Getting Started with Metamerge Integrator

2001.06.21

Copyright and Trademark Acknowledgements

Contents

Preface

About this manual

This book is a simple introduction to a simple system.

Make no mistake; we are using the word ‘simple’ in its most positive and powerful context, because the best way to wrap our minds around a complex problem is to break it down into simpler, more manageable bits and then master these constituent parts. Divide and conquer. And this applies equally to the problems related to engineering information exchange across an office, an enterprise or the globe.

Reducing the complexity of an integration problem means unraveling the intricate weave of information flows and examining it one thread at a time; Studying their direction and content, as well as which part of the patchwork they belong in. We all know why this is important: Understanding is measured in depth, and that means sticking your head below the surface to get a glimpse of how deep the rabbit hole really goes.

But this isn't much help if the integration tools we use don't let us express our understanding of the problem in such simple terms.

That's where the Metamerge Integrator comes in: Helping you preserve simplicity by giving you the tools to do three things:

- Implement your integration solution in terms of individual (and comprehensible) threads of communication between systems and partners.
- Deploy each thread rapidly, and with immediate feedback.
- Maintain and evolve your solution as your organization and the world around it changes.

Instead of imposing a new view on your information infrastructure, the Integrator gives you tools to construct the view your organization wants and needs.

The rest of this document is about tapping into the radical simplicity of the Integrator.

Installing the Integrator

Before you can install the Metamerge Integrator, you must first make sure that you have Java 2, version 1.3 or newer, installed on the machine.

The best place to get a copy of Java 2 is at

`http://java.sun.com/products`.

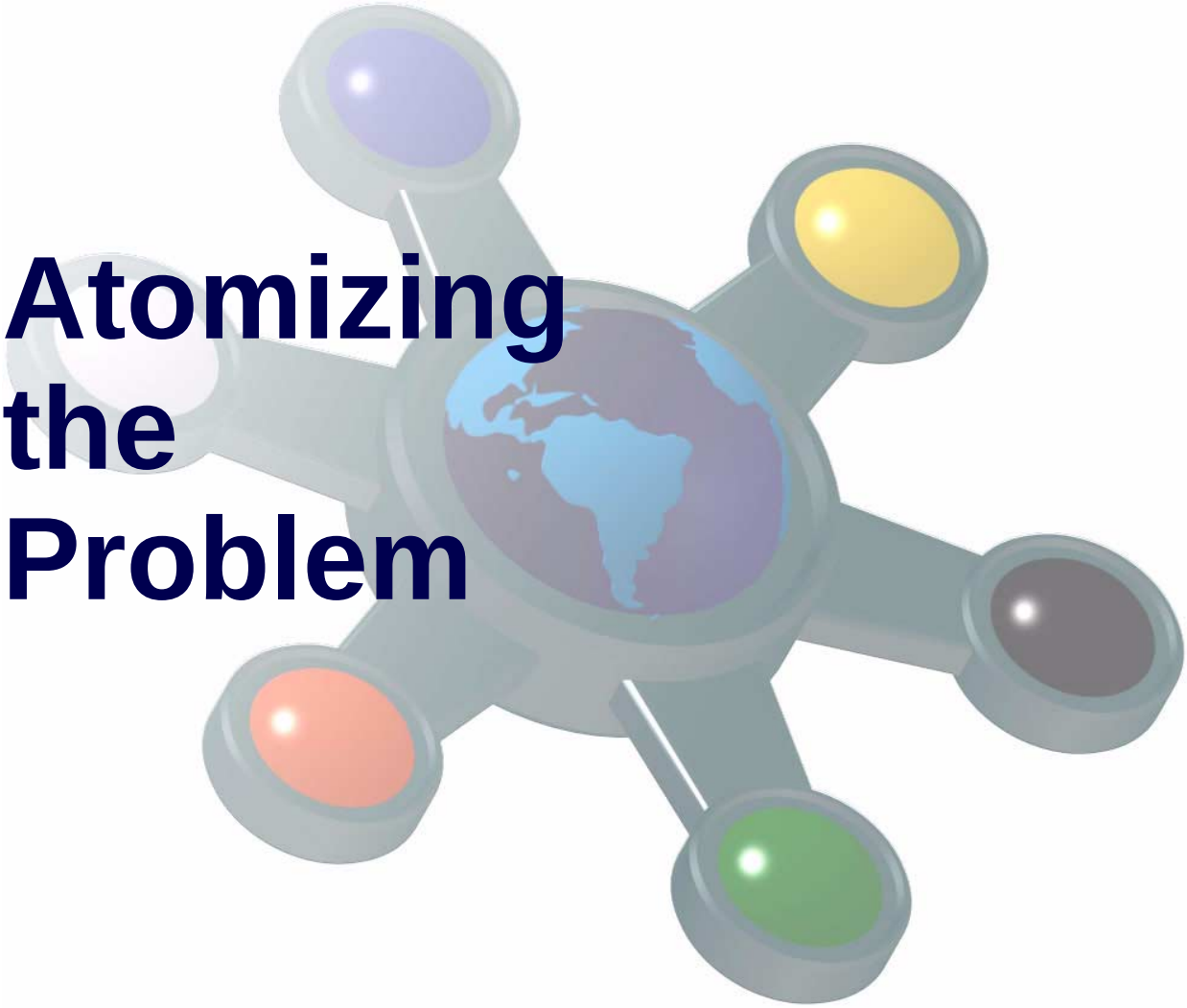
Unless you are planning on writing your own Java programs or applets, all you will need is the run-time.

Once you've got Java 2 in place, you can install the Integrator using the automatic installation program.

Be sure to read the release notes for the version you are installing. This is particularly important when upgrading an existing installation.

In addition, please make sure that you have the latest version of the Integrator installed. Otherwise some of the screenshots in this and other documentation may differ from those presented by your system.

