DataConnect SDK Developer Guide

Authors:

Wendy Bunton

Introduction

This guide is intended for developers who wish to use the DataConnect engine in Java programs. It provides a brief overview of the DataConnect SDK, then (using the DataConnect SDK Samples (insert link here)) describes how to accomplish varying tasks.

This guide presumes you are familiar with the DataConnect product and already know how to use DataConnect Studio to design transformations and processes.

Important Notes

- **Engines** The native objects that load/save/execute transformations and processes have traditionally been referred to as "engines". To differentiate them, there is the "map engine" and the "process engine". Starting with V10, these would be further distinguished as V9 or V10. The collective set of artifact engines are known as the engine. We will use the terms "map engine" and "process engine" (adding a version modifier where appropriate), and continue to use "engine" to refer to the entire set of native code.
- **Versions** References to V10, V11 and V12 are specific. References to V9 actually means V9 and earlier, back to V7.5.
- **Map vs Transformation** The transformation is a specific set of instructions regarding the source(s), target(s), and how to map between them. This is implemented via a ".map" artifact which contains the actual mapping information, and a ".map.rtc" runtime configuration which defines the sources and targets. In this document, we try to keep the correct usage of "map" to mean the mapping information/.map artifact and transformation to mean something you can actually execute (the runtime source/target configuration in conjunction with the mapping information).
- **Artifact Versions** All the examples and samples are V10/JSON artifacts. In general dealing with V9 a

History of the Engine

It is important to understand some of the history of the engine and its relation the UIs.

In V9 there were two engines, the map engine and the process engine. Artifacts were stored as XML documents (name.tf.xml as the "runtime config" for transformation and name.map.xml for the actual map, name.ip.xml for a process, and name.ss.xml for a "structured" schema).

The map engine was written in conjunction with the UI, which was written in Visual Basic and used COM to interact with the map engine. In many cases, the easiest way for the UI to manage objects was through the idea of a "current" object. Thus, to access a Field object, the UI would tell the map engine to set the current Field to a given index. It could then get/set Field properties/call Field methods) which would interact with this "current" Field.

The Process Designer was written in C++, which means no interface layer (i.e. COM for the Map Designer) was needed. However, the process engine and process designer were developed at the same time, in conjunction with each other, so some aspects of the process engine interface were designed to make access easier through MFC.

In V10 the artifact format was changed to JSON. The product became service-based, and implemented a repository for artifacts. In the repository, the repository itself kept track of what an artifact was (i.e. map, process, runtime config), thus the artifact names did not include extensions. When exported, artifacts would be given an extension corresponding to their artifact type, i.e. name.process, name.map, name.configuration. New map and process engines were added.

The map engine was a complete rewrite, due to a completely new design. The biggest change was an entirely new event model, along with multiple source/targets, the addition of intermediate targets, etc. Since designing was done without involving the engine, the new map engine was able to read the new map artifact, but could not save, and was written to simply load the transformation and run it. The external interface presented by the new map engine was relatively simple, since it was a "load and go" engine. Due to this, it was relatively simple to enable it to present the same interface for V9/XML maps, just delegating the actual operations to a V9 map engine instance.

The new process engine was written on top of the old one. The major change was the new artifact format, along with a couple new step types, and changing the transformation step to execute V10 transformations instead of v9. The artifact loading capability in the new process engine was written to examine an artifact to determine if it was V9/XML or V10/JSON, and properly load/execute either one.

In V11, the artifact formats (largely) did not change. However, the V10 map and process engines were originally written to only support loading and executing an artifact. Saving both artifact types was implemented in the Design SDK, however engine changes were also needed.

Since the V10 process engine sat on top of the V9 process engine, most of the editing support was already in place. One important change to the artifact format was configuration of the transformation step. In the V10 engine, this consisted of referencing a map artifact, and specifying datasets within the process step. In V11, the step was changed to specify a runtime config instead (and the engine was modified to detect the old step configuration, locate the datasets and build a runtime config, and reference that instead). In addition, the concept of an "alternate transformation step" was introduced, allowing V9 processes to run V10 transformations as well as V9, and allowing V10 processes to run V9 transformations as well as V10.

Since the V10 map engine was an entirely new engine, it contained no support for interactive editing. This was a major rewrite to the V10 map engine, including adding all the methods needed to support editing a transformation. When designing this new interface, a decision had to be made between using the old "current object" model, or a newer model. Ultimately, due largely to developer familiarity with the older model, the decision was made to retain this model in many places.

History of the SDK

In v9, the design time environment used a live engine. The APIs used by the designers were not published. The artifacts produced by the designers were in XML format (referred to variously as XML or V9 artifacts). For customer use (embedding) a COM SDK was offered, along with a Java SDK created by generating Java code from the COM source code. This was found to be impractical, and the code bases were separated. While these published SDKs were able to create/load/edit/save and execute processes and transformations, they were still limited in what could be done.

In version V10, the designers were written in Flash for the UI, and in Java for the corresponding services. The UIs did not, in general, interact with a live engine (a couple of design services did use a live engine to retrieve data type info, connection parts and options info, etc.). Aside from those limited cases, design was done without engine interaction. The services used did contain help text describing the APIs, so in theory could be used by customers. There was also a Client SDK (in Java) to allow both remote and local execution of artifacts. This would require designing and packaging the artifacts through the UI, then either deploying (in the case of remote execution) or exporting/downloading the package (in the case of local execution).

For V11, a return to live interaction with the engine was desired, along with writing the UI in Java. To support this, a new Design SDK was developed. It supported full access to the engine (both V9/XML and V10/JSON map and process engines). The V10 engines were modified to support interactive use. The UI is fully capable of creating/loading/editing/saving and executing either V9 or V10 artifacts (although to create new V9 artifacts required switching to "compatibility mode" as an indicator these were not the preferred formats/versions). The SDK was written both to be the interface used in our Designers, and with the intent of eventually making it available for customer use.

About the SDK

For V12, the Design SDK has been renamed the DataConnect SDK to reflect it being the published as the "official" SDK to use when embedding /interfacing with the engine. It is the same SDK used within Actian. The SDK consists of the following parts:

- **Java side** The Java side of the interface. There is some functionality implemented in Java (macro management, saving of V10/JSON artifacts, etc.)
- **designshim shared library** This shared library is written in C++, and acts as a proxy to the actual JNI code. The source for this library is generated from the actual JNI code. Its intent is to isolate the actual JNI shared library, and all its dependencies, from the Java side. To do this, it dynamically loads the JNI shared library, and as JNI calls are made, looks them up in the actual JNI shared library and passes the call on. This work was done specifically to support using the SDK in an OSGI container.
- **designsdk shared library** This shared library actually implements the JNI side of the SDK. Most of the implementation consists of retrieving the referenced engine objects (i.e. map engine, process engine, etc.) and passing the calls on, or retrieving the desired values.

As noted above, the SDK was originally called the Design SDK. This was to distinguish it from the "Java SDK", and to indicate that it is the SDK used by Actian to produce the designers, and in general interface with the engine. While the name has been changed to DataConnect SDK, due to the major impact on existing code the package names still refer to "com.actian.di.designsdk" (the "designsdk" part is obvious, "di" because the V10 was known as "Data Integrator").

The majority of the work for the SDK is implemented within the map engine itself. Just as the JNI shared library is written in C++ to interface directly with the map engine, other code (such as the "djengine" command-line engine) can interact directly with the map engine (granted, the command line engine doesn't have much interaction, but some editing is actually supported on the command line). The process engine already largely supported editing interaction, so not much change was needed.

Important Concepts

This section discusses some important concepts, and the classes/objects that support them. This is not a technical reference to the classes, see the JavaDocs for that.

ObjectRep

A number of classes/interfaces are named SomethingRep. The "Rep" is a reference to the nature of the SDK in general. Many objects in the SDK are acting essentially as proxies for underlying native (C++) objects. In general, the object referred to is obvious, but sometimes not. A partial listing includes

- **MapRep** This is a reference to a map engine (either V9 or V10). The underlying native object is a MapEngine.
- **ProcessRep** This is a reference to a process engine (either V9 or V10). The underlying native object is a JsonProcessEngine (which may be acting strictly as a V9 process).
- **SourceRep** This is a reference to a source node. It may be part of a transformation, or stand-alone.
- **TargetRep** This is a reference to a target node. It may be part of a transformation, or stand-alone.
- **FieldRep** This is a reference to the "current" field in a source or target node. Technically, this does reference an object (though there is a lot of indirection required to get to that object), but it actually references a control/node which allows it to be positioned to any field in the "current" record for that node (RecordRep).
- **ScriptRep** This is a reference to an object created specifically for the SDK, the provides an EZscript compiler, and an EZscript execution object.

EngineFactory class

This is the first class that will be used in any program using the DataConnect SDK. All new instances of artifacts are created through the EngineFactory. It also helps support licensing and some macro support. Note that EngineFactory follows the singleton pattern, you must use the static EngineFactory. instance() method to retrieve the actual instance.

Licensing

This is not very complicated. Both map and process engines will check that they are licensed before actually executing. This check is against a license key, which by default is "ENG". The `setLicenseKey(String key)` method allows checking a different key.

Macros

Loading a macrodef file is generally done through the EngineFactory. It can either be specified once via the `setMacrodefs(String macrodefs, @Deprecated String[] macrosets)`method, or individually in each artifacts SystemContext. In general, when using this method, specify null for the 'macrosets' parameter. Other macro handling is available (defining "override" macros, getting macro values, etc.) through EngineFactory, but is deprecated in preference to SystemContext.

Artifact Creation

All major artifacts (map engines and process engines, MapRep and ProcessRep) are created through the EngineFactory. In addition, EZScript scripts (ScriptRep) and stand-alone controls (SourceRep and TargetRep) are created here.

SystemContext

In V11 and previous versions, macros were handled on a global basis, and messages (DJmessage EZscript object/URI type) were handled on a thread basis (where multiple threads ended up sharing the same message container, resulting largely in effectively making them global). In V10, both macro and message handling have been moved to a SystemContext object. Whenever an engine (map engine or process engine) is created, it will contain a new SystemContext instance. It is possible to share SystemContext instances (see setSystemContext and getSystemContext methods on various objects), although this may not always be desirable.

Macros

When an engine is created, all of the macrosets (including GLOBAL) contained in the engine factory are copied to the SystemContext of the new engine. This is effectively the same as if the SystemContext loaded the macrodefs file loaded in the engine factory. Of course, if the macrodefs file has not been set in the engine factory, then there is nothing to copy over to the new system context. Not only can the system context be told which macrosets to use, it contains its own macro overrides/local definitions. It can also be told to load a macrodefs file, enabling every transformation and process to use a different macrodefs file if desired.

Messages

Each system context is created with its own message container. This means that by default, messages created in one system context are not visible in any other. However, the SystemContext provides the `shareMessageContainer(SystemContext src)` method which allows sharing a message container between multiple SystemContext instances. Messages and message containers are not automatically cleared before execution. If you try to execute a process or transformation that creates a new message (without trying to find one with the same name first) then you should be sure to clear messages before you execute the artifact.

Schemas

This section discusses the schema artifact document. The *IntermediateTargetAndSchemasSample* sample will discuss building schemas through the *RecordRep* and *FieldRep* objects. This is just a very brief overview of the artifact. You can look in any .schema file to see the bare artifact, or any .map or .map.rtc to see embedded schemas.

The schema artifact contains four sections:

- **root_defs** this is basically a separate type_def, that describes the top level record(s) allowed.
- **type_defs** describes the actual record types that build the allowed record instances.
- **rule_defs** describes various rules, such as recognition rules and validation rules.
- **ui** allows preserving information for use outside the engine itself

The root_defs and type_defs section combined are similar in both concept and construct to an xml schema (XSD).

The type_defs section defines the available record types, and consists of one or more typedef objects. Each typedef contains information about the typedef in general, i.e. name and description. In addition, it contains an array of one or more groups.

Each group consists of a name and a collection of field definitions. This collection is one of four types,

- **sequence** specifies that contained fields (data and child record references) must appear in the order specified according to the min (and max) occurs.
- **choice** specifies that one (and only one) of the contained fields must appear (again according to the min and max occurs).
- **all** specifies that any and all fields can appear either 0 or 1 times (depending min occurs), and in any order.
- **inherited** is not commonly used. It's an indicator that all the record types referenced by the fields in the group derive from a common base class, and that it should be handled as an instance of polymorphism. By convention, the "inherited" type should contain a field referencing an abstract type. This field will be used to determine the name (and namespace) of the field/element actually read or written.

Each field may be a data field (see the Data Types section below), or a record reference, thus allowing building a schema with multiple roots ("multiple record types") or a single root (most common), or a hierarchy of records. This is controlled by the root_defs section, which is basically the root typedef. It is always named "Root Defs" when accessing it through RecordRep. It normally (by default, and expected by most connectors) contains a single 'choice' group named "DocumentRoot" (left over from a long history that won't be discussed here). This group is expected to contain only record references, and defines the top-level record(s).

Data Types

The DataConnect engine has support for a large number of discrete data types (101 as of this writing, excluding those that represent various record and group types, of which there are six). Of these, 12 are generic data types, and the remaining 89 are extensions/specializations of these. For example, the "64-bit VAX floating-point" type is a specialization of the "float" type. In some cases, the specialization is more a convenience anything else (a database connector would rather know the type is "varchar", than have to look at a "text" type, determine it is variable length, and its maximum length is less than a certain value in order to decide it maps to the database's VARCHAR type).

In other cases, the specialization refers to data conversion instead. The "64-bit VAX floating-point" mentioned previously has all the same properties as the "float" type. The routines that read or write the bits need to know that the bit format is not ieee, but the (64-bit) floating point representation used on VAX computers.

