and absolute positioning of fields and text labels to enable easy deployment on any application server. Although BMS maps are represented as web pages in the refactored environment, the data fed into the new web interface is the same as that which fed the legacy screen. While the refactored screens retain the same screen layout and functionality, these web pages can be re-structured to take advantage of modern web design (HTML5, JavaScript, CSS) for a more modern user experience.

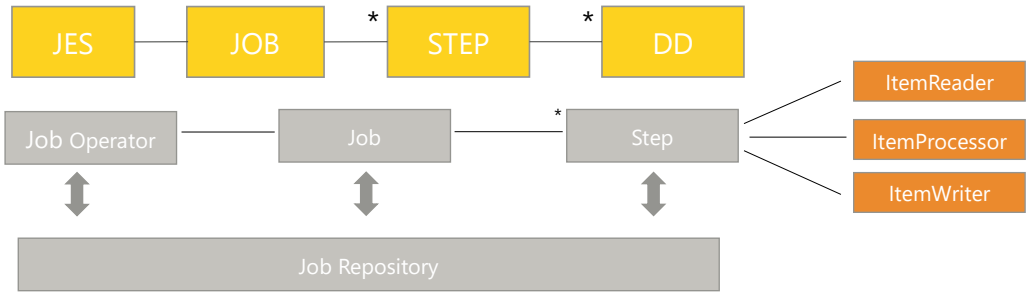
Automated COBOL Refactoring

When it comes to security, a basic authentication mechanism is provided out of the box. While the default authentication module is customizable, an API is also provided to accommodate all authentication requirements.

JCL Handling

JCL is handled with priority placed on functional equivalence and human maintainability in much the same way as its COBOL counterparts. Jobs and procedures (PROCs) are refactored to JSR-352 XML and supported using Spring Batch along with replacements for common utilities (IKJEFT01, IDCAMS, IEBGENR, IEBCOPY, SORT, etc.) so that the jobs and procs execute the refactored code and access the new relational database. Maintainability is maximized by refactoring the JCL during the automated refactoring process to accommodate for differences in the form and function of components in the target modernized environment. For example, many dependencies that are required for pre-relational database models can be safely removed because they are no longer necessary when referencing the newly migrated database, which is relational. Certain file handling steps can also be eliminated. If an organization uses VSAM files, steps will often be placed in the JCL to backup those files to mitigate the impact of failure or corruption. However, modern relational databases enact these steps automatically, rendering the backup steps in the JCL unnecessary.

JSR 352 introduces a Java specification for building, deploying, and running batch applications and addresses three critical concepts: a batch programming model, a job specification language, and a batch runtime. JSR 352 provides developers with clear, reusable interfaces for construction batch, it provides job writers with an expression language for how to execute the steps of a batch execution, and it exposes a runtime API for initiating and controlling batch execution.



JCL	JSR-352
//JOB JNAME (Job)	Job (typically JNAMEJob)
//PROC PNAME (Procedure)	Job (PNAME)
//... EXEC (Step)	Step
//DD ..	?

JCL translations: Similarities between JCL and JSR-352

WJSR 352 defines a Job Specification Language (JSL) to define batch jobs, a set of interfaces that describes the artifacts that comprise the batch programming model to implement batch business logic, and a batch runtime for running batch jobs according to a defined life cycle.

The batch runtime is a part of the Java EE 7 runtime and has unfettered access to other features of the platform, including persistence, messaging, transaction management, and more.

JSR 352 stops short of job scheduling, as there are a myriad of products available for this task and distributed schedulers are often responsible for jobs across multiple systems, a subset of which could fall outside the scope of the modernization effort. Fortunately, the refactored JSR 352 XML batch enables organizations to easily apply scheduling mechanisms ranging from EJB timers and cron, to enterprise schedulers such as BMC Control-M, IBM Tivoli Workload Scheduler, and others.

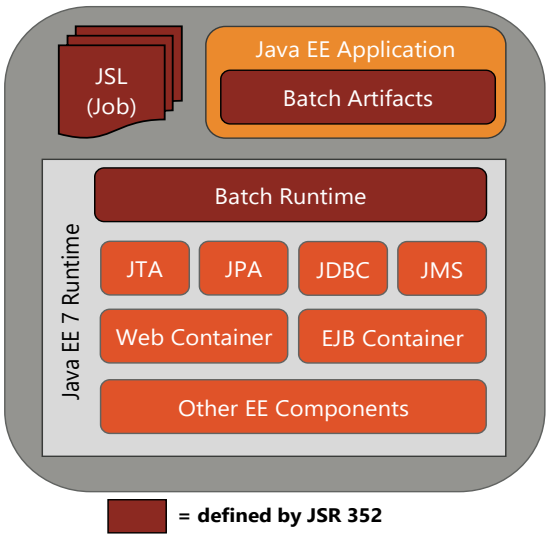
Automated COBOL Refactoring

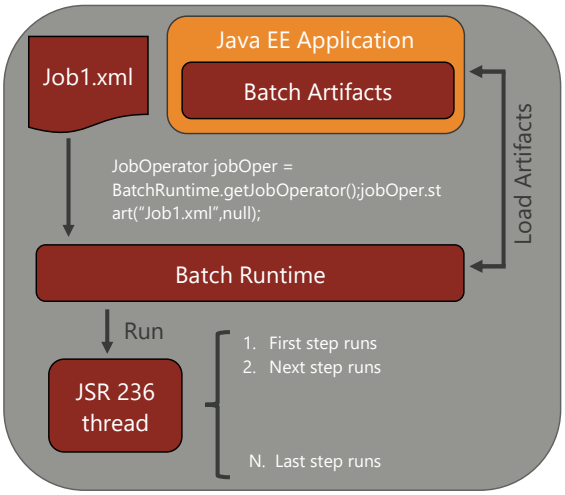
Maintenance, Refinement, and Extensibility

The design and function of CTU and its output is focused on empowering developers by extending native Java concepts and conventions in a readable, maintainable format. Since the general program structure is preserved along with original comments, individuals familiar with the legacy application can easily navigate the new Java code. Developers who are accustomed to modern application practices will experience familiar Java coding conventions and object-oriented concepts such as DAO design patterns, Java native data types, normalized program flow, and application modularization. Maintainability is maximized with clone detection and handling, reusable copybook and data record class definitions, and the utilization of working storage fields as private to the program class and programs as a service, exposing the main entry point.

The Java produced by CTU is cloud-ready and can easily be optimized to maximize the cloud's elastic architecture, harness cloud-native databases, and employ modern application practices. Adherence to native Java concepts and simple integration with standard frameworks and solutions eases the transition towards stateless applications and cloud-native microservices.

To learn more about our approach and multi-phase methodology to approaching modernization projects, take a look at our 'Don't Kick the Can Down a Crumbling Road' whitepaper.





JSR 352 support for batch applications and the Java Platform

More information

w modernsystems.oneadvanced.com **e** hello@oneadvanced.com

UK +44 0333 230 1884

Ditton Park, Riding Court Road Datchet,
Slough, Berkshire, SL3 9LL

US +1 855-905-4040

3200 Windy Hill Road, Suite 230 West,
Atlanta, GA 30339