Atomizing the Problem



Keeping it Simple

The first challenge in solving an integration problem is specifying exactly what needs to be integrated: which systems and devices are involved, what information they need to exchange and when.

This can be a difficult task, even if you feel that you have a good grasp of the situation. For example, try explaining the problem to someone who isn't as close to it as you are; Or to someone who is very close to only a few of the systems to be integrated. You'll probably discover a number of new angles and considerations that you were not aware of. What seemed up front to be a cut-and-dried integration project can suddenly begin to look very complex.

The key to success is to lessen the complexity by breaking the problem up into smaller, simpler pieces. *Atomize and implement*. The Metamerge Integrator gives you the tools to incrementally implement your integration solution, letting you literally grow your integration infrastructure one dataflow at a time, and giving you constant feedback on how the solution is evolving. By turn-

ing the implementation process into an exploratory one, the Metamerge Integrator will also help you to discover more about your own installation, evolving the integration solution as your understanding of the problem set grows.

A great way to get a good mental picture of the problem to be atomized is to make a picture of it. Grab a pencil and a piece of paper and sketch out a flow diagram that maps out the integration problem in broad strokes. This exercise will not only help you to understand the solution better, it will serve as the blueprint for implementing it in the Metamerge Integrator.

Integration problems are communication problems, and as such can typically be broken down into four parts: The systems and devices that make up the participants in the communication, when they're supposed to talk, what they should say to each other and how they should say it. In an integration setting, the constituent parts can be described as:

Table 1: Integration elements

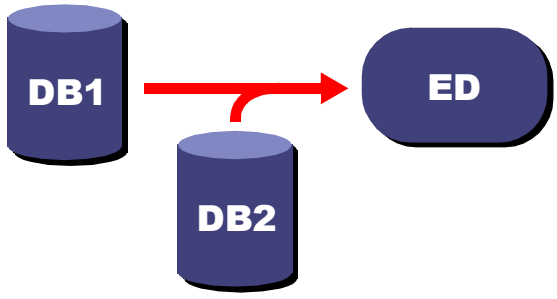
Data Sources	These are the data repositories, systems and devices that will be talking to each other. Like that Enterprise Directory you're implementing or trying to maintain, or your CRM application or the office phone system, or maybe that Access database with the list over company equipment and to whom it's been issued. Because the Metamerge Integrator supports a number of standard data access API's, like JDBC and ODBC, it can talk to most databases, including Oracle, Sybase, IBM DB2, Microsoft SQL Server and Access. And a database is just one specific type of data source. In fact, data sources can be a wide variety of systems or repositories, like a directory service (e.g. Exchange), an XML document, an LDIF or SOAP file, specially formatted email, or any number of interfacing mechanisms that internal systems and external business partners use to communicate with your company. It can even be a stream of raw data coming in over an IP port.
Events	Events can be described as the circumstances that dictate when one data source is to talk to another. For example, whenever a new employee is added to, or deleted from the HR system. Or when the access control system detects a keycard being used in a restricted area. Or a calendar/clock based timer, which starts communications at 12:00 midnight, every day except on Sundays. Or maybe it's a one-off event: like populating a directory from a collection of underlying data repositories.

Table 1: Integration elements (Continued)

Data Flows	These are the threads of the conversations and their content. They are drawn as arrows which point in the direction the data is to move, with each flow representing a unique message being passed from one data source to another. These flow arrows should also be labeled to indicate the actual data being communicated (e.g. employee name, phone number, date of employment, etc.)
Attribute Mapping and Transformation	It goes without saying that for a conversation to be meaningful to all participants, everyone involved must understand what is being said. But that doesn't mean that all data sources look at each piece of data the same way. One system might represent a telephone number as textual information, including the dashes and parenthesis used to make the number easier to read. Another data source might store them as numerical data. If these two systems are to talk together about this data, then the information will have to be translated during the conversation. Furthermore, the information may need to be aggregated with data from other data sources, and parts of the conversation may be directed at different receiving data source, resulting in branching arrows.

There are many diagramming conventions and styles available to choose from, but the actual shape and type of symbols is less important than your understanding of the problem. Use boxes or balls or bubbles or whatever you're comfortable with, but be consistent and be sure to label everything clearly and legibly. That way, when you look at your diagram in a couple of months, you will still understand it.

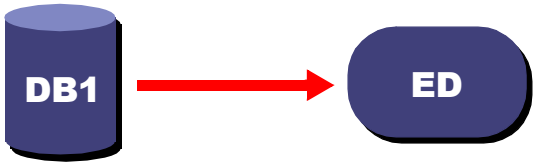
If we start by sketching out the data sources and data flows, then one example of a flow diagram (minus the message content labels) could look like this:



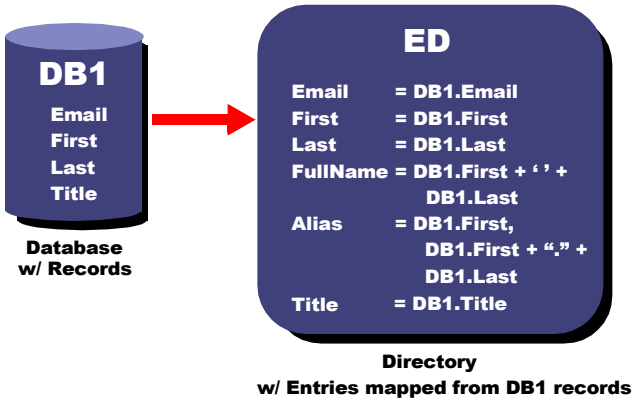
Here we see the Enterprise Directory (ED) getting data from the database (DB1). Along the way, the dataflow also picks up information from DB2.