Connector Data Types

Each connector can supply a list of the datatypes it recognizes. Each entry in the list will provide metadata about that type, and metadata about that metadata. For example, the *DatatypeInfo* interface has a method *getAlign()* to indicate whether the data is left-aligned, right-aligned, etc. within the field. This bit of metadata may change on an instance by instance basis. There are two methods to get metadata about this particular property: *getAlignMode()* will indicate whether the alignment has any meaning, and if so is it writable or only readable. *getAlignMask()* will return a list of the valid alignment types (i.e. left and right may be allowed, while center, general and decimal may not).

Most of the type properties will provide at least a method to indicate the mode (meaningless, readable, read/write). Many, but not all, will provide some way of determining allowed values. One method is with a getSomethingMask() method, to get a list of valid settings. In some cases, there are properties to get minimum and maximum values (i.e. for the length setting, there are *getMinLength()* and *getMaxLength()* methods).

Connectors return a *DatatypeInfo* object, which implements the *DatatypeProperties* interface. This object implements the size-and-set-index model mentioned above; it has a *size()* method to get the number of items in the underlying list, and a *setIndex()* method to set which object in the underlying list is being examined. This list is all the datatypes supported *by this connector at this time*. There are no connectors that support all the available types; the closest is the Binary connector, but it excludes SQL data types, Interval, and internal (record) datatypes.

In addition, some connectors support *user-defined* datatypes (i.e. most of the database connectors, as most database systems allow defining data types on a server or database level). In these cases, getting the list, then connecting (or more specifically, starting a session), then retrieving the type list again may result in different lists. The connector provides a built-in list of types (which may consist only of one text type), and needs to connect to the server and database to retrieve the specific types for that session.

Field Data Types

Each field (in each record) has a specific data type. It is retrieved through the *FieldRep.getDataType()* method, and returns a *DatatypeRep* object. This object also implements the *DatatypeInfo* interface, but adds setters for each of the datatype properties (not the mode or min/max/mask metadata). This data type will usually correspond with one of the connector data types (though obviously properties will probably differ). The field type may not correspond to a connector data type in the following cases:

1. Loading a transformation that references a connector (source or target) that supports user-defined data types. The field type will not necessarily correspond to a connector data type until a session is established (*startSession()* or *connect()* on the control, or execute the transformation).
2. Assign a schema created with another connector. In this case, the types may not correspond until the control is connected (establishing a session is not enough). The schema contains different data types than the connector supports. Part of the connect process involves mapping the datatype of each field to one that the connector does support.
3. Using *changeConnectionTypeName()*. Along with setting the connection type, connectors allow changing the connection type. Setting the connection type and defining a schema (either through using an external schema, or connecting) and then changing the connection type can result in a type mismatch. This is really the same situation as the previous item, and is resolved the same way, by connecting so the data types will be remapped.

Transformation Event Model

This section describes the basic event model used by the (V10) map engine. The V9 map engine used a completely different model. While V12 (and V11) support both map engine versions, V10 is the preferred version, and the only one described here. The event model consists of three basic parts: the control link, the event, and the action. All events are contained in *EventList* objects, which may be retrieved from a *MapRep* object (transformation events), *ControlInterface* (source or target events), or a *ControlLink*.

EventList

The *EventList* object serves two purposes; it provides information about the available/allowed events (*EventInfo* object), and it contains a list of events (*EventHandler* object). The *EventInfo* object describes a single event type, providing a mapping between its name and integer type code, along with a description. Each event list provides an event info objects for each valid event type *for that event list*. This is important because the list of events allowed as a transformation-level event is different than the events allowed in a control link, which itself may vary depending on the target type.

ControlLink

A *ControlLink* is an object that links a source record to a target record, and contains event. It is created via the *SourceControl.addControlLink()* method, and takes two paths: a relative path to the source record, and an absolute path (pointing to the target control as well as record). For any source record and target record, only one control link is allowed. The ordering of control links within the source governs the order of event handling, thus in addition to adding and deleting control, a method is provided to move a control link with the source.

EventHandler

The *EventHandler* manages the event. It contains an event type, a (possibly empty) condition expression, and "if" action and "else" action lists (the assumption is that at least one of the lists will be non-empty). The event handlers fall, internally, into three types:

- **EA** This is an event/action handler. The condition expression is empty, so it only compreses the event and one or more actions. Any "else" actions are ignored.
- **ECA** This is a conditional (event/condition/action) handler. It contains a condition expression, and the actions ("if" actions) are only executed if the condition expression evaluates to true.
- **ECAA** Finally, the event/condition/action/action handler. It contains a condition expression, an "if" action list that are executed if the condition evaluates true, and an "else" action list if the condition is false.

To the SDK, the above classification is not important, and event handler contains a condition expression, and "if" and "else" action lists. If the condition expression is empty, it is considered to always evaluate to true and so the "if" actions will be executed.

The event handler also provides information about what actions are available via the *EventHandler.getAvailableActions()* method.

EventAction

The *EventAction* object represents an actual action to be performed (i.e. OutputRecord). Most actions have parameters (i.e. for ExeccuteExpression, one obvious parameter is the expression to execute). The action has methods to get/set parameter values. It also has a method to get the action type (*EventAction.getType()*) which returns an *EventActionType* object. This object provides name and description, as well as the ability to retrieve information about the parameters.

Event/Action Execution

Events are triggered throughout the transformation execution process. Before anything else, a TransformationStarted event is fired. After this a SourceStarted event is fired, then the read loop begins, which will cause RecordStarted, GroupStarted, SubtreeStarted, etc. events to be fired. Each of the record-level events will be triggered on each control link whose source record path matches the record just read. Execution proceeds in a mostly linear fashion. For each control link whose source record path matches, the event list is retrieved and scanned. Each event handler with the triggered event type has all of its actions executed, in order.

Loading and Executing Processes and Transformations (LoadAndExecuteSample)

The basic process of loading and executing either a transformation or process is quite simple. Following are examples of transformations and processes.

Load and Execute Transformation

The following code is from the `runTransformation()` method in the `LoadAndExecuteSample` sample. Note that line numbers are strictly for reference, and parts of the ConversionEventListener definition have been removed for clarity.

Load and execute a transformation

```
    private boolean runTransformation() {
        MapRep map = null;
        MapRep dup = null;
        String mapname = suppliedArtifactPath("AccountsExample.map.rtc");
        try {
            map = EngineFactory.instance().createMap(V10);
            if (!map.load(mapname)) {
                logger.severe("There was a problem loading the transformation");
                return false;
            }
            logger.info("Running " + mapname);
            dup = map.duplicate();
            dup.execute(new ConversionEventListener() {
                @Override
                public boolean progress(long recordNumber, long estCount, long readCount, long writeCount) {
                    logger.info("Progress: Record " + recordNumber + " of (estimated) " + estCount + ", " +
readCount + " read, " + writeCount + " written");
                    return true;
                }

                @Override
                public boolean onError(long recordNumber, int errorCode, String message) {
                    logger.severe("Error " + errorCode + " on record " + recordNumber + ": " + message);
                    return false;
                }
            });
        }
        catch (DesignSdkException e) {
            logger.severe("Error executing " + mapname + ": " + e.getLocalizedMessage());
            return false;
```

```
        }
        finally {
            if (map != null)
                map.close();
            if (dup != null)
                dup.close();
        }
        return true;
    }
```

The basic process is:

1. Create a MapRep (line 6) using the EngineFactory.createMap() method. The version (ArtifactVersion.V9 or ArtifactVersion.V10) must be specified.
2. Load the transformation (line 7). Note that the name of the runtime config should be specified, the corresponding .map will be read from the rtc file. This method will throw a DesignSDKException if unable to load the transformation. It will return false if there was a problem loading, and true if the load went okay.
3. Optional: duplicate the transformation (line 12). This step is entirely optional. The duplicate will be an exact copy, and will share the original MapRep's SystemContext.
4. Execute the transformation (line 13). You can provide a ConversionEventListener to receive progress and error notifications (note the example code above will not compile, as the ConversionEventListener has additional methods that must be implemented. See LoadAndExecuteSample.java in the samples for the full code). If you do not with to receive notifications, pass null instead.
5. Close the MapRep(s) (lines 32-35). When done using any created artifact, it is a good idea to close() it. Remember the object references an underlying native object which uses actual resources; the close() notifies the native code the object can be released.

Load and Execute Process

The following code is from the `runProcess()` method in the `LoadAndExecuteSample` sample. Note that line numbers are strictly for reference, and parts of the ProcessEventListener definition have been removed for clarity.

Load and execute a process

```
    private boolean runProcess() {
        ProcessRep process = null;
        String processname = suppliedArtifactPath("SimpleProcessSample.process.rtc");
        try {
            process = EngineFactory.instance().createProcess(V10);
            if (!process.load(processname)) {
                logger.severe("There was a problem loading the process");
                return false;
            }
            process.setForceUnloadMapsAndProcesses(true);
            logger.info("Running " + processname);
            process.execute(new ProcessEventListener() {
                @Override
                public boolean progress(String step, String item, long recordCount, int percentComplete) {
                    if (!step.isEmpty())
                        logger.info("Progress: " + step + ", " + item + ": " + recordCount
                                    + " records, " + percentComplete + "% complete");
                    return true;
                }

                @Override
                public boolean result(String item, String resultStatus, long returnCode, String lastMessage,
                                      long errorCount, long rejectCount, long outputCount) {
                    logger.info("Result: step " + item + " status " + resultStatus);
                    return true;
                }
            });
        }
        catch (DesignSdkException e) {
            logger.severe("Error executing " + processname + ": " + e.getLocalizedMessage());
            return false;
        }
        finally {
            if (process != null)
                process.close();
        }
        return true;
```

```
}
```

The basic process is similar to loading and executing a transformation:

1. Create a ProcessRep (line 5) using the EngineFactory.createProcess() method. The version (ArtifactVersion.V9 or ArtifactVersion.V10) must be specified.
2. Load the process (line 6). Note that the name of the runtime config should be specified, the corresponding .process will be read from the rtc file. This method will throw a DesignSDKException if unable to load the process. It will return false if there was a problem loading, and true if the load went okay.
3. Optional: duplicate the process (not shown). This step is entirely optional. The duplicate will be an exact copy, and will share the original ProcessRep's SystemContext.
4. Optional: force reloading transformation and subprocess steps (line 10). By default, transformations and subprocesses are loaded once. This means any configuration (i.e. macro expansion in connect information) is only done once. If these values need to change during execution of the process, then the unloadMapsAndProcesses flag must be set. The default is for this flag to be off. It can be set to always on in the .ini file, and can be controlled on an instance by instance basis using the setForceUnloadMapsAndProcesses() method.
5. Execute the process (line 12). You can provide a ProcessEventListener to receive progress and error notifications (note the example code above will not compile, as the ProcessEventListener has additional methods that must be implemented. See LoadAndExecuteSample.java in the samples for the full code). If you do not with to receive notifications, pass null instead.
6. Close the ProcessRep (lines 34-35). When done using any created artifact, it is a good idea to close() it. Remember the object references an underlying native object which uses actual resources; the close() notifies the native code the object can be released.

Connector Information (PrintConnectorInformationSample)

This sample shows how to retrieve various pieces of information provided by each connector. It sets up the connector/connection, then displays (on the console) a list of all the source connector types that refer to delimited file types, some of the capabilities, each supported datatypes, then the connection parts and options.

Connector setup and flow

The *run()* entry point does two things: it sets up the connector, and then calls methods to display the various pieces of information. Connector setup involves the following (note that all of the operations can throw a DesignSDKException):

Connector Setup

```
ControlInterface ci = EngineFactory.instance().createSource(V10);
ci.setConnectionTypeName("ASCII (Delimited)");
ci.setConnectionPart("file", "$(SRC)/Accounts.txt");
ci.setOption("header", "true");        // First record is a header
ci.startSession();
```

1. The EngineFactory method *createSource()* is used to create a stand-alone source connector (one not associated with a MapRep). There is a corresponding *createTarget()* method to create a target connector. At this point, the connector does not have a type, so no useful information can be obtained. Since the ControlInterface instance actually refers to (and accesses) a connector in the native engine, many operations will result in an exception reporting "Connector type unknown or not set". At this point, about all the can reasonably be done with the control is set the connector type, or get a list of the valid connector types.
2. Here we set the connector type to delimited ASCII. Once the connector type is set, any of the other operations called in the sample are valid (though see line 5 below).
3. Set the "file" connection part, so when we display the parts it will show a value.
4. Similarly, set the "header" property to the non-default value, so the listing of options will contain one value that differs from the default values.
5. Start a session. This actually doesn't do anything useful in this specific example. It is shown here because it can affect the datatype information. Looking at the output of this sample, you will notice the capability indicating user-defined datatypes is false. If this connector did support user-defined types, then a session must be started to retrieve the actual list of types. To see this, set the connector type to any of the databases (i.e. SQL Server 2019). Instead of setting the "file", set the server, database, username and password parts, then display the data types. After this, call *startSession()* and display the datatypes again. You will see a completely different list.

The rest of the *run()* method simply calls other methods to display the various bits of information.

Display information

```
showConnectors(ci);
showCapabilities(ci);
showDatatypes(ci);
showParts(ci);
showOptions(ci);
```

The following sections will discuss each of the methods called above. Each of the methods will call a helper method *displaySection(String section)*, which does nothing more than print three dashes followed by the section name passed. There is also a *yesOrNo(boolean b)* method that will return either "Yes" or "No" depending on whether the passed boolean value is true or false.

