**EA Java to COBOL Demo Script**

N.B. Prior to doing a demonstration, the workspace should be prepared as per the instructions in the ‘Java Boundary Detection Workflow’ document contained in the Demonstration Package.

| **Script** | Keystrokes |
| --- | --- |
| Modern Enterprise Information Systems typically comprise applications across different platforms – mainframe, distributed, midrange, and open. Insight into the connection points between these applications is vital for a complete understanding of the dependencies across the system.  Let’s start by opening a workspace, which includes both Java and COBOL sources, and ask ourselves how do we go about looking for the connections from Java Classes to COBOL programs? | * Open EA * Select J2C\_Demo workspace |
| Upon entry to Enterprise Analyzer, we see at left a repository browser which shows us all of the sources. The repository is organized into projects, each representing a sub-system we have designated, or – as in the case of this demo project, the entire system. | * Pan cursor to left – repository browser |
| At right, we see graphical views of some key metrics, such as Inventory for a project and a Complexity report to give us some initial information about the application’s size and composition. | * Pan cursor to right – Quick view charts |
| Let’s now have a look at the J2C Demo workspace. As we drill into its artifacts, information about its member sources and their dependencies on other system objects is revealed. | * Repository Browser, click on J2C\_Demo project |
| For example, for the source ORDRENT1, we are able to see that it defines a program called ORDRENT1, calls the INTEDIT and NUMEDIT programs, reads from the customer master and product files, writes into the invoice file, interacts with screens, starts certain transactions, and includes a number of copybook files. Navigating this browser, you are essentially viewing the Enterprise Analyzer meta-model at its basic level, corresponding to OMG’s knowledge domain model. | * Expand COBOL File/ ORDRENT1 and its dependent objects |
| Equally, we can see how Java File GetBusinessProcessName has also been expanded into Classes and Methods following successful verification. | * Expand Java File/GetBusinessProcessName and its dependent objects |
| For the purposes of this example, we know that the customer uses the CICS Transaction Gateway to call COBOL programs, passing the request via the ECI Request Data Area. The customer thinks this starts with the characters “ECI”. |  |
| We will use the Interactive Analysis feature for Java to look for uses of this data area. Note that the Clipper search facility contains pre-defined queries that may be customized according to requirements. In this case we will use the query ‘ECI Variable Identification’ to look for relevant variables. | * Right-click on a Java file and select Interactive Analysis * Select Clipper from ‘View’ Menu (if not already open) * Click on ‘Start Search’ icon, and run ‘ECI Variable Identification’ query * Expand and show results |
| Having identified the variable, we can now tailor the query ‘Method Invocation with ECI Variable’ to find all method invocations that use this variable. Note how selecting a result is reflected in the other open views, e.g. source and context. | * Click on ‘Start Search’ icon, and run ‘Method Invocation with ECI Variable’ query. * Expand and show results * Select ‘jgaconnection.flow(ECIRequest) and then position on the line in Source View |
| By interrogating the properties of this method invocation, we can find the underlying method name. | * Ensure position on ‘flow’ and Right-click. * Select ‘Properties’ |
| We can now configure EA to recognize where a cross-platform call exists by creating a ‘Boundary Decision’. We add the method name to the ‘Program Entry Point’ pane of the Boundary Decision Dialog. We could add a generic pattern using wildcards, but in this case we know the specific method name. For the purposes of this demo, the Boundary decision has already been prepared. | * Tools * Workspace Options * Boundary Decisions * Program Entry Point |
| Boundary Decisions can be generated as part of the Verification Process, or later from the ‘Prepare’ menu on previously verified code. We can see the decisions created by viewing the Decision Resolution pane. We can see the decision we have created for our method, as well as other decisions that EA has automatically resolved. | * Prepare * Decision Resolution * Sort by Type |
| As yet, we have not resolved this decision to a target COBOL program. We do this by checking the box for RTEMPLOY in the lower pane. If we are satisfied that at this point we have completed our analysis, we can set complete on, to indicate that we have now completed all our decisions for a specific method. | * Check RTEMPLOY * Set Complete ON via right-click menu * ‘File’ Menu, and Save. |
| We now have the ability to report on our cross-platform calls. By selecting the Query Repository function, we see that there are specific Java to COBOL queries according to context. For example, we can see the queries available for Java Class, and for COBOL Program. | * Click on ‘Java Class’ * Right-click and Select ‘Query Repository’ * Run ‘Java COBOL Integration’ * Click on ‘Program’ * Right-click and select ‘Query Repository’ * Run ‘Program ECI Dependencies’ |
| We can also add Java to COBOL scopes to the EA diagramming function. We have a basic scope that shows all the steps from Java Method to the called program. However, this can lead to a lot of ‘noise’ as much of the information relates to incomplete COBOL chains. This can be limited by creating a project that only contains classes that make calls to COBOL programs. | * Start Diagrammer * Select ‘JavatoCobolCalls’ Project * Select ‘Java to COBOL (Basic)’ scope, and build the diagram |
| Another way of ensuring clarity is to create a composite relationship, thus ensuring that incomplete relationships are not shown, and leaving only the specific Java Files and the Legacy programs that they invoke. | * Select ‘Java to COBOL (Composite)’ scope, and rebuild the diagram |
| In summary, what we’ve done is to show how we can enable EA to identify how a front-end Java application communicates with a legacy (in this case COBOL) back end, and thus identify the cross-platform legacy calls that can be invoked. |  |