The Metamerge philosophy is all about dealing with the flows one at a time: atomizing the problem. So let's take a closer look at the part of the flow arrow going from DB1 to the directory.

This flow describes the operation of populating the directory, filling it with information gathered from the database. We'll deal with the branch coming from DB2 later.



Now we need to detail the data that will be sent, as well as the circumstances for the transfer. Databases tend to view data as records that are made up of a number of fields. Directories on the other hand handle entities, each of which contains a number of attributes. In order to complete our visualization of the data flow, we will need to show how the fields of a database record are mapped (and possibly modified) to become the attributes of the directory.



Now that we have a good representation of our solution, let's take a look at the interface of the Metamerge Integrator administration tool.

The Integrator

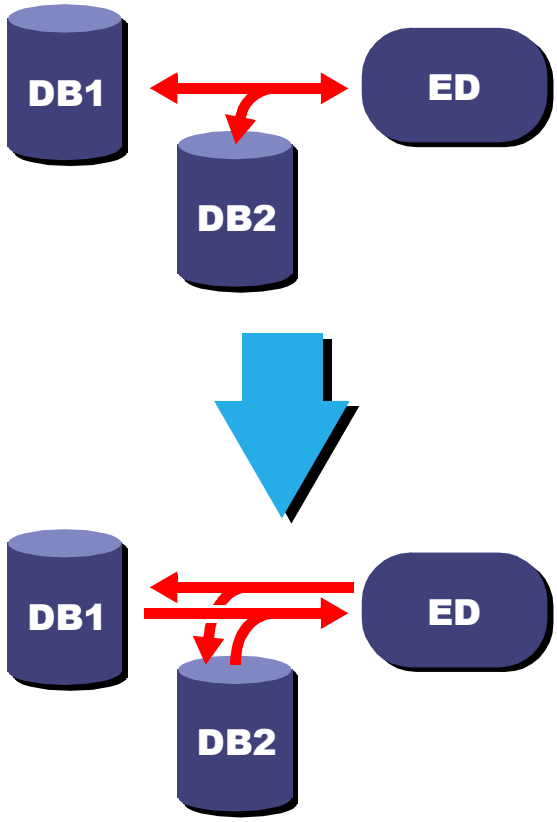
The data flow arrows in our diagram translate in the Metamerge Integrator to *AssemblyLines*, which work in a similar fashion to real-world industrial assembly lines.

> Real-world assembly lines are made up of a number of specialized machines that differ in both function and construction, but have one significant attribute in common: they can be linked together to form a continuous path from input source(s) to output.
>
> An assembly line generally has one or more input units designed to accept fish fillets, cola syrup, wood shavings - whatever the raw materials needed for production are. These materials are processed, often with by-products being extracted along the way. Finally the finished goods are delivered from the line in the desired form.
>
> If a production crew gets the order to produce something else, they break down the line, keeping the machines that are still relevant to the new order. New units are connected in the right places, the line is adjusted and production starts again.
>
> The Metamerge Integrator's AssemblyLines work in much the same way, except that at the end of the day, you don't have to clean out the leftover fish bits.

An AssemblyLine is made up of two or more processing units called *Connectors*, and each Connector is linked into either an input or an output data source.

It's important to emphasize that each AssemblyLine should be used to implement a single uni-directional dataflow line in our diagram. If we wish to support bi-directional synchronization

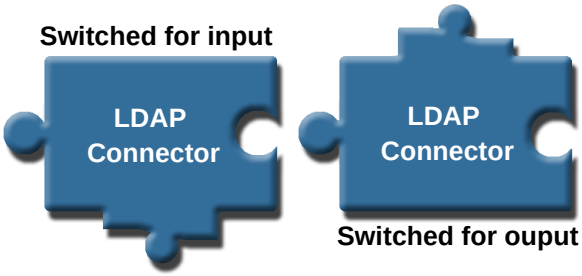
between two or more data sources, then we will want to use a separate AssemblyLine for each unique flow.



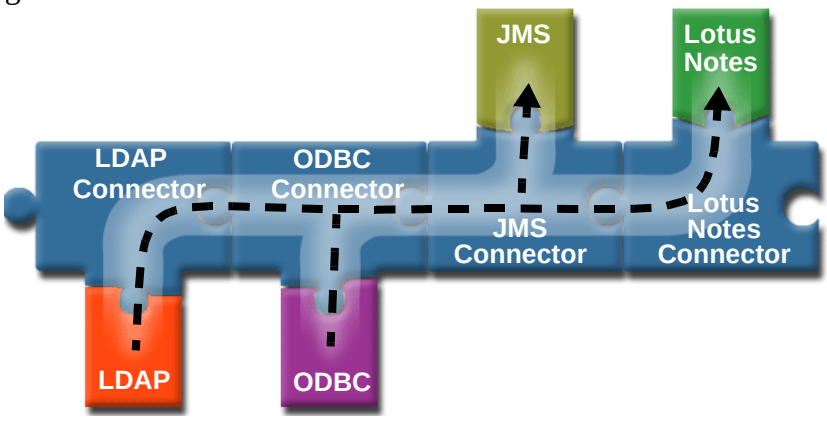
At the same time, our AssemblyLines should be made up of as few Connectors as possible—generally one per data source participating in the flow.

Connectors are like puzzle pieces that click into place with each other while plugging into a specific data source, like an SQL database, an LDAP-compliant Directory or a text file.

Each time you select one of these puzzle pieces and add it to an AssemblyLine, you chose whether it is going to be an input Connector, gathering information into the AssemblyLine, or an output Connector, inserting, updating or deleting data in the connected system or device[1].



Whenever you need to include a new data source to the flow, simply choose the relevant Connector, set it to input or output mode and insert it into the AssemblyLine where you want it to go.