Show Connectors

The method *showConnectors()* retrieve the list of available connectors. This method requires an instance (rather than being static) because the available connector list will vary depending on the type of the control (the source list is different from the target list, and the connectors available for a SQL lookup will be different than the list for an incore lookup). In this case, the display is a list of available sources that read some type of delimited (CSV) file.

showConnectors method

```
    private void showConnectors(ControlInterface ci) throws DesignSdkException {
        displaySection("Connector Types");
        for (String type : ci.getConnections()) {
            if (type.contains("Delimited"))
                System.out.println("  " + type);
        }
    }
```

The call *ci.getConnections()* returns a list of the available connector types. This method filters the list to only display connectors with the word "Delimited" in the type. The output of this method will be something like:

```
---- Connector Types
ASCII (Delimited)
Delimited Text
Text (Delimited - EDI)
Text (Delimited - EDIFACT)
Unicode (Delimited)
```

Show Capabilities

The *showCapabilities()* method returns a *ConnectionCapabilities* object, with information about the connector. Connector capabilities are meaningless until the connector type has been set. This method displays some of the capabilities, see the javadoc for *ConnectionCapabilities* for all the information available.

showCapabilities method

```
    private void showCapabilities(ControlInterface ci) {
        ConnectionCapabilities caps = ci.getCapabilities();
        displaySection("Capabilities");
        if (caps.mayDefineExternalSchema())
            System.out.println("  May provide schema");
        else if (caps.mustDefineFields())
            System.out.println("  Must provide schema");
        else
            System.out.println("  Schema defined solely by connector");
        System.out.println("  Supports tables: " + yesOrNo(caps.supportsTables()));
        System.out.println("  User defined data types: " + yesOrNo(caps.supportsUserDefinedDataTypes()));
        System.out.print("  Multiple record types: ");
        if (caps.definesMultipleRecordTypes())
            System.out.println("Defines multiple record types");
        else
            System.out.println(caps.supportsMultipleRecordTypes() ? "Supported" : "Not Supported");
        System.out.println("  Hierarchical records: " + yesOrNo(caps.supportsHierarchicalRecords()));
        if (caps.supportsSql()) {
            System.out.println("  SQL Connector: Yes");
            System.out.println("  SQL Listerals: " + yesOrNo(caps.supposrtsSqlLiterals()));
        }
        System.out.println("  Multimode: " + yesOrNo(caps.supportsMultimode()));
    }
```

The *getCapabilities()* method returns a *ConnectionCapabilities* object that describes various aspects of the selected connector. Some of the capabilities shown are:

Schema handling

Schema handling by connectors falls into three categories.

Some connectors require a schema be provided. These connectors have no way of determining record/field boundaries. Examples are the Binary and ASCII (Fixed) connectors. Without a schema, these connectors will decide the entire file is one big field. This is indicated by the *ConnectionCapabilities.mustDefineField()* method. If this method returns true, then a schema must be provided to the connector.

Some connectors can a schema on their own, but providing one is allowed. This can be useful, for example, with ASCII (Delimited). With no schema supplied, it will create one on the fly. If the first 1000 records have 10 fields, then after that the records contain 11 fields, the in-connector schema will change at that point, but the transformation will still only know about 10 fields. Providing a schema containing all 11 fields will simply cause the 11th field to be read as empty for the first 1000 records. It can also be useful when multiple record types are supported; the schema contains the recognition rules that are sometimes used to determine record types. The *ConnectionCapabilities.mayDefineExternalSchema()* method indicates that a user-provided schema is allowed, but not required.

The final category is connectors that provide a schema, and do not allow the user to supply one. Examples of this category would be the various CRM connectors (where entity definitions are fixed), database sources, etc. This is simply indicated when neither of the other schema handling categories is allowed.

Note that the category may depend on many factors, including whether the connector is a source or target. Additionally, these categories are often not indicated (correctly) for targets, as it is somewhat dynamic, depending on output mode and other factors. For example, when the output mode is REPLACE or MULTIMODE an external schema is required. In APPEND (and related modes) a schema is required if the table does not exist, but not allowed if the table already exists.

Multiple record types

Multiple record types is another somewhat complex capability.

First of all, a connector may be hierarchical (for example, XML-DMS, JSON, all the v10 EDI connectors). These connectors do support multiple record types, but there is also a parent/child relationship, and the record type needs to be defined by a path as well as a typedef. This is indicated by the *ConnectionCapabilities.supportsHierarchicalRecords()* method.

For non-hierarchical support of multiple record types, there are again three categories. The connector may define multiple record types (*ConnectionCapabilities.definesMultipleRecordTypes()*). This means the connector does not need any external support to provide multiple record types. Many of the CRM connectors fall into this category, where returning a sequence of records may involve one parent record, then multiple "child" records. Another example (though capabilities will indicate otherwise) is the ASCII (Delimited) (and Unicode (Delimited)) connector when the **Field1IsRecTypeId** option is set to true.

When the connector does not define multiple record types, it may still support them, indicated by the *ConnectionCapabilities.supportsMultipleRecordTypes()* method. The ASCII (Delimited) and Binary connectors are examples of this. These connectors can return (source) and output (target) multiple record types, with the aid of recognition rules that tell the connector which field (and record type) to look in to determine the record types, and rules that match the value found with the record type to return. Strictly speaking, in these cases recognition rules are **not** required on the target, but they can be useful to have. In the Design Studio, target data browsing is implemented by creating a source of the same connector type, then copying many of the settings from the target to the source. Part of what is copied is the recognition rules, so the source can determine what type each record is. Without this, browsing would show only one record type.

Show Datatypes

The *showDatatypes()* methods displays each datatype supported by the control, along with some information about the data type. See the javadoc for *DatatypeProperties* to see all of the information available about a datatype.

showDatatypes method

```
    private void showDatatypes(ControlInterface ci) throws DesignSdkException {
        displaySection("Datatypes");
        DatatypeInfo info = ci.getDatatypes();
        System.out.println("  Index  Type            Code  Alias           Generic");
        System.out.println("  _____  __________      ____  __________      __________");
        for (int dt = 0; dt < info.size(); ++dt) {
            info.setIndex(dt);
            System.out.println(String.format("  %5d  %-15.15s  %4d  %-15.15s  %-15.15s",
                                             dt, info.getTypeName(), info.getTypeCode(), info.getAlias(),
                                             info.getGenericTypeName()));
        }
    }
```

The *ci.getDatatypes()* call returns a *DatatypeInfo* object, which implements the *DatatypeProperties* interface. This object also implements a "collection" pattern described in the History of the Engine section, where the object provides a count and a method of setting an index to determine which actual object is being examined. In this case, those methods are *DatatypeInfo.size()* to get the number of types supported, and *DatatypeInfo.setIndex(int idx)* to determine which actual one to examine. This example displays each datatype with its name, internal type code, alias (display name), and the corresponding generic type. See the Data Types concept for more general information about data types.

The output from this method will look something like:

```
---- Datatypes
  Index  Type            Code  Alias           Generic
  _____  __________      ____  __________      __________
      0  64-bit binary     12  64-bit binary    64-bit binary
      1  Boolean            9  Boolean         Boolean
      2  Date               6  Date            Date
      3  DateTime           8  Date/Time       DateTime
      4  Decimal            4  Decimal         Decimal
      5  Float              3  Float           Float
      6  Integer            2  Integer         Integer
      7  Name              10  Name            Name
      8  Text               0  Text            Text
      9  Time               7  Time            Time
```

Show Connection Parts

The *showConnectionParts()* method displays each connection part supported by the connector, and some information about it.

showConnectionParts method

```
    private void showParts(ControlInterface ci) throws DesignSdkException {
        displaySection("Connection Parts");
        System.out.println("  Partname    Prompt               Name        Type        Value");
        System.out.println("  __________  __________           __________  __________  __________");
        for (ConnectInfoEntry info : ci.getConnectionEntries()) {
            System.out.println(String.format("  %-10.10s  %-20.20s  %-10.10s  %-10.10s  %s",
                                             info.getPartName(), info.getPrompt(), info.getName(),
                                             info.getType(), ci.getConnectionPart(info.getPartName())));
        }
    }
```

The *ci.getConnectionEntries()* call returns a list of *ConnectionInfoEntry* objects, one for each connection "part" supplied by the connector. Each part may or may not be required (see *ConnectionInfoEntry.isRequired()*, although in some cases this may actually by dynamic (a few connectors can log in via a token or username/password pair, which changes which parts are required). There is additional information available (not displayed above), such as a suggested order/priority for displaying parts and allowable values. This method displays the following information about each part:

- **partName** is the name to use internally, when getting or setting the part value.
- **prompt** is the suggested text to display for this part.
- **name** this property (and its name) is a holdover from long ago. Historically, connectors had a predefined, fixed set of possible connection parts (i. e. SERVER, DATABASE, FILE, etc.). The introduction of DMS connectors required a more freeform approach, thus the addition of entry names such as OTHER. This is actually a *ConnectionInfoEntry.EntryName* enum (see javadoc for a complete listing of values). This property still supplies useful information (i.e. the *ControlInterface.getTableList()* method is really only useful when the name is TABLE.
- **type** is the type of value to supply. This also gives a hint as to what type of UI control should be used. Some possible values are EDIT (freeform text edit field), ENUM (must pick from a list of allowable values), LIST (may pick from a list of allowable values, but may enter text as well). This is a *ConnectionInfoEntry.EntryType* enum, see the javadoc for a complete list of values.
- **value** is the actual current setting/value for this connection part

The output from this method will look like:

```
---- Connection Parts
  Partname    Prompt               Name        Type        Value
  __________  __________           __________  __________  _________
  file        Source File/URI      FILE        FILELIST    $(SRC)/Accounts.txt
```

Show Connection Options

And finally, the *showOptions()* method displays all of the options and some information about them.

showOptions method

```
    private void showOptions(ControlInterface ci) throws DesignSdkException {
        displaySection("Connector Options");
        System.out.println("  Name                          Alias                         Type
Default Value         Value");
        System.out.println("  __________                    __________                    __________
__________            __________");
        for (OptionInfo info : ci.getOptions()) {
            String value = info.isVisible() ? ci.getOption(info.getName()) : "---";
            value = escapeString(value);
                System.out.println(String.format("  %-30.30s  %-30.30s  %-10.10s  %-20.20s  %s",
```

```
                                                info.getName(), info.getAlias(), info.getType().name(),
                                                escapeString(info.getDefaultValue()), value));
        }
    }
```

The *ci.getOptions()* returns a list of *OptionInfo* objects, one for each option. Options have a lot of additional metadata not shown here, of particular note are allowable values. The *OptionInfo.getAllowableValues()* method will return a list of *OptionInfo.AllowableValue* objects describing the allowed values. Each allowed value contains a name (which is a displayable name for the value, i.e. "CR-LF"), a corresponding value ("\r\n"), and a description. When the type of option is ENUM, the entries in the allowed value list are actually the only values allowed. For other types, it's a convenience for UI use (for most users it's more convenient to select "CR-LF" than to type in "\r\n").

The option information displayed here (again a subset of everything available, see the javadoc) are:

- **name** the internal name of the option. This is the value that should be used when getting/setting option values.
- **alias** this is the desired display name for the option.
- **type** this is an *OptionInfo.OptionType* enum value, and indicates what type of value is expected. Some possible types are TEXT, ENUM, NUMBER, FILE and BOOLEAN. See the javadoc for the complete list.
- **defaultValue** as the name implies, the value that will be used if one is not explicitly set.
- **value** the current value for the option. In the following, note that the header option has a default value of False, but the current value is True. This is because part of the setup was to set this option to True. Note also that both altfldsep and nullind indicate the default value is "" (empty string), one reports a current value of "None" while the other reports "none". This is because when getting the value, a reverse lookup is done in the allowable value list, and if the current value is also an allowable value, its display name is returned. These two options have different mapping in their allowable value lists, thus the difference. You can also see the mapping in the recsep option, where "\r\n" is mapped to "CR-LF".
- **visible** this property isn't actually displayed, but is used when getting the value. If an option is not visible, then while all of its metadata is valid, it's value cannot be set or retrieved. Generally, options are hidden either because they don't currently apply (based on the value of another option), or have been deprecated. In the latter case, they are hidden so they can be set when loading an existing transformation, but won't be visible to the UI.

Here is the output from running this method:

```
---- Connector Options
  Name                          Alias                         Type         Default Value         Value
  _____________                 _____________                 _________    _____________         _____________
  codepage                      CodePage                      ENUM         0                     ANSI
  recsep                        RecordSeparator               TEXT         \r\n                  CR-LF
  fieldcount                    RecordFieldCount              NUMBER       0                     0
  fldsep                        FieldSeparator                TEXT         ,                     ,
  fldsdelim                     FieldStartDelimiter           TEXT         "                     "
  fldedelim                     FieldEndDelimiter             TEXT         "                     "
  header                        Header                        BOOLEAN      False                 True
  altfldsep                     AlternateFieldSeparator       TEXT                               None
  soffset                       StartOffset                   NUMBER       0                     0
  autostyle                     AutomaticStyling              BOOLEAN      False                 False
  stylesamplesize               StyleSampleSize               NUMBER       5000                  5000
  lstrip                        StripLeadingBlanks            BOOLEAN      False                 False
  tstrip                        StripTrailingBlanks           BOOLEAN      False                 False
  field1isrectypeid             Field1IsRecTypeId             BOOLEAN      False                 False
  nullind                       NullIndicator                 TEXT                               none
  emptyfieldsnull               EmptyFieldsNull               BOOLEAN      False                 False
  numericformatnormalization    NumericFormatNormalization    BOOLEAN      False                 False
```

Create a Simple Transformation (CreateSimpleMapSample)

This sample will show how to create a simple transformation. A new MapRep is created, one source and target are added and configured, and target fields are added and mapped to the corresponding source fields. The transformation is then saved and executed. This example does not configure any events. It is simple enough that it meets the requirements to allow adding a default event and output action at execution time (or during validation).