The Metamerge Integrator gives you a library of Connectors to choose from, like LDAP, JDBC, Exchange and JMS to name a few. And if you can't find the one you are looking for, you can extend an existing Connector, overriding any or all of its functions, or even roll your own using one of the leading scripting languages, including JavaScript, Visual Basic and Perl. Or you can use Java to extend the objects of the engine itself, since the Metamerge Integrator is written entirely in Java.

The Integrator supports most transport protocols and mechanisms, like TCP/IP, SSL, HTTP and FTP, and it can access data through a multitude of API's and protocols: LDAP, JDBC, ODBC, JMS, plus many more, as well as databases and proprietary repositories such as Oracle, Microsoft SQL Server, Lotus Notes and Exchange. A Connector can even read unstructured data by pairing the Connector up with a *Parser* (see page 10 for more details on Parsers). The Metamerge Integrator is shipped with a variety of Parsers, like LDIF, XML, CSV and Fixed-length field, as well as giving you the tools to extend these or write your own.

In addition to plugging into data sources, Connectors are also the place where you put the business logic needed to reformat and re-contextualize the data that is being transferred (but we'll get back to that later.)

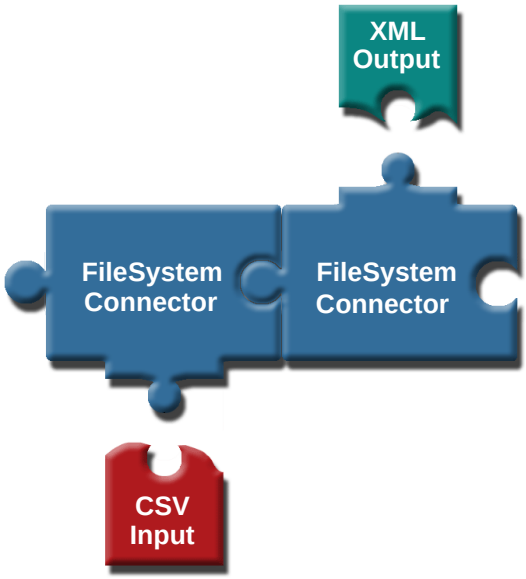
1. You might think that we've chosen to draw these puzzle pieces the wrong way: that data should be flowing in from above and then downwards to the receiving data sources. But anyone who has ever tried to implement an integration solution will testify that data doesn't tend flow on its own; it has to be sucked out of input sources and then pumped into the output sources. And that's what Connectors are good at.

In our first integration problem, we have two data sources (the database and the Enterprise Directory), so we will need two Connectors to implement the data flow. Looking more closely at the problem, we see that the Integrator will be reading from a file in comma-separated values format (CSV), which has been dumped from the database. We will update our directory by sending it a document in XML format that contains this data.

So the first step is to pick which Connectors to use.

Parsers

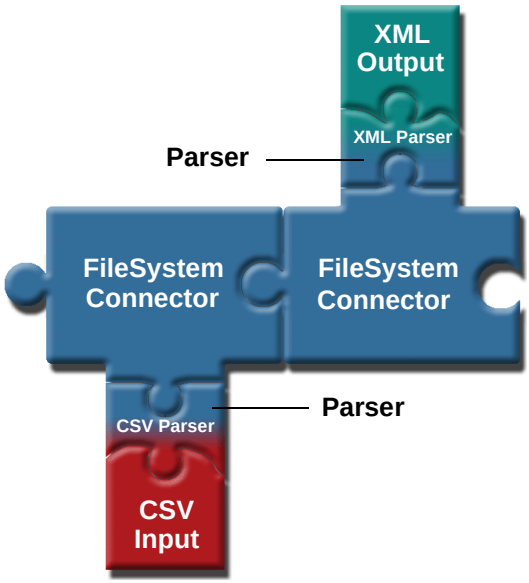
This is simple: Since both the CSV input file and output XML document are specially formatted flat files, then we will use a FileSystem Connector in each case. The CSV Connector will be switched to input mode, and the XML Connector will be switched to output.



But something is missing. FileSystem Connectors only read and writes raw bytestreams. They have no idea of how to format and structure the data that they read or write.

This is what a Parser does: it translates the data from the data source's format to the Integrator's own internal representation, and vice versa. So while some Connectors link into data sources

that have an implicit data structure, like a database or directory, all others must be matched up with a relevant parser.



Regardless of the way in which information flows (or is pulled) into the input Connector, and irrespective of its structure, once the data is inside the system, it is represented in a format that all Connectors can understand and use. Furthermore, this data is also made available to any custom scripts that are hooked into the AssemblyLine[2].

Connector modes

Up until now, we've spoken about input and output modes for a Connector. There are six Connector modes: two input modes (*Iterator* and *Lookup*) and three output modes (*AddOnly*, *Delete* and *Update*), plus one mode, named *Passive*, for Connectors that are only called as needed and are not part of the normal AssemblyLine execution.

Table 2: Connector modes

AddOnly *(output)*	AddOnly mode is an output mode, and a variation on Update. This mode causes the Connector to simply append to the output source, like adding to the end of an XML document, log file or CSV file.
Delete *(output)*	An output mode, this will cause the Connector to remove the specified record or entry from the system it is plugged in to. Of course, you can script it yourself to handle multiple record deletes.