Setup and Execution

The *run()* method is the main entry point for the sample. It will create the transformation, call methods to set up the sources and targets, then save and execute the transformation. Finally, it will verify the output.

run method

```
    protected boolean run() {
        try {
            MapRep map = EngineFactory.instance().createMap(V10);
            // Set the log file
            map.getOptions().getLogSettings().setLogFile("$(LOG)/CreateSimpleMap.log");
            SourceRep src = setupSource(map, ACCOUNTS_SRC);
            TargetRep trg = setupTarget(map, TFILE, "TARGET_1");
```

```
        // Connecting the source should give us a schema, since the delimited ASCII connector does not
        // require a schema be set (see src.getCapabilities)
        copyFields(src, trg, null);
        // Now save it
        map.saveAs(createdArtifactPath(MAPNAME + ".map.rtc"), createdArtifactPath(MAPNAME + ".map"));
        // At this point, since we only have a single source and target record type, and the target is
        // not multimode, we can execute the transformation.  We didn't add any action that would
        // cause a record to be output, but the engine detects that and adds a default event/action
        // that will write a record.
        map.execute(null);

        // Verify
        File tfile = new File(map.getSystemContext().expandMacros(TFILE));
        if (tfile.length() != 44702) {
            logger.severe("Target file should be 44702 bytes, was actually " + tfile.length() + " bytes");
            return false;
        }
    }
    catch (DesignSdkException e) {
        logger.severe("Error running sample: " + e.getLocalizedMessage());
        return false;
    }
    return true;
}
```

The operations performed are as follows:

1. Create the transformation (MapRep) (line 3), using the *EngineFactory.createMap()* method.
2. Set the log file (line 5). The "$(LOG)" macro is set to point to the target/runtime/logs directory.
3. Configure the source and target (line 6 and 7). These are done using method calls (that will be described later), rather than inline, because basically the same setup is done by additional samples that subclass this one.
4. Create a schema in the target as a copy of the source, and map the corresponding fields. This again is done in a method call and will be described later (line 11).
5. Now the transformation is saved (line 13). The *createArtifactPath()* method is a convenience function declared in the *Sample* base class, and simply takes the passed artifact name and prepends a path to it. Note that the names of both the runtime config and map files are supplied when saving. If this were a V9 transformation, the extensions would be ".tf.xml" and ".map.xml" rather than ".map.rtc" and ".map".
6. Execute the transformation (line 18). As noted in the comment, we do not needs to add any events/actions to produce output, as this simple transformation meets the requirements for auto-creating the necessary event and action. The next sample (MapWithEventsAndRejectsSample) will explicitly add events and actions. Note that, unlike the previous sample showing execution we pass a null listener. Errors will still be reported (via exceptions), but there will be no progress reports and no additional information if an error should occur.
7. Verify the output (lines 21-25). This doesn't actually validate the entire output, but it does verify that the output file length is as expected.

Set Up Source

The *setupSource()* method takes a transformation (MapRep), adds a source to it, and configures the source. The source type is set to delimited ASCII, the header option is set to true, and the source file is set to the passed file name. Finally, the source is connected. This will cause the connector to open the source file and read some records to determine the number of fields and their lengths. Since the header option is set to True, the first record will be read as field names.

setupSource method

```
protected SourceRep setupSource(MapRep map, String file) throws DesignSdkException {
    // Add the new source to the transformation
    SourceRep src = map.addSource("SOURCE_1");

    // Set the type and the needed connection parts and options
    src.setConnectionTypeName("ASCII (Delimited)");
    src.setConnectionPart("file", file);
    src.setOption("header", "true");        // First record is a header

    // Connect.  This will make the connector open the file and read the header, and create a schema
    // based on the data found.
    src.connect();

    return src;
}
```

Set Up Target

The *setupTarget()* method is a little different, in that it also takes a control name (note above the source is automatically named "SOURCE_1"). This is because this method will also be used by a sample that uses two targets. The connector type is set to fixed ASCII instead of delimited, just to make things interesting. We also specifically set the record separator, because for this connector the default value is "\r\n" on Windows but "\n" on Linux/AIX. We want to be sure we can properly check the file length.

setupTarget method

```
protected TargetRep setupTarget(MapRep map, String file, String name) throws DesignSdkException {
    TargetRep trg = map.addTarget(name);

    // Configure and connect the target
    trg.setConnectionTypeName("ASCII (Fixed)");
    trg.setConnectionPart("file", file);
    // Note that "CR-LF" is in the allowed value list, so will get translated properly into
    // "\r\n" when being outputl
    trg.setOption("recsep", "CR-LF");

    trg.connect();
    return trg;
}
```

Copy and Map Fields

The final method to cover is *copyFields()*, which duplicates the source schema in the target. This method assumes there is only one source record, and that the target is one that automatically installs a single empty record. This method takes the SourceRep to copy fields from, the TargetRep the fields should be copied to, and a list of fields (names) to copy. This third parameter is not used in this sample (passed as null), but will be used in a future sample to define the fields to be copied.

copyFields method

```
protected void copyFields(SourceRep src, TargetRep trg, List<String> fields) throws DesignSdkException {
    if (fields != null && fields.isEmpty())
        fields = null;
    src.getRecords().setRecordPosition(0);
    FieldRep srcFields = src.getRecords().getFields();
    trg.getRecords().setRecordPosition(0);
    FieldRep trgFields = trg.getRecords().getFields();
    // Copy source fields to target, and set map expression
    for (int i = 0; i < srcFields.fieldCount(); ++i) {
        srcFields.setFieldPosition(i);
        if (fields == null || fields.contains(srcFields.getName())) {
            trgFields.insertAfter();
            trgFields.setFieldPosition(trgFields.fieldCount() - 1);
            trgFields.copy(srcFields, true);
            // Make the length a little longer, to make it more readable to humans
            trgFields.getDataType().setLength(trgFields.getDataType().getLength() + 2);
            trgFields.setExpression("fieldat(\"/SOURCE_1/" + src.getRecords().getName() + "/"
                                    + srcFields.getName() + "\")");
        }
    }
}
```

The steps involved are:

1. Set the source and target record positions to 0 (lines 4 and 6). Remember we are only copying the first source record, and assume the target already has a record (most targets, once connected will contain a record).
2. Loop through all the fields (lines 9 and 10). This is another example of the "collection" pattern where an object provides a count of the "contained" objects (line 9, *FieldRep.fieldCount()*), used to limit the for loop, and a method to set the index of the "current" contained object (line 10, *FieldRep.setRecordPosition()*).
3. Check to see if the field should be copied (line 11). Either the list of field names must be null, or the current field's name is contained in the list.
4. Insert a new field in the target record (line 12). You may insert a new field either before or after the "current" field. If there are no fields in the record, you may insert either before or after. Here we choose after because that is the only way to actually append new fields (position to the last field, then insert after).
5. Set the current field position to the field just inserted (line 13). This simply sets the field position to the number of fields - 1.
6. Copy all the field properties (line 14). The first parameter to *FieldRep.copy()* is the field to copy from (the *FieldRep* object positioned to the field to copy), the second indicates whether the name should copied as well.

7. Next we increase the length of each field by 2 (line 16). This is not strictly necessary, although often you will want to make some changes to the copied field's data type.
8. Finally, set a mapping expression (lines 17 and 18). We are simply building an expression that looks like *fieldat("/SOURCE_1/R1/fieldname")*. No te that this expression assume the source is named "SOURCE_1", ideally we should have also passed the source control name. Also, we should have considered using *RecordRep.getPath()* rather than *RecordRep.getName()* when building the path component. We could also have just used *FieldRep.getFullPath()* rather than trying to build the path ourselves.

Transformation With Two Targets and Added Events (MapWithEventsAndRejectsSample)

This sample is built upon the previous one. It creates two targets, with each mapping a subset (the same subset in this case) of the source fields. Event handling is added so that one output file will contain all accounts with a balance of 500 or greater, while the other contains accounts with a balance less than 500. There are multiple ways to build a transformation to accomplish this. We have chosen to use the Reject action and corresponding RecordRejected event (you could just as well have OutputRecord actions to each target, with each having a conditional expression to select the appropriate records).

Setup and Execution

The *run()* method, as always, is the entry point that controls building, saving and executing the transformation.

run method

```
protected boolean run() {
    List<String> fieldsToCopy = Arrays.asList("Account Number", "Company", "City", "State", "Zip",
                                              "Balance");
    MapRep map = null;
    try {
        map = EngineFactory.instance().createMap(V10);
        // Set the log file
        map.getOptions().getLogSettings().setLogFile("$(LOG)/MapWithEventsAndRejectsSample.log");
        SourceRep src = setupSource(map, ACCOUNTS_SRC);
        TargetRep trg = setupTarget(map, ACCFILE, "TARGET_1");
        TargetRep rej = setupTarget(map, REJFILE, "TARGET_2");
        // We're copying a limited number of fields.
        copyFields(src, trg, fieldsToCopy);
        copyFields(src, rej, fieldsToCopy);

        // Go set up the events
        setupEvents(src, trg, rej);

        // Go ahead and save it, then execute
        map.saveAs(createdArtifactPath(MAPNAME + ".map.rtc"), createdArtifactPath(MAPNAME + ".map"));
        map.execute(null);
    }
    catch (DesignSdkException e) {
        logger.severe("Error running sample: " + e.getLocalizedMessage());
        return false;
    }
    return true;
}
```

This method performs basically the same steps as the previous sample (which this one subclasses). Notable points are:

- Lines 2 and 3 create the list of field names. Both targets will contain the same subset of fields from the original source.
- Line 11 creates the second target.
- Line 13 and 14 copy the selected fields to both targets
- Line 17 contains the other major difference between this and the previous sample. Here we call a method to specifically set up events and actions for the transformation.
- The rest of the sample just saves and executes the transformation, as in the previous example.

Set Up Events and Actions

Please see the Important Concepts section for information regarding the general event model used in transformations.

The *setupEvents()* method is responsible for creating control links, events and actions. We have chosen to use a main target/reject target model. The other way to accomplish this would be to create RecordStarted/OutputRecord event/actions for both targets, with one containing an expression to select records with a balance >= 500, and the other selecting records with a balance < 500. The reason for choosing the reject model is that we only have to write one condition expression to select the records we want, with the remainder being handled by the RecordRejected event.

setupEvents method

```
protected void setupEvents(SourceRep src, TargetRep trg, TargetRep rej) throws DesignSdkException {
    ControlLink link = createLink(src, trg);
    // Add the RecordStarted event, set the condition and add 'if' and 'else' actions
    EventInfo info = link.getEvents().getEventInfoByName("RecordStarted");
    EventHandler eh = link.getEvents().addEvent(info.getEventType());
    FieldRep sFields = src.getRecords().getFields();
    sFields.setFieldPositionByName("Balance");
    eh.setCondition("CDec(FieldAt(\"" + sFields.getFullPath() + "\")) >= 500");
    EventAction action = eh.addAction();
    action.setType("OutputRecord");
    action = eh.addElseAction();
    action.setType("Reject");

    // We now have the condition set up to reject.  We now need to write those records
    // to the reject target
    link = createLink(src, rej);
    info = link.getEvents().getEventInfoByName("RecordRejected");
    eh = link.getEvents().addEvent(info.getEventType());
    action = eh.addAction();
    action.setType("OutputRecord");
}
```

The *setupEvents()* method performs the following actions. Note that it uses a method called *createLink()* to create data links, described shortly.

1. Create a control link from the source to the main target (line 2).
2. Add a *RecordStarted* event to the control link (line 5). Note the event type is specified as an int value, we get this by retrieving the *EventInfo* by name, then getting the event type from the info object.
3. Set a condition expression on the event (line 8). This is an EZscript expression that will check if the source "Balance" field is greater than or equal to 500. Note that since the source field type is text, we use the CDec function to convert it to a decimal value before the comparison. Here we are assuming it will always be a numeric string, if not additional error handling/checking will be needed.
4. Add an "if" action and set it to OutputRecord (lines 9 and 10). This is the action that will be executed if the condition is true. Note that in the general case, the list of "if" actions may be empty or may contain more than one action, that will be executed in order.
5. Add an "else" action and set it to Reject (lines 11 and 12). This action will be executed if the condition evaluates to false. Again, the list of "else" actions may be empty or may contain more than one action.
6. Now, create a link from the source to the "reject" target (line 16).
7. Add a RecordRejected event to the sourcereject link (line 18).
8. Finally add an OutputRecord action to the event (line 20).

createLink method

```
protected ControlLink createLink(SourceRep src, TargetRep trg) throws DesignSdkException {
    src.getRecords().setRecordPosition(0);
    trg.getRecords().setRecordPosition(0);
    return src.addControlLink(src.getRecords().getPath(),
                              "/" + trg.getName() + "/" + trg.getRecords().getPath());
}
```

The *createLink()* method creates a control link between the first record of the passed source to the first record of the passed target. Note that the call to *SourceRep.addControlLink()* takes two record paths. The first is a relative path, since it contains only the path to the record, not the source control name. The second is an absolute path that contains the target name. Assuming the target is the main target, the actual method call will evaluate to

```
return src.addControlLink("R1", "/TARGET_1/R1");
```

Manipulating Data Types (WorkingWithDatatypesSample)

This sample demonstrates changing field datatypes and datatype properties, and how to set up sorting. Since it extends the previous sample, the regular *run()* entry point is not overridden, rather all the setup methods are overridden, and set up to call the base versions then modify the results.