2. Hooking scripts into AssemblyLines and Connectors is explained on page 39.

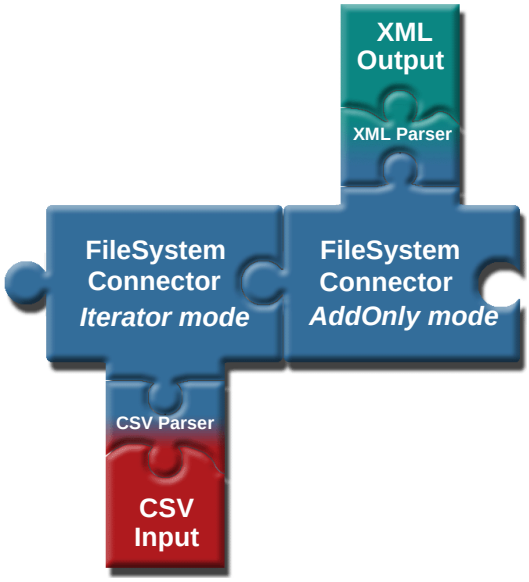
Table 2: Connector modes (Continued)

Iterator *(input)*	Connectors set to Iterator mode spin through an input source and evoke the AssemblyLine for each data entry returned. When the end of input is reached, the AssemblyLine stops, unless there are other Connectors set to Iterator mode in the AssemblyLine; Then each one is executed in turn.
Lookup *(input)*	Lookup mode means that the Connector will query a data source for a specific record, passing the data into the AssemblyLine for processing
Passive	Setting the mode to Passive will result in the Connector never being called during normal AssemblyLine processing. Instead, the must be activated through calls from scripts in the AssemblyLine, e.g. in another Connector. So what's the point then? Well, Connectors in Passive mode are used to handle exceptions, like writing status or error messages to a log database[a], or sending a message to the console, or even firing off an email to the systems administrator.
Update *(output)*	The Update mode causes the Connector to take the desired set of the data, transform it as specified and drop it into place in the destination system, while checking to see if new entries are to be added or existing ones updated.

a. The Metamerge Integrator already offers log file handling. However, if you want your log stored in a database, then you use the relevant Connector in Passive Mode

In order to solve our integration problem, we will need to set the input Connector to *Iterator* mode, so that it reads through the entire CSV file. The output Connector will be in *AddOnly* mode, since we are creating and then appending to a flat file. If we

update our diagram to include this new information, we get the following:



Now the final step is to take a look at how the data is to be mapped from the CSV file to the XML document, and what, if any, transformations will be necessary.

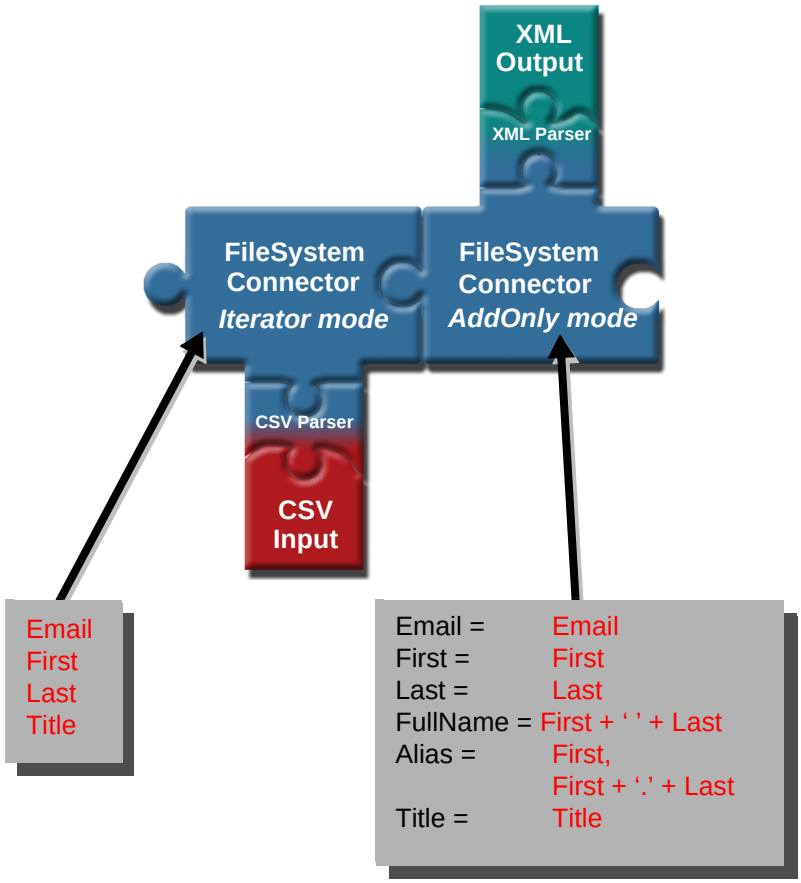
The CSV input data looks like this:

```
First;Last;Email;Title
Michael;Knagenhjelm;michael@metamerge.no;CEO
Bjorn;Stadheim;bjorn@metamerge.no;Chief Scientist
Johan;Varno;johan@metamerge.no;CTO
Skip;Skippy;;This should be skipped
```