Set up Source

The *setupSource()* method calls *super.setupSource()* to create and do normal configuration of the source, then calls new methods to configure the field of interest and set up sorting:

setupSource method

```
protected SourceRep setupSource(MapRep map, String file) throws DesignSdkException {
    SourceRep src = super.setupSource(map, file);
    src.getRecords().setRecordPosition(0);
    setupSrcDatatype(src);
    setupSorting(src);
    return src;
}

private void setupSrcDatatype(SourceRep src) throws DesignSdkException {
    FieldRep sFields = src.getRecords().getFields();
    sFields.setFieldPositionByName("Balance");
    DatatypeRep dt = sFields.getDataType();
    dt.setDataType("Decimal");
    dt.setLength(16);
}

private void setupSorting(SourceRep src) throws DesignSdkException {
    FieldRep sFields = src.getRecords().getFields();
    sFields.setFieldPositionByName("Balance");
    SortInterface sl = src.getSortLogic();
    sl.insertKeyAfter();
    sl.setPosition(0);      // Position to first (and only) key
    sl.setKeyType(FieldType.DECIMAL);
    sl.setKeyExpression("FieldAt(\"" + sFields.getFullPath() + "\")");
    sl.setAllowDuplicateRecords(true);
}
```

Note that in line 3, *setupSource()* sets the record position. The methods to set up the datatype and sorting will retrieve the *FieldRep* object from the passed source.

The *setupSrcDatatype()* method retrieves the "Balance" field and changes its datatype to "Decimal". It also changes the length to 16, though strictly not necessary.

The *setupSorting()* method is more interesting. At line 20 it retrieves the *SortInterface* object for this source; this object controls all sorting. Sorting allows controlling whether duplicate keys are allowed (at line 25 we set it to true). When duplicate keys are allowed, there is no guarantee as to the order the duplicate records will appear (specifically, there is no guarantee they will appear in the same order as the input). When duplicate keys are not allowed, only one key of any particular value is allowed; there is no guarantee as to which (first, last, other) record will be included.

Aside from duplicate control, the *SortInterface* is a collection (following the count/setindex model) of keys Each key consists of a value expression (the key value) and information about the key (its datatype, length, and whether it is ascending/descending). In the above code, we insert a new key (line 21), then position to it. We then set the key type to DECIMAL, and set the expression to retrieve the contents of the "Balance" field. It is important to set the key type, without it being numeric the value 100 would come before the value 11 (text sorting).

Set up Events

All of the hard work of actually setting up the event handling occurs in the *MapWithEventsAndRejectsSample* class. Here we modify the results:

setupEvents method

```
protected void setupEvents(SourceRep src, TargetRep trg, TargetRep rej) throws DesignSdkException {
    super.setupEvents(src, trg, rej);
    // Now change the conditional expression
    ControlLink link = src.getControlLink(0);
    EventHandler eh = link.getEvents().getEvent(0);
    // Records should still be positioned properly
    FieldRep sFields = src.getRecords().getFields();
    sFields.setFieldPositionByName("Balance");
    // The new conditional does not do the CDEC conversion, since the source field is a numeric
    // data type.  If it were still text, the second record in the primary output file would have
    // a balance of 75.09.
    eh.setCondition("FieldAt(\"" + sFields.getFullPath() + "\") >= 500");
}
```

Here we retrieve the OutputRecord event handler from the source to target control link (line 5). We then change the expression to remove the CDec function call (line 12). This function call is not needed, because the field type itself is now decimal so the ">= 500" comparison will still be performed as a

numeric comparison. Note that removing this function call is not required, this change is made purely to show that we actually have changed the field datatype, and what effects that can have.

Modify Target Datatypes

We will also modify the "Balance" field of both targets. We do this in the *copyFields* method. For the source, we changed the datatype after connecting the source. We can't do the same for the target, as the target may not be able to determine the schema (the file may not exist), and even if it did determine a schema, the *copyFields* method replaces that schema.

copyFields method

```
protected void copyFields(SourceRep src, TargetRep trg, List<String> fields) throws DesignSdkException {
    super.copyFields(src, trg, fields);
    trg.getRecords().setRecordPosition(0);
    FieldRep tFields = trg.getRecords().getFields();
    tFields.setFieldPositionByName("Balance");
    DatatypeRep dt = tFields.getDataType();
    if (trg.getName().equals("TARGET_1")) {
        dt.setDataType("Decimal");
        dt.setDecimalPlaces(4);
    }
    else {
        dt.setAlias("Number");
        dt.setAlign(DatatypeProperties.AlignType.Left);
    }
}
```

We start by calling *super.copyFields()* to build the target schema. We then set the record position, retrieve the *FieldRep* object and position to the "Balance" field (lines 3-5). We then retrieve the *DatatypeRep* object for this field, and modify it. If this is the main target, we change the datatype to Decimal (note this is not strictly necessary, as we changed the source to be Decimal so the target will also be decimal after copying) and set the number of digits after the decimal point to 4. If it's the secondary "reject" target, we set the datatype to "Number" and set it to left-aligned. Note that at line 12 we actually use the *DatatypeRep.setAlias()* method rather than *setDataType()* (as was done at line 8). This is because this datatype is an actual type name of "dBASE Numeric", but is aliased to "Number" in this connector. The *setDataType()* method is strict, in that it will only accept the actual data type. The *setAlias()* method will accept either the actual data type name of the alias, this we could also have used *setAlias("dBASE Numeric")*.

After running this, you can look at the output files and see that StateAccountsDT.asc has the "Balance" field decimal-aligned, with four decimal digits. The StateAccountsRejectDT.asc file has the contents left-aligned, and still at two decimal digits.

Schemas and Intermediate Targets (IntermediateTargetAndSchemasSample)

This sample will read the Accounts.txt file we have used in previous samples. There will be two major differences: it will write to a hierarchical XML file, and will use an incore lookup in an intermediate target to translate two-character states codes (i.e. 'NM') to real names ('New Mexico'). The target schema will be built manually in a separate SchemaBuilder class, that will build the schema and save it as a schema artifact, then the target transformation will reference this external schema. The hierarchy will be as shown:

Build the Schema

All of the code in this section is from the SchemaBuilder class. The class implements one public method:

SchemaBuilder.build method

```
    public String build() throws DesignSdkException {
        TargetRep trg = EngineFactory.instance().createTarget(V10);
        trg.setConnectionTypeName("XML-DMS");
        RecordRep schema = trg.getRecords();
        addStatesToSchema(schema);
        addStateToSchema(schema);
        addCompanyToSchema(schema);
        addAddressToSchema(schema);
        addFinancialsToSchema(schema);

        String sname = Sample.createdArtifactPath(SCHEMANAME);
        trg.getSchemaRef().save(sname);
        return sname;
    }
```

The first thing the method does is creates a target (in this case, since we are creating a target schema) and set the connector type (lines 2 and 3). It's not strictly necessary that the control type (source or target) match the match the intended use, nor is it strictly necessary for the connector type to match. However, it provides the most accurate schemas. Some connectors can support different data types for source and target, and some require entirely different schemas. For the connector type, at runtime an attempt will be made to map the datatypes specified in the schema with the types supported by the connector. If we had used the JSON connector here instead, we would not be able to specify that the single data field for the State and Company elements is an attribute; we could only specify as string and at runtime the connector would have no way of knowing it should be an attribute. In addition, multiple/hierarchical record support becomes an issue. In general, it is best to make sure the connector and control types match what will be used at runtime.

After setting up the control, we call different methods to add each record type to the schema (lines 4-9).

Finally, we set up a full path name for the schema artifact, save it, and return the path. The main class will use this string when setting the schema reference for the XML target.

The *addRecordReference()* method is a helper to add a new record type to the schema, and reference it from a parent type. We assume the passed schema has already been positioned to the parent that will contain this reference.

SchemaBuilder.addRecordReference method

```
    private void addRecordReference(RecordRep schema, String type, String path, boolean unbounded) throws
DesignSdkException {
        schema.addRecordType(makeTypename(type));
        FieldRep fields = schema.getFields();
        if (fields.fieldCount() > 0)
            fields.setFieldPosition(fields.fieldCount() - 1);
        fields.insertAfter();
        fields.setFieldPosition(fields.fieldCount() - 1);
        fields.setName(type);
        fields.getDataType().setDataType(DatatypeProperties.DT_STRUCT);
        if (unbounded)
            fields.setMaxOccurs(Long.MAX_VALUE);
        fields.setRecordReference(makeTypename(type));
        // We've added the record reference to the parent.  Now switch to the child record type
        // and fix the group name
        schema.setRecordPositionByPath(path);
        schema.getGroups().setGroupPosition(0);
        schema.getGroups().setName(makeGroupname(type));
    }
```

Note that the *makeTypename()* and *makeGroupname()* methods are simply helpers that append "_Type" or "_Group" to the passed argument. In general, schemas produced in Studio (either manually or via connector describing the schema) will have the record type be the same name as the record reference, and will append "_Group" for the group name. We differ here to show this is not a requirement.

The first thing this method does is add (an empty) type with the desired name (line 2). We are assuming here that there will not be two references to the same record type. If that were a possibility, then before adding the record type you would want to check and see if it already exist.

The next few lines (3 - 8) add a new field to the parent record. We check to see if there are any fields yet, and if so position to the last one. We then insert (append) a new field, position to it, and set its name. In line 9, we set it's type to indicate it's a record reference. This connector supports hierarchy, but does not include a record reference type in the supported types. Because of this, we can't set it by name, so instead we use the actual type code *Datatype Properties.DT_STRUCT*. In lines 10 and 11, we set the max occurs to unbounded if requested.

Next, we set the actual record reference. This is a seemingly simple operation with a large impact. After doing this, any RecordRep or FieldRep objects for the control are not set to a current field or record. That's because setting the record reference may add a new record instance and delete an existing record instance. That invalidates all positioning.

Assuming the type name that was passed is "State", we now have a field named "State" that references a type named "State_Type". When adding the record type, it was given a single group that is the type name with "_Group" appended, which in this case would be "State_Type_Group". We want this to be "State_Group", so we position to the new instance record using the path, position to the first group, and correct its name.

The *makePath()* method that will be used shortly is another helper:

SchemaBuilder.makePath method

```
public static String makePath(String... segments) {
    StringBuilder path = new StringBuilder();
    for (String seg : segments) {
        if (path.length() > 0)
            path.append("/");
        path.append(seg);
    }
    return path.toString();
}
```

It takes a variable number of strings as arguments, and returns a string with each segment appended with a '/' in between.

Now we get to the real work of adding the record types and building the hierarchy:

SchemaBuilder.addStatesToSchema method

```
private void addStatesToSchema(RecordRep schema) throws DesignSdkException {
    // Here we position to the unlisted "Root Defs" typedef.  Doing this allows adding new
    // root record references.
    schema.setRecordTypePosition("Root Defs");
    addRecordReference(schema, STATES, STATES, false);
}
```

If you have read the Schemas section in "Important Concepts" you know regular typedefs are added to the "type_defs" section, but that the "root_defs" section acts like a stand-alone typedef. You can position to that record type using the name "Root Defs" to allow adding record references. Here we add the top-level States element by positioning to the root typedef (line 4), and then calling *addRecordReference()* to add a "States_Type" typedef and a "States" record reference pointing to it.

After adding the "States_Type" type, we want to add a "State" element, referencing a "State_Type" type:

addStateToSchema method

```
private void addStateToSchema(RecordRep schema) throws DesignSdkException {
    // Position to the parent, add a record reference.
    schema.setRecordPositionByPath(STATES);
    addRecordReference(schema, STATE, makePath(STATES, STATE), true);
    FieldRep fields = schema.getFields();
    fields.insertAfter();           // Add the "name" field
    fields.setFieldPosition(0);
    fields.setName("name");
    fields.getDataType().setAlias("String Attribute");
}
```

Now at line 3 we position to the parent (States_type) by setting the record position using the path (which, since it's the root, is simply "States"), and then add the "State" record reference (line 4).

The rest of this method adds the "name" field. Note we don't have to reset the record position, as addRecordReference leaves the RecordRep positioned to the just-added record reference. We add a field (lines 6 and 7) and set its name to "name" (line 8). Finally, we set its type to "String Attribute" (line 9), using *setAlias()* since this is a type alias.

Adding the "Company" element works the same:

addCompanyToSchema method

```
private void addCompanyToSchema(RecordRep schema) throws DesignSdkException {
    schema.setRecordPositionByPath(makePath(STATES, STATE));
    addRecordReference(schema, COMPANY, makePath(STATES, STATE, COMPANY), true);
    // Now add our name field
```

```
    FieldRep fields = schema.getFields();
    fields.insertAfter();           // Add the "name" field
    fields.setFieldPosition(0);
    fields.setName("name");
    fields.getDataType().setAlias("String Attribute");
}
```

The only difference here is the name of the type being added ("Address") and the path ("States/State/Company").

Finally, we add the Address and Financials record references. Aside from names, the two methods are the same:

add remaining types

```
private void addAddressToSchema(RecordRep schema) throws DesignSdkException {
    schema.setRecordPositionByPath(makePath(STATES, STATE, COMPANY));
    String mypath = makePath(STATES, STATE, COMPANY, ADDRESS);
    addRecordReference(schema, ADDRESS, mypath, false);
    // Now add the data fields
    FieldRep fields = schema.getFields();
    addStringField(fields, "Street", 35);
    addStringField(fields, "City", 16);
    addStringField(fields, "Zip", 10);
}

private void addFinancialsToSchema(RecordRep schema) throws DesignSdkException {
    schema.setRecordPositionByPath(makePath(STATES, STATE, COMPANY));
    String mypath = makePath(STATES, STATE, COMPANY, FINANCIALS);
    addRecordReference(schema, FINANCIALS, mypath, false);
    // Now add the data fields
    FieldRep fields = schema.getFields();
    addStringField(fields, "Standard_Payment", 6);
    addStringField(fields, "Payments", 7);
    addStringField(fields, "Balance", 6);
}

private void addStringField(FieldRep fields, String name, int length) throws DesignSdkException {
    fields.insertAfter();
    fields.setFieldPosition(fields.fieldCount() - 1);
    fields.setName(name);
    fields.getDataType().setAlias("String");
    fields.getDataType().setLength(length);

}
```

The *addFinancialsToSchema()* and *addCompanyToSchema()* are the same, as noted above, except for the type name and the names/lengths of the text fields added. Both methods call *addStringField* which appends a field, positions to it, then sets its name, datatype (via *setAlias()*) and length.

Sample Setup and Execution

The standard *run()* entrypoint performs the usual setup (creating/configuring the transformation), saves the artifacts and then executes the transformation:

run method

```
protected boolean run() {
    try {
        MapRep map = EngineFactory.instance().createMap(V10);
        // Set the log file
        map.getOptions().getLogSettings().setLogFile("$(LOG)/IntermediateTargetAndSchemasSample.log");
        SourceRep src = setupSource(map, ACCOUNTS_SRC);
        setupSorting(src);
        IntermediateTarget lookup = setupLookup(map);
        TargetRep trg = setupTarget(map, TFILE, "TARGET_1");

        src.getRecords().setRecordPosition(0);
        lookup.getRecords().setRecordPosition(0);
        mapFields(trg, src.getRecords().getFields(), lookup.getRecords().getFields());
```

```
            setupEvents(src, lookup, trg);

            map.saveAs(createdArtifactPath(MAPNAME + ".map.rtc"), createdArtifactPath(MAPNAME + ".map"));
            map.execute(null);
        }
        catch (DesignSdkException e) {
            e.printStackTrace();
            logger.severe("Error running sample: " + e.getLocalizedMessage());
            return false;
        }
        return true;
    }
```

Here we start by creating the transformation (line 3) and then create the source, lookup and target (lines 6-9). Note that we use the *setupSource* method from from the base class, then configure sorting. As a convenience (since they only have one record type each) we then set the current record in the source and lookup, then map the fields and set up the event handling. We then save the artifacts and execute the transformation.

Sort the Source

We configure the source to sort on the State field:

setupSorting method

```
    private void setupSorting(SourceRep src) throws DesignSdkException {
        FieldRep sFields = src.getRecords().getFields();
        src.getRecords().setRecordPosition(0);
        sFields.setFieldPositionByName("State");
        SortInterface sl = src.getSortLogic();
        sl.insertKeyAfter();
        sl.setPosition(0);      // Position to first (and only) key
        sl.setKeyExpression("FieldAt(\"" + sFields.getFullPath() + "\")");
        sl.setAllowDuplicateRecords(true);
    }
```

We want the XML file to be grouped by states, so we will use the *GroupStarted* event to write a new State element each time the state changes in the source. While *GroupStarted* does not, in and of itself, require sorting the input, if you are grouping on a field's contents you generally will want to sort. Here we configure sorting on the "State" input field. This actually produces an interesting result: in the output, the states are slightly out of order. This is because we actually present the state names in the output file, but the input values we are sorting are state abbrevations. So the output presents Alaska first, followed by Alabama. This is because the state code for Alaska is AK, while the code for Alabama is AL.

Create and Configure the Lookup

Using a lookup intermediate target can produce powerful results, but setup of the lookup is quite simple:

setupLookup method

```
    private IntermediateTarget setupLookup(MapRep map) throws DesignSdkException {
        IntermediateTarget lookup = map.addIntermediateTarget(IntermediateTarget.IntermediateType.IncoreLookup,
                                                              "LOOKUP");
        lookup.setConnectionTypeName("ASCII (Delimited)");
        lookup.setConnectionPart("file", "$(SRC)/StateCodeStateData.txt");
        lookup.setOption("header", "true");        // First record is a header
        lookup.connect();           // Connect so we build the schema
        return lookup;
    }
```

This is really no different than setting up a source or target. In line 2 we add a new intermediate target, the only difference is we specify the type of intermediate target to add as well as the name. After that we set up the connection like any source or target: set the connection type, set the connection parts then the options, and possibly connect so we can retrieve the schema (if appropriate). Note that intermediate targets function as both a source and target. When we set up event handling, you will see the intermedate target used as both the source and target of control links.

Of course, how complex the setup is depends on the type of intermediate target. Event handling related to intermediate targets can also be complex (discussed more in the appropriate section). Here we could have used any lookup type except dynamic SQL lookup (and we could use that too, if we had the state codes table in a database), and we could have also used the actual join intermediate.

Create and Configure the XML Target

The target setup is also quite simple, although it is a bit more complex than we have been seeing:

setupTarget method

```
protected TargetRep setupTarget(MapRep map, String file, String name) throws DesignSdkException {
    String sname = new SchemaBuilder().build();
    TargetRep trg = map.addTarget(name);
    trg.setConnectionTypeName("XML-DMS");
    trg.setConnectionPart("file", file);
    trg.getSchemaRef().setName(sname);
    trg.getSchemaRef().load(sname, null);
    return trg;
}
```

The first thing we do (line 2) is create a *SchemaBuilder* and tell it to build a schema. We save the resulting artifact path in a local variable, then proceed to create the target as usual (lines 3-5).

In line 6, we retrieve the *SchemaRef* object from the target and set its name. All controls have a *SchemaRef*, used for configuring external schema handling. By setting the name, we are telling the control that it will load its schema from the specified path. Note that this is all this line does, it sets the name to be stored as the schema_rref element of the saved artifact, which will cause it to be loaded if the transformation is loaded from disk. Right now, we need to load it ourselves, which we do in line 7. The second parameter to *SchemaRef.load()* is the zip key, in case the artifact is in an encrypted zip /djar. It's not, so we can safely pass *null* as the value.

Note we do not connect the target here. There is no need, since we are supplying the schema. If the schema were very complex, and a target file already existed, connecting could take a long time while the target schema is described.

Mapping Fields

In the other transformation samples, we map fields as part of copying fields from the source to the target. Here, we can't do that as we are using an external pre-built schema on the target. The process is further complicated by the multiple, hierarchical records. So we will specifically map certain fields. The method we use to map a single field is:

mapField method

```
private void mapField(FieldRep trg, String tname, FieldRep src, String sname) throws DesignSdkException {
    trg.setFieldPositionByName(tname);
    src.setFieldPositionByName(sname);
    trg.addLink(src.getFullPath(), false);
}
```

This method is a convenience, it positions the source and target controls' fields properly, then creates a DataLink. In the past, we have added a target field mapping expression of FieldAt("/SOURCE_1/R1/fieldname"). This (in theory tells the engine to execute the specified EZscript expression. Now, we are instead using the *addLink()* method instead of *setExpression()*. The V10 engine was optimized for simple field to field mapping, by using datalinks instead of expressions. In V11, the engine was modified to look at an expression being set, and if it was a simple field reference would replace it with a datalink. When asking for a field's mapping expression, if it was actually a datalink it would build a simple field mapping expression. Either method is valid and produces the same results at runtime. However, explicitely adding the link results in slightly easier to read code.

We have one field that needs special consideration. That is the "name" field (attribute) in the "Company" record/element. In our source, some of the company names are empty. We want check and map an empty company name to use the "Name" field instead. The method

mapCompanyName method

```
private void mapCompanyName(FieldRep trg, FieldRep src) throws DesignSdkException {
    // This says check to see if the "Company" field is empty.  If so, we want to use the "Name" field,
    // otherwise the "Company" field.
    trg.setFieldPositionByName("name");
    StringBuilder exp = new StringBuilder("if Len(FieldAt(\"");
    src.setFieldPositionByName("Company");
    exp.append(src.getFullPath());
    exp.append("\"))== 0 then \r\n    FieldAt(\"");
    src.setFieldPositionByName("Name");
    exp.append(src.getFullPath());
    exp.append("\")\r\nelse\r\n    FieldAt(\"");
    src.setFieldPositionByName("Company");
```

```
        exp.append(src.getFullPath()).append("\")\r\nend if");
        trg.setExpression(exp.toString());
    }
```

simply builds an if-then-else expression to map the source "Name" field if the source "Company" field is empty (condition Len(FieldAt("/SOURCE_1/R1 /Company")) == 0 true).

Finally, the code to actually manage the mapping:

mapFields method

```
    private void mapFields(TargetRep trg, FieldRep srcFields, FieldRep lookupFields) throws DesignSdkException {
        // First, set up the lookup action key
        mapField(lookupFields, "ST", srcFields, "State");
        RecordRep records = trg.getRecords();
        FieldRep trgFields = records.getFields();
        records.setRecordPositionByPath("States/State");
        mapField(trgFields, "name", lookupFields, "STATE");
        records.setRecordPositionByPath("States/State/Company");
        mapCompanyName(trgFields, srcFields);
        records.setRecordPositionByPath("States/State/Company/Address");
        mapField(trgFields, "Street", srcFields, "Street");
        mapField(trgFields, "City", srcFields, "City");
        mapField(trgFields, "Zip", srcFields, "Zip");
        records.setRecordPositionByPath("States/State/Company/Financials");
        mapField(trgFields, "Standard_Payment", srcFields, "Standard Payment");
        mapField(trgFields, "Payments", srcFields, "Payments");
        mapField(trgFields, "Balance", srcFields, "Balance");
    }
```

This is mostly very simple. The source and lookup controls have already been positioned to the proper (only) record, so we pass the *FieldRep*s for these controls instead of the control itself. The target record positioning is done by the path to the record. So we simply position the target, then call *mapField()* (or *mapCompanyField()* when appropriate).

Note that in line 3, we map with *lookupFields* as the target, but in line 7 we map it as the source. An intermediate target acts as both a source and a target. For lookups, the target mapping defines the key(s). In this case, we map the "ST" field in the lookup (which contains the state codes) to the source "State" field, which contains the state codes coming in (acting as a target). We then map the target "name" field in the target State record to the "STATE" field in the lookup (acting as a source).

Set Up Event Handling

Finally, we set up the event handling. To avoid repeating the same code many times, we use a helper method to actually create events and actions:

setEventAndAction method

```
    private void setEventAndAction(SourceInterface src, TargetInterface trg,
                                   String event) throws DesignSdkException {
        ControlLink link = src.addControlLink(src.getRecords().getPath(),
                                              "/" + trg.getName() + "/" + trg.getRecords().getPath());
        EventInfo info = link.getEvents().getEventInfoByName(event);
        EventHandler eh = link.getEvents().addEvent(info.getEventType());
        if (event.equals("GroupStarted")) {
            eh.setParameterValue(0, "field_list");  // groupby_type
            eh.setParameterValue(1, "State");       // State field is the group
        }
        EventAction action = eh.addAction();
        action.setType("OutputRecord");
    }
```

This method start off by adding a control link from the source current record to the target current record (lines 3 and 4). It then looks up the event info for the named event (line 5), and adds the event to the control link (line 6).

In lines 7-10, we check to see if this is a "GroupStarted" event (there will be two). If it is, we need to set two parameters: what kind of group we're creating (a field list as opposed to an EZscript expression), and then supply the field list (which is a single field in this case). Note that we do not need to provide any path information, since the control link is from a source record to a target record, the field(s) are in the source record.

Finally (lines 11 and 12) we add an action and set it to "OutputRecord". Note we chose a specific event model (discussed in more detail later) that allows all actions to be OutputRecord.

The actual event model for this transformation is controlled by the *setupEvents()* method:

setupEvents method

```
    private void setupEvents(SourceRep src, IntermediateTarget lookup, TargetRep trg) throws DesignSdkException
{
        src.getRecords().setRecordPosition(0);
        lookup.getRecords().setRecordPosition(0);
        trg.getRecords().setRecordPositionByPath("States");
        // Write States element at start
        setEventAndAction(src, trg, "SourceStarted");
        // Each time the state changes:
        setEventAndAction(src, lookup, "GroupStarted");     // Do the lookup
        trg.getRecords().setRecordPositionByPath("States/State");
        setEventAndAction(src, trg, "GroupStarted");        // And write the state element
        // Now, write the Company, Address and Financials for each record
        trg.getRecords().setRecordPositionByPath("States/State/Company");
        setEventAndAction(src, trg, "RecordStarted");
        trg.getRecords().setRecordPositionByPath("States/State/Company/Address");
        setEventAndAction(src, trg, "RecordStarted");
        trg.getRecords().setRecordPositionByPath("States/State/Company/Financials");
        setEventAndAction(src, trg, "RecordStarted");
    }
```

We start off by positioning the source and lookup current record to the only record they each contain. In this sample this isn't necessary, as they will be positioned correctly by *run()* before the call to *mapFields()*. However, resetting it here protects against adding code that resets (unsets in this case) the current record.

We then proceed to create the actual control links, events and actions. An important point to remember is that control links are processed in the order they appear (and why the *SourceInterface.moveControlLink()* method exists).

So the first thing we do is position to the "States" record (root of the resulting XML), and specify that when we are connected and ready to read records (SourceStarted event), we will write a record (remember, setEventAndAction always adds an OutputRecord action) (lines 4-6)

We then (line 8) set up to output a record to the lookup target whenever a new values is seen in the State field (GroupStarted). We then position to the States/State record in the target (line 9), and also do an output record (line 10). We chose this model because it was easier to implement with shared code, and because we know there will never be multiple records produced in the lookup matching the key value. The preferred way to do this is actually on the source to lookup control link GroupStarted event, do the OutputRecord action, then on the same event handler add an OutputRecordset action, specifying the lookup as the port. We would then add a link from the lookup to the target States/State record, and add a RecordStarted/OutputRecord event/action there (and of course remove the second GroupStarted at line 10). This model would be required if the intermediate could output multiple records (i.e. a lookup with multiple results, a "temporary target", etc.).

After that, we simply add RecordStarted/OutputRecord events for the remaining three target records (lines 12-17). This is because we want to produce Company, Address and Financials elements for each input record. The target will always write these under the most recently written State element.