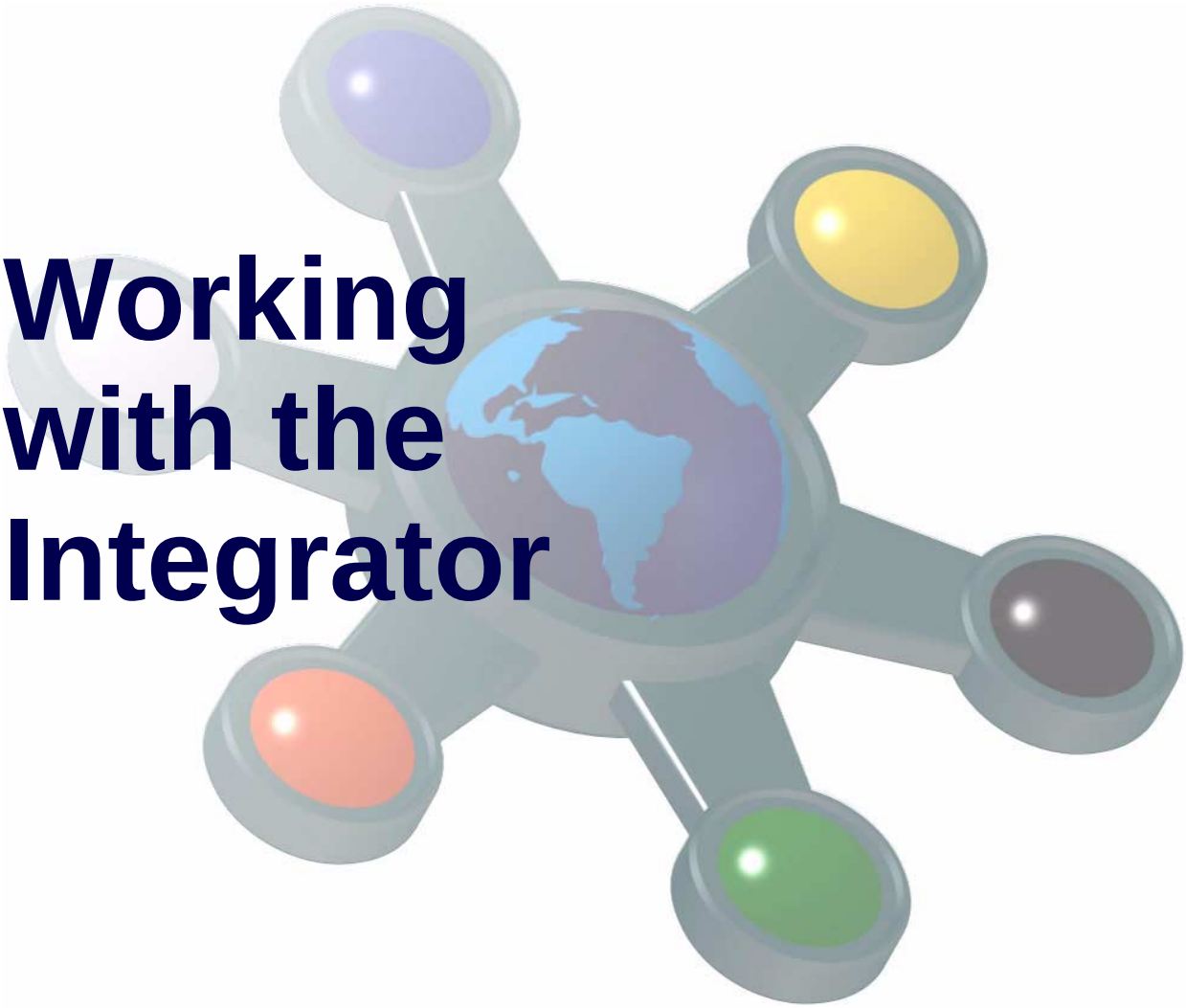
The first line describes the fields that are included in the file, while the following lines contain the data to be transferred[3]. This means that once the information has been pulled into the AssemblyLine, then the variables “First”, “Last”, “Email” and “Title” will be available for any Connectors and scripts that need to work with them.

3. Of course, if the field names are not in the file, or if we wish to override them, then you can specify this in the connector directly.

The XML document that we are going to write is to contain the following fields: “Email”, “First”, “FullName”, “Last”, “Alias” and “Title”. Let's add this to our diagram.



Now that our diagram is complete, it’s time to implement the solution in the Integrator.



Working with the Integrator

The Admin Interface

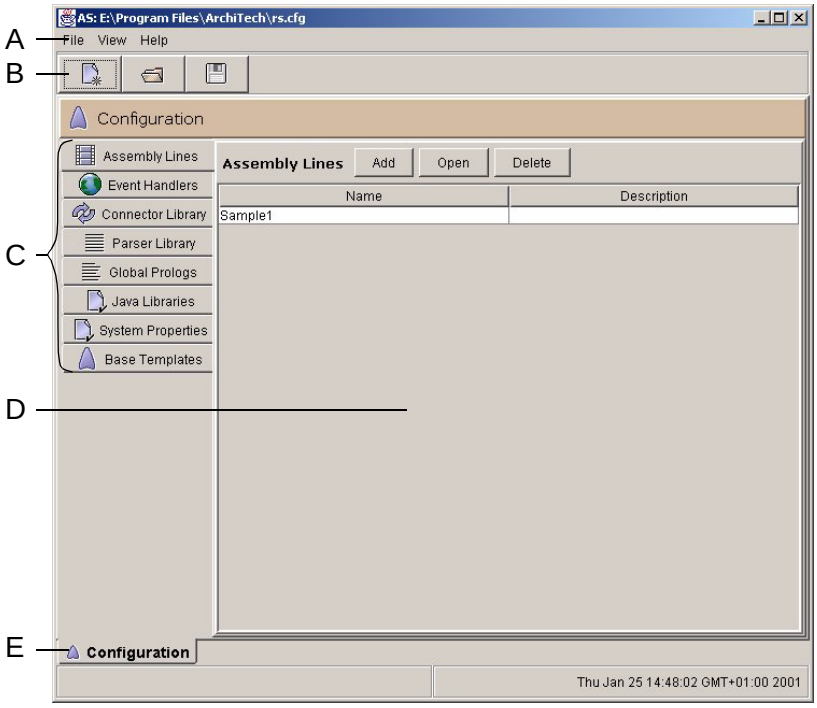
The Metamerge Integrator system is actually two programs:

- The Integrator Server, that reads the configuration file you create and then runs the desired AssemblyLines. This program file is called `MISERVER`.
- The Admin Tool, the program that you use to configure the Integrator, as well as for testing and debugging your AssemblyLines. The Admin Tool program is called `MIADMIN`.

Like the Integrator Server, the Admin Tool is also written 100% in Java, and runs in any environment that offers a Java 2 compliant Virtual Machine.

If you haven't already started the Admin Tool, do so now.