Create a Simple Process (CreateSimpleProcessSample)

This sample creates a very simple process. It has only three steps, start and stop steps plus a scripting step that will write a message to the (DataConnect) log. It is all contained in standard *run()* entry point:

Create Simple Process

```
    protected boolean run() {
        String logname = "$(LOG)/CreateSimpleProcess.log";
        ProcessRep process = null;
        try {
            process = EngineFactory.instance().createProcess(V10);
            process.getOptions().getLogSettings().setLogFile(logname);

            // Add the required start and stop steps
            Step startStep = process.addStep(Step.StepType.Start);
            Step stopStep = process.addStep(Step.StepType.Stop);

            // Now create a script step, and set the script
            ScriptingStep scriptingStep = (ScriptingStep) process.addStep(Step.StepType.Scripting);
            scriptingStep.setName("Scripting");
```

```
            scriptingStep.setExpression("LogMessage(\"INFO\", \"" + MESSAGE + "\")");

            // Now create the step links.  None of the steps are decision steps, so the third argument
            // can always be null.
            process.addLink(startStep.getName(), scriptingStep.getName(), null);
            process.addLink(scriptingStep.getName(), stopStep.getName(), null);

            // Save the process, and execute it
            process.save(createdArtifactPath(PROCESSNAME + ".process"));
            process.execute(null);

            // Verify the process actually logged the message
            File lfile = new File(process.getSystemContext().expandMacros(logname));
            String log = FileUtils.readFileToString(lfile);
            if (!log.contains(MESSAGE)) {
                logger.severe("The log file did not contain the expected message");
                return false;
            }
        }
        catch (DesignSdkException e) {
            logger.severe("Error running sample: " + e.getLocalizedMessage());
            return false;
        }
        catch (IOException e) {
            logger.severe("Error reading log: " + e.getMessage());
            return false;
        }
        finally {
            if (process != null)
                process.close();
        }
        return true;
    }
```

The operations performed are:

1. Create the process engine and set up logging (lines 5 and 6)
2. Add the start and stop steps (lines 9 and 10). We keep references to all the steps so we can use them to get step names when creating links. Note that we do not set names on the start and stop steps. While calling setName is allowed, it will be ignored on these steps. Both start and stop steps are required, and only one of each is allowed.
3. Create the scripting step, set its name and add the expression it will execute (lines 13-15).
4. Create the links (lines 19 and 20). We link the start step to the scripting step, and then the scripting step to the stop step. The third argument to the *addLink()* method is a Boolean. It should always be *null* unless the source step for the link is a decision step, in which case it must indicate whether it is the true or false branch.
5. Save the process and execute it (lines 23 and 24).
6. In this sample, we will also verify the sample ran and produced the expected results. We read the log file, and make sure it contains the message we told the process to write to the log.

Transformation Step in Process (ProcessWithTransformationSample)

This sample shows how to execute a transformation from a process. We do this with a Transformation step. In order to run without relying on other samples (allowing this sample to run stand-alone), we will create a simple transformation and save it as a built artifact.

run method

```
    protected boolean run() {
        String logname = "$(LOG)/" + PROCESSNAME + ".log";
        try {
            ProcessRep process = EngineFactory.instance().createProcess(V10);
            process.getOptions().getLogSettings().setLogFile(logname);

            // Add the required start and stop steps
            Step startStep = process.addStep(Step.StepType.Start);
            Step stopStep = process.addStep(Step.StepType.Stop);

            // Create the transformation step, and add links
            Step tstep = createTransformationStep(createTransformation(), process);
            process.addLink(startStep.getName(), tstep.getName(), null);
            process.addLink(tstep.getName(), stopStep.getName(), null);
```

```
        // Save the process, and execute it
        process.save(createdArtifactPath(PROCESSNAME + ".process"));
        process.execute(null);
        process.close();
    }
    catch (DesignSdkException e) {
        logger.severe("Error running sample: " + e.getLocalizedMessage());
        return false;
    }
    return true;
}
```

The operations here are basically the same as the previous sample, except rather than create a scripting step, we call *createTransformationStep()* to create a transformation step. The first argument is the name of a runtime config, which comes from calling the *createTransformation()* method.

Create and Configure the Transformation Step

createTransformationStep method

```
protected Step createTransformationStep(String tfname, ProcessRep process) throws DesignSdkException {
    V10TransformationStep tstep = (V10TransformationStep) process.addStep(Step.StepType.Transformation);
    tstep.setName("Transformation_1");
    tstep.setConfigFile(tfname);
    return tstep;
}
```

This method creates the transformation step, sets its name, and configures the step with the runtime config to execute. Note that for a V10 process, *Step.StepType.Transformation* creates a *V10TransformationStep* object, while *Step.StepType.AlternateTransformation* would produce a *TransformationStep* object. For a V9 process, these objects types are reversed.

The values stored as the config file (line 4) is simply the name of a runtime config whose entry point is the map file to use. It can be a simple file name because it is located in the same directory as the process. If it were in a different directory, or the process will be run without saving, then this name needs to resolve to a full path to the runtime config file. Remember that multiple runtime config files can have the same map as their entrypoint, thus allowing re-use of a single map in multiple steps with different runtime settings.

There are two versions of *setConfigFile()*, the other version takes the name of a config file and a flag indicating whether in SQL sessions set in the step should be cleared (the single argument version automatically clears the SQL Sessions). The only other configuration to be done for this step types would be setting up SQL sessions, which unlike the *TransformationStep()* is done with a single method that provides both the SQL Session and the name of the node/control it applies to.

Create the Transformation Artifacts

As mentioned above, in the interest of providing artifacts and outputs unique to each sample, we create our own transformation artifacts to run with this sample. We will actually load an existing transformation, change the output, and save it as a local transformation.

createTransformation method

```
protected String createTransformation() throws DesignSdkException {
    String mapname = suppliedArtifactPath("AccountsExample.map.rtc");
    MapRep map = EngineFactory.instance().createMap(V10);
    map.load(mapname);
    TargetRep trg = map.getTarget(map.getTargetNames().get(0));
    trg.setConnectionPart("file", TFILE);
    map.saveAs(createdArtifactPath(MAPNAME + ".map.rtc"), createdArtifactPath(MAPNAME + ".map"));
    String tfname = map.getTransformationFileName();
    map.close();
    return tfname;
}
```

The *TFILE* and *MAPNAME* variables are class members that provide names specific to this example. Note towards the end, we retrieve the transformation file name (this is the .map.rtc file) from the *MapRep*, then close the map and return the filename.

Alternate Transformation Step in Process (ProcessWithAlternateTransformationSample)

This sample extends the previous (ProcessWithTransformationSample) sample. It uses the same *run()* method, but overrides the *createTransformationStep()* and *createTransformation()* methods.

createTransformationStep method

```
protected Step createTransformationStep(String tfname, ProcessRep process) throws DesignSdkException {
    // For a V10 process, Transformation returns a V10TransformationStep and AlternateTransformation
    // returns a TransformationStep.  In a V9 process, these are reversed.
    TransformationStep tstep =
            (TransformationStep) process.addStep(Step.StepType.AlternateTransformation);
    tstep.setName("Transformation_1");
    // Instead of supplying a configuration, we supply the map file.  We then set an option to
    // determine if the configuration is found by using MAPNAME.tf.xml, or if an "override"
    // configuration is used instead, which would be named processname$stepname.tf.xml
    tstep.setMapFile(tfname.replace("tf.xml", "map.xml"));
    tstep.setOverrideTransformationFile(false);
    return tstep;
}
```

Unlike the *V10TransformationStep*, configuration of the (V9) *TransformationStep* involves specifying a map file, and leaving the step to infer to config (tf.xml) file. Notice that we simply replace "tf.xml" in the passed name with "map.xml", and the next line specifies we are not using an override tf.xml. There are two methods the step can use to infer the tf.xml name:

1. If the overrideTransformationFile property is false, then replace ".map.xml" with ".tf.xml" (the opposite of what we did above to get the map name).
2. If the overrideTransformationFile property is true, then it looks for a file called Processname$Stepname.tf.xml. In this sample, the process is named "ProcessWithAltTransformation" and this step is named "Transformation_1", the step will look for "ProcessWithAltTransformation$Transformation_1.tf.xml".

Create the Transformation Artifacts

This sample also overrides the *createTransformation()* method, although the code is almost exactly the same as for ProcessWithTransformationSample:

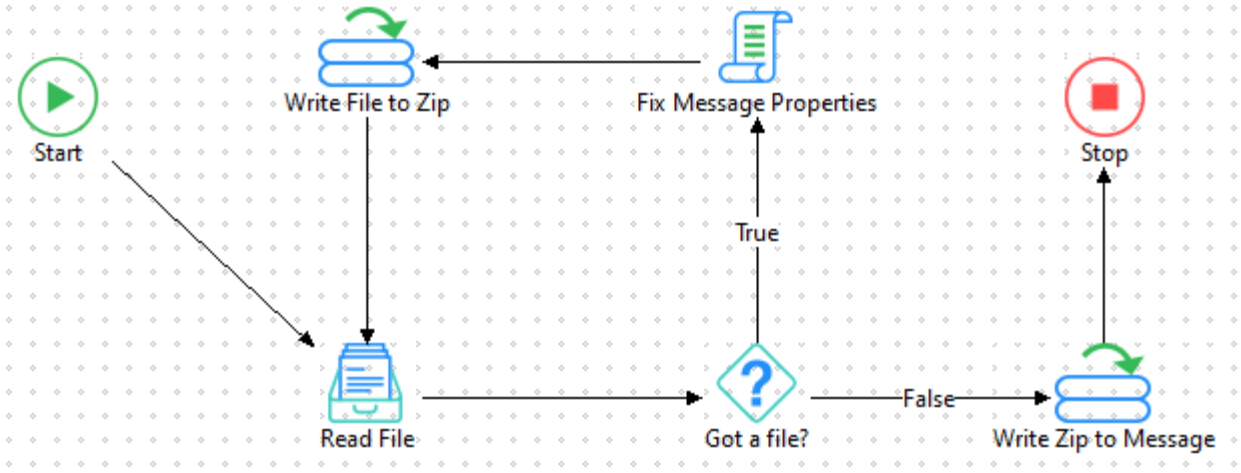
createTransformation method

```
protected String createTransformation() throws DesignSdkException {
    String mapname = suppliedArtifactPath("AccountsExample.tf.xml");
    MapRep map = EngineFactory.instance().createMap(V9);
    map.load(mapname);
    TargetRep trg = map.getTarget(map.getTargetNames().get(0));
    trg.setConnectionPart("file", TFILE);
    map.saveAs(createdArtifactPath(MAPNAME + ".tf.xml"), createdArtifactPath(MAPNAME + ".map.xml"));
    String tfname = map.getTransformationFileName();
    map.close();
    return tfname;
}
```

The only difference is we create a V9 *MapRep* and load a supplied V9 transformation into it. When we save the artifacts, we specify the V9 extensions (.tf.xml and .map.xml).

Message Component/Queue in Process

This sample is much more complex than previous examples. It will read a set of files out of the included data directory, add those files to a zip archive, and write that out to a file. It will ultimately appear as follows in DataConnect Studio:



This sample uses the File Folder Queue component to read files, and the Zip Aggregator to build the zip file. The basic flow read a file, then look to see if we actually read one. If so (true branch), we fix message properties in the message, add it to the zip, and read the next file. When there are no more files (false branch), we write the zip out and then finish the process.

Configuration of each component and each step is broken up into separate methods, leaving the entrypoint to look like this:

run method

```
    protected boolean run() {
        try {
            // First, we create and configure the process
            ProcessRep process = configureProcess();
            // Create the components.  Configure the ffqueue, but the zip aggregator doesn't need
            // configuration.  We could do this in configureProcess, but we'll need access to the
            // components when configuring the steps that use them.
            Component ffqueueComponent = configureFFQueue(process);
            Component zipComponent = process.addComponent("Zip", Component.ComponentCategory.AGGREGATOR,
                                                          "Zip Aggregator", "1.0.0");
            // Now configure the steps.  We'll do start and stop separately as well, since they will
            // both have EZscript expressions.
            Step startStep = configureStartStep(process);
            Step stopStep = configureStopStep(process);
            Step queueStep = configureQueueStep(process, ffqueueComponent);
            Step decisionStep = configureDecisionStep(process, queueStep);
            Step scriptStep = configureScriptStep(process);
            Step zipPutStep = configureZipPutStep(process, zipComponent);
            Step zipGetStep = configureZipGetStep(process, zipComponent);

            // Link the steps
            process.addLink(startStep.getName(), queueStep.getName(), null);
            process.addLink(queueStep.getName(), decisionStep.getName(), null);
            process.addLink(scriptStep.getName(), zipPutStep.getName(), null);
            process.addLink(zipPutStep.getName(), queueStep.getName(), null);
            // Links from the decision step need to indicate whether they are for the true
            // branch or false branch
            process.addLink(decisionStep.getName(), scriptStep.getName(), true);
            process.addLink(decisionStep.getName(), zipGetStep.getName(), false);
            process.addLink(zipGetStep.getName(), stopStep.getName(), null);

            // Save the process, and execute it
            process.save(createdArtifactPath(PROCESSNAME + ".process"));
            process.execute(null);
            process.close();
        }
        catch (DesignSdkException e) {
            e.printStackTrace();
            logger.severe("Error running sample " + this.getClass().getSimpleName() + ": "
                          + e.getLocalizedMessage());
            return false;
```

```
        }
        return true;
    }
```

The following main steps are:

- Create and configure the process itself.
- Create and configure the components. Note that the zip component doesn't require any configuration, so it is added directly in *run()* using the *add Component()* method. The components must be created before the steps that use them can be fully configured.
- Create and configure each of the steps.
- Create the links. Note that there are two links with the decision step as the source, one for the true branch and one for the false. (You can have multiple links for each branch if desired, just like you can have multiple links starting from or ending with any given step).
- Save and execute the process.

Create and Configure the Process (configureProcess)

The *configureProcess()* method is used to configure the *ProcessRep* after creating it:

configureProcess method

```
    private ProcessRep configureProcess() throws DesignSdkException {
        ProcessRep process = EngineFactory.instance().createProcess(V10);
        String logname = "$(LOG)/" + PROCESSNAME + ".log";
        process.getOptions().getLogSettings().setLogFile(logname);
        process.getOptions().setEnableLogging(true);
        // Normally, we like to break when an error is encountered.  In this case, however, the step
        // that reads file generates an error when EOF is hit.  We want to keep processing, so we need
        // to turn off the break on first error property.  By default, steps that error will still
        // terminate the process, we will also need to turn off a step property.
        process.getOptions().setBreakOnFirstError(false);

        // Now add variables to access the two messages we'll use, one for the files we read and
        // one for the body of the zip file.
        VariablesContainer vars = process.getGlobalVariables();
        UserVariable srcMsg = vars.add("SrcMessage");
        srcMsg.setDatatype(UserVariable.Datatype.DJMessage);
        UserVariable zipMsg = vars.add("ZipMessage");
        zipMsg.setDatatype(UserVariable.Datatype.DJMessage);

        return process;
    }
```

We first create the process engine, then set up the logging.

At line 10, we set the breakOnFirstError to false. Normally, any step that generates an error will automatically terminate the process. Each step does have a property to ignore errors, but this property acts as a master switch, and if it is true then the step property to ignore errors is ignored. We do this because some components (queue and iterator mainly) produce an error to indicate no more input. By turning this off and setting the queue read step to ignore errors, we can test the result in the decision step to determine if we should loop or go write the zip and terminate.

Finally (line 15) we retrieve the global variables container and create two global variables of type DJMessage. Message objects are separate from message variables, but we need message variables to access messages in EZscript. Components access message objects directly. So in the scripts that will be added to various steps, the variable is being accessed. In the components when we specify a message, we supply the name of the actual message. It is frequently convenient to name the variable and message the same, but we haven't here to illustrate they are different.

Create and Configure Queue Component (configureFFQueue)

The *configureFFQueue()* method creates and configures the component used to read files:

configureFFQueue

```
    private Component configureFFQueue(ProcessRep process) throws DesignSdkException {
        Component qc = process.addComponent("FFQueue", Component.ComponentCategory.MESSAGE_QUEUE,
                                            "File Folder Queue", "1.0.0");
        qc.setOption("Directory", "$(SRC)");
        qc.setOption("GetMsgClass", "asc");
        qc.setOption("BrowseMode", "True");
        return qc;
    }
```

The *ProcessRep.addComponent()* method has the following signature:

```
    Component addComponent(String name, Component.ComponentCategory category, String type, String version,
boolean...validate) throws DesignSdkException;
```

The *name* parameter sets the name that will identify this particular component instance. Here we save the resulting component object, but it can also be retrieved from the process by name (and category), and the name is also used in the saved process artifact. The *category* parameter specifies one of size component types (this sample uses a MESSAGE_QUEUE and an AGGREGATOR). The *type* and *version* parameters uniquely identify what type of component is desired. It is possible to have multiple versions of the same component type installed, and the component can specify what versions are compatible with each other (currently, as one example, DataConnect supplies three different versions of the REST invoker: 1.0.0, 2.1.0, and 3.0.0). The final *validate* parameter is an optional boolean value indicating whether the component type should be validated. Omitting this parameter (as we have here) does validate the component type. You can use the static method *Component.getComponentTypes()* to get a list of actual available types.

AFter creating the component, we do some configuration (this component has many more options than we actually configure here. Some are not needed, and some the default value is fine. The *Component.getOptions()* method will return a list of *OptionInfo* objects describing the options available for this component.

- **Directory** specifies where to look for files
- **GetMsgClass** specifies the extension to look for. There is also a **Pattern** option that allows specifying a filename pattern, but the default of '*' works fine.
- **BrowseMode** specifies whether the component runs in browse mode or not. With browse mode off, files will be deleted after they have been read. We want to leave the files, so we set BrowseMode on.

Create and Configure the Start Step (configureStartStep)

We use a separate method to create the start step, because we also need to do some configuration:

configureStartStep method

```
        StartStep step = (StartStep) process.addStep(Step.StepType.Start);
        step.setExpression("set SrcMessage = findMessage(\"SrcMessage\")\r\n"
                           + "if SrcMessage is nothing then\r\n"
                           + "\tset SrcMessage = new DJMessage(\"SrcMessage\")\r\n"
                           + "end if\r\n"
                           + "set ZipMessage = findMessage(\"ZipMessage\")\r\n"
                           + "if ZipMessage is nothing then\r\n"
                           + "\tset ZipMessage = new DJMessage(\"ZipMessage\")\r\n"
                           + "end if\r\n");
        step.getPresentationInfo().setX(77);
        step.getPresentationInfo().setY(145);
        return step;
    }
```

The *StartStep* and *StopStep* steps subclass *ScriptingStep* and thus allow setting an expression. We use this to initialize the two DJMessage variables as part of the script step. The paradigm used here

```
  set MessageVar = findMessage("Message Name")
  if MessageVar is nothing then
    set MessageVar = new DJMessage("Mesage Name")
  end if
```

is considered the best practice when initializing a DJMessage variable. This allows successful initialization of the variable whether the message already exists (findmessage) or not (new djmessage). This may not seem very important in this case, but if the process is ever added to another process (or a transformation is referenced in a process) and the variable has been initialized using "new djmessage", the parent process will probably also have initialized the variable so initialization here would fail.

Note that after setting the expression in the start step, we set some presentation info. This is not necessary, the process will execute perfectly well without it. However, if you were look at the process artifact from any of the previous samples in Studio, you would see all the steps bunched in the top left corner, since they will all have an X,Y origin of 0,0. In this sample, we will set the coordinates of each step, and if you later load the process in Studio it will appear as shown in the screen shot at the beginning of this chapter.

Create and Configure the Stop Step (configureStopStep)

We also do some configuration in the stop step. Here is where we write the body of the zip file (in ZipMessage) out to a file. We could also do this in a script step.

configureStopStep method

```
    private Step configureStopStep(ProcessRep process) throws DesignSdkException {
        StopStep step = (StopStep) process.addStep(Step.StepType.Stop);
```

```
        step.setExpression("FileWrite(\"$(TRG)/test.zip\", ZipMessage.Body, ENC_ISO8859_1)");
        step.getPresentationInfo().setX(600);
        step.getPresentationInfo().setY(145);
        return step;
    }
```

Note we include an encoding in the FileWrite function call. The zip file body (in *ZipMessage.body*) is binary content. The ISO-8859-1 encoding is essentially an "identity" encoding, it indicates to "transform" the unicode characters by simply taking the low-order byte of each character and writing that to disk.

Create and Configure the Queue Step (configureQueueStep)

This method creates the File Folder Queue step named "Read File", so named because it is responsible for reading each file selected by the ffqueue component.

configureQueueStep method

```
    private QueueStep configureQueueStep(ProcessRep process, Component queue) throws DesignSdkException {
        QueueStep step = (QueueStep) process.addStep(Step.StepType.Queue);
        step.setName("Read File");
        step.setComponent(queue);
        // Configure the action.  We need to set the action type, then configure the single (in
        // this case) parameter.
        step.setActionType(ActionType.GetMessage);
        step.getAction().setParameterValue("Message", "SrcMessage");
        //   To see the available parameters:
//        ComponentAction action = step.getAction();
//        for (int i = 0; i < action.getParameterCount(); ++i) {
//            ComponentActionParameterInfo info = action.getParameterInfo(i);
//            System.out.println("Parameter info:" + info.getName() + " (" + info.getPrompt() + ") has type "
//                               + info.getType());
//        }

        // Since this is a queue, we don't want the error that comes with EOF to terminate the
        // process.
        step.setIgnoreErrors(true);
        step.getPresentationInfo().setX(228);
        step.getPresentationInfo().setY(288);
        return step;
    }
```

We start off by creating the step with type *Queue*, which returns a *QueueStep* object. We set its name to "Read File" (line 3). Note this is the first time we have set a step name. The start and stop steps must be named "Start" and "Step" respectively (and calling *setName* on either step type will be ignored). We also call *setComponent* to associate this step with the queue component (line 4).

Component Actions, Parameters and Options

Components execute actions, using parameters, and support options/properties, both instance-level and related to each action. Each component step executes causes the component instance associated with the step to execute a single action. In this step, we execute the *ActionType.GetMessage* action on the 'queue' ffqueue component instance (line 7), using *SrcMessage* as the value for the *Message* parameter (line 8). Note that the commented lines 9-15 show how to get information about the parameters associated with the set action.

The ffqueue component has a number of instance properties (see *configureFFQueue()* for details), but no properties associated with the get action. See *configureZipGetStep()* for an example of setting action-associated component properties.

At line 19, we tell this step to ignore errors. Queue components signify no more messages to read by generating a step error, which would normally terminate the process. Instead, we want to ignore the EOF error so we can properly complete processing (see configureDecisionStep). Remember, the step must be set to ignore properties **and** the process must be set to not break on first error (see *configureProcess()*) for this to work.

Create and Configure the Decision Step (configureDecisionStep)

After reading each file, we execute a decision step. Each decision step evaluates an expression, and determines which link(s) to follow based on the boolean result of that expression:

configureDecisionStep method

```
    private DecisionStep configureDecisionStep(ProcessRep process, Step keyStep) throws DesignSdkException {
        DecisionStep step = (DecisionStep) process.addStep(Step.StepType.Decision);
        step.setName("Got a File?");
        step.setExpression("Project(\"" + keyStep.getName() + "\").ReturnCode == 0");
        step.getPresentationInfo().setX(420);
        step.getPresentationInfo().setY(288);
        return step;
    }
```

After creating and naming the step, we set its expression (line 4). The expression can be arbitrarily complex, it just needs to result in a value to can be evaluated as a boolean. In this case, we are simply checking that the return value from the previous step (provided as the second parameter) is 0. The script step we execute if the decision is true will set properties on the message containing the file contents; we could just as easily done that work as part of this expression.

Create and Configure the Script Step (configureScriptStep)

After the decision step, we create the script step which will modify the message properties:

configureScriptStep method

```
    private Step configureScriptStep(ProcessRep process) throws DesignSdkException {
        ScriptingStep step = (ScriptingStep) process.addStep(Step.StepType.Scripting);
        step.setName("Fix Message Properties");
        // The old ffqueue component sets a couple message properties regarding the source file name.
        // The new zip aggregator requires a file name, but the property name is different.  So we
        // want to use the source file info to create the zip entry name.
        step.setExpression("SrcMessage.Properties(\"FileName\")"
                           + " = SrcMessage.Properties(\"DJFT SourceName\") & \".txt\"");
        step.getPresentationInfo().setX(386);
        step.getPresentationInfo().setY(120);
        return step;
    }
```

DataConnect includes a few older components related to file transfer that set and consumed some specific message properties all named "DJFT something". The ffqueue component is one of these, and one of the properties it sets is "DJFT SourceName" containing the base filename of the file in the message body. The zip aggregator component expects to find a "FileName" property, either as a step property or a message property. The step property isn't useful in this process, since the file name will vary with each message, so we need to include it as a message property. This step reads the "DJFT SourceName" property set by the ffqueue component, appends a ".txt" extension and sets that as the "FileName" property needed by the zip aggregator component.

Create and Configure the Write File to Zip Step (configureZipPutStep)

The *PutMessage* action is used to add a file to the zip being built:

configureZipPutStep method

```
    private AggregatorStep configureZipPutStep(ProcessRep process, Component zip) throws DesignSdkException {
        AggregatorStep step = (AggregatorStep) process.addStep(Step.StepType.Aggregator);
        step.setName("Write File to Zip");
        step.setComponent(zip);
        step.setActionType(ActionType.PutMessage);
        step.getAction().setParameterValue("Message", "SrcMessage");
        // We will put the files we are adding under the (relative) directory "test"
        step.setOption("TargetFolder", "test");
        step.getPresentationInfo().setX(210);
        step.getPresentationInfo().setY(120);
        return step;
    }
```

Again, we create the step (as a *Step.StepType.Aggregator*) in this case, set its name and component, and then set the action. The "Message" parameter is the message containing the file to add to the zip. Here we also set an action property (line 8) to specify to put the file in a "test" folder rather than at the root of the zip file.

This is the final step in the read-add-to-zip loop.

Create and Configure the Write Zip to Message Step (configureZipGetStep)

The final step we need to create is the one that assembles the final zip file and puts it in a message:

configureZipGetStep

```
    private AggregatorStep configureZipGetStep(ProcessRep process, Component zip) throws DesignSdkException {
        AggregatorStep step = (AggregatorStep) process.addStep(Step.StepType.Aggregator);
        step.setName("Write Zip to Message");
        step.setComponent(zip);
        step.setActionType(ActionType.GetMessage);
        step.getAction().setParameterValue("Message", "ZipMessage");
        // The zip aggregator by default encrypts the zip.  We want to turn that off.
        step.setOption("EncryptionMethod", "NONE");
        step.getPresentationInfo().setX(564);
        step.getPresentationInfo().setY(288);
        return step;
    }
```

Aggregator components use the *PutMessage* action to add an item to the aggregation, and (if appropriate) the *GetMessage* action to finalize the aggregation. This step performs the finalization and specifies putting the final zip file in the *ZipMessage* method. As noted, this component defaults to encrypting the contents. We don't want that, so we set the action-specific "EncryptionMethod" property to "NONE".

Macro Management (not implemented yet)

EZscript (not implemented yet)