You will be presented with the main screen:

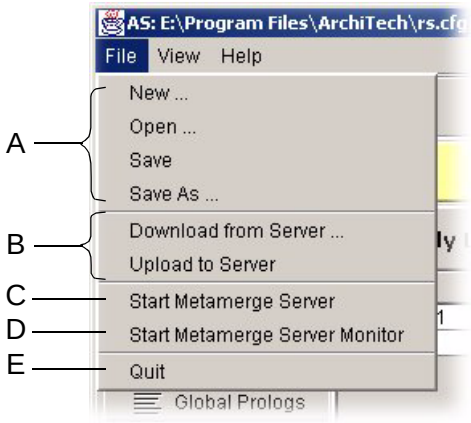


A The *Main menu* provides pull-down menus for a number of functions, like opening and saving configuration files, viewing registered certificates and changing the look and feel of the Admin Tool.
B The *Button bar* provides shortcuts to the commands to create a new configuration file, open an existing one or save changes you've made to the currently open one.
C *Object tabs* are used to select the various types of objects in the Integrator (like Connectors and AssemblyLines) for setting system parameters, as well as for accessing libraries of Connectors, Parsers and scripts.
D The *Display area* is where the information specific to the current object you are working with is displayed and edited.
E *Context tabs* are for switching between different objects that are open. These tabs line up along the bottom of the Display area.

You may also have noticed that the Display area already shows an AssemblyLine called Sample1. This is because the Integrator was started with the command to load the `rs.cfg` file, the default configuration file (if you look at the title bar of the Admin Tool window, you can see the name of the configuration file currently open.)

However, we are going to add our first AssemblyLine manually ourselves in order to stretch our legs a bit with the system. But before we get into the meat of the implementation, we'll take a closer look at some of the main elements of the interface.

The Main menu has three pull-down menus: **File**, **View** and **Help**. The **File** menu the following selections:



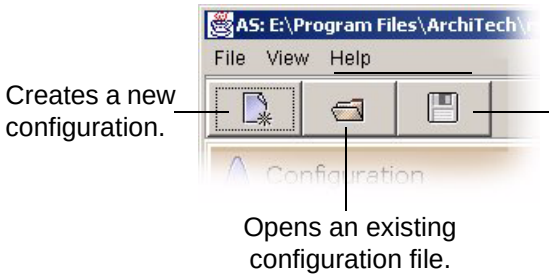
A Here are the standard File menu selections for creating a new configuration file, opening an existing one, saving changes, as well as saving the current configuration to a new filename.
B These two selections are for opening and saving a configuration file for a server on a different machine that has been configured to allow remote administration.
C Here is where you start the server and execute your AssemblyLines.
D This next selection is to start a monitor window for a remote server
E Exits the administration program.

If you click on the **View** menu, you will get the following options:

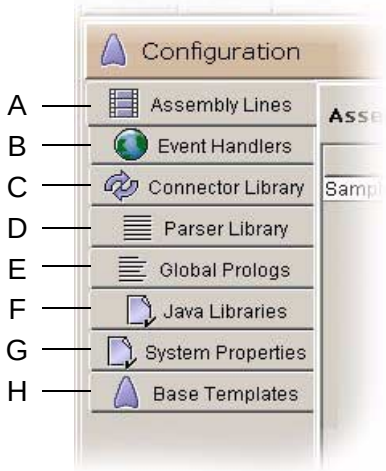
F Creates a printable report describing the currently selected AssemblyLine.
G Here is where you control information about any certificates that are needed by your connectors.
H This selection gives you three Look and Feel (L&F) options for the Admin Tool: *Windows*, *Metal* and *Motif*.
I Opens the debug monitor window.

Under the Help menu you will find only the **About** selection.

Just under the main menu is the Button bar, with shortcuts for creating new configuration files, opening existing ones and saving changes to disk.

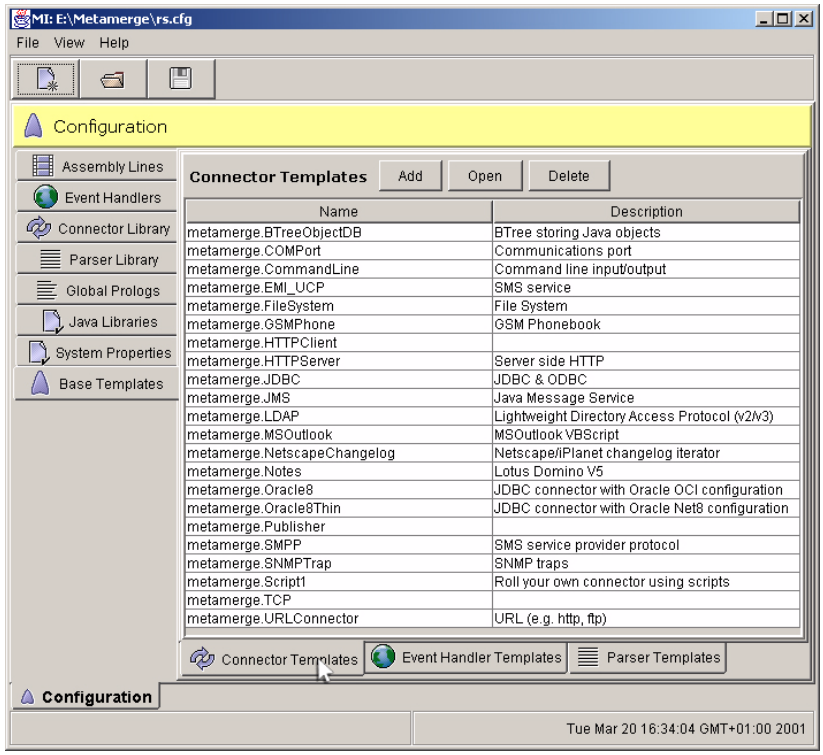


Down the left side of the screen are the Object tabs:



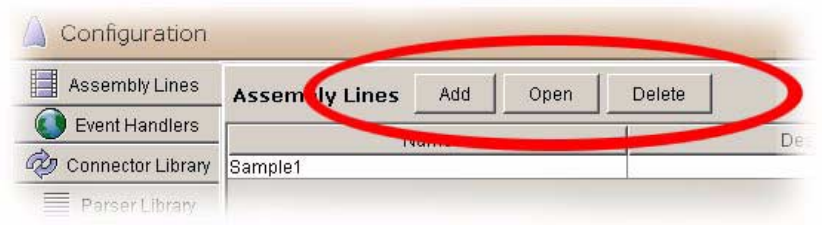
A This tab is for creating and managing your AssemblyLines.
B Here you can work with EventHandlers. These are objects that can be configured to wait for events—like incoming mail, or a trigger in a database—format the incoming data, if necessary, and then fire up one or more AssemblyLines to handle this information. EventHandlers are not covered in this manual.
C Click here to access to your own library of Connectors. These have been either written by you, or inherited from one of the basic ones and configured or extended to meet your needs.
D Where you will find the collection of Parsers that you've written or extended from the base templates.
E Here you write the script that are to be evaluated first, before AssemblyLines are started. Global prologs can be used to register global variables that you will be using, like counters and accumulators for keeping track of statistics or other status information**.**
F Any Java classes that you want to make available to your Connectors and scripts are kept here.
G This tab allows you to access Java VM parameters.
H All the base templates for Connectors, EventHandlers and Parsers are found here.

The area to the right of the Object tabs is the *Display area* and this part of the screen changes as you select different Object tabs, giving you a list of objects to work with.



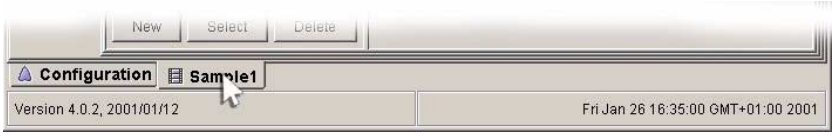
The screenshot above shows the list of Base Templates. Unlike the other lists, the Base Templates screen presents you with a set of tabs at the bottom of the list. This is because these templates come in three flavors: Connector Templates, EventHandler Templates and Parser Templates. You can get the list you want by clicking on the relevant tab.

Like the other object lists, Base Templates also has a set of buttons at the top of the Display area. These are **Add**, **Open** and **Delete**, and they are available for all objects except for Java Libraries and System Properties, where you only get **Add** and **Delete**[4].



4. If you want to edit a Java Library then you must do so in a Java development tool or editor. The Java Library list is simply a reference to the libraries (packages) that you want to include.

If you open an Integrator object (like an AssemblyLine, an EventHandler or a Connector) by either double-clicking on the desired object, or by selecting the object and then clicking the **Open** button, the Display area changes to show the details of the selected object. In addition, a new tab is added to the Context tabs at the bottom of the screen. All you have to do is select the Context tab that you want to work with to move from one to another.



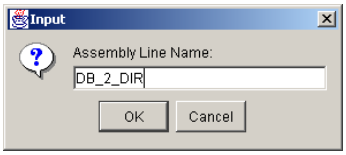
In addition, the Context title (which is located at the top of the Display area just under the Button bar) also changes to show you what you are looking at, as well giving you a **Close** button to remove this context tab from the display.

Often when you open up a detail view, it covers the entire Admin Tool. Don't panic: You can always go back to the previous screen by clicking on the Context tab, or by pressing the **Close** *button as shown above.*

Now that we've taken a look at the main interface elements, it's time to add our own AssemblyLine.

Adding AssemblyLines

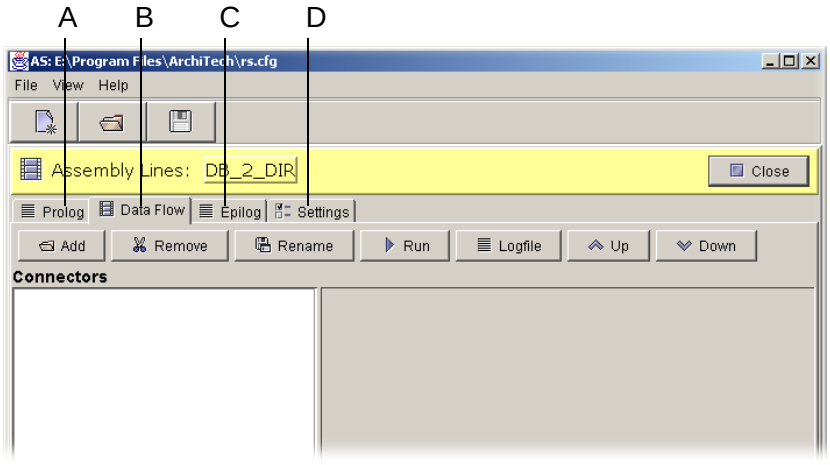
The first step is to select "AssemblyLine" tab from the Object tabs at the left of the screen. Then press the **Add** button at the top of the Display area. Enter the name "DB_2_DIR" in the dialog that appears.



Now there should be two AssemblyLines in the list shown in the Display area.

Select the one entitled "DB_2_DIR" and press the **Open** button above the list. This will add a "DB_2_DIR" Context tab at the bottom of the screen and show you an empty AssemblyLine screen.

If you look just below the Context title, you will see a set of tabs for the AssemblyLine, and a row of buttons below these. These tabs present you with the various aspects of an AssemblyLine that you can configure, like adding Connectors, or writing a Prolog or Epilog script.



A “Prolog” is for the script that is to be run first each time the AssemblyLine is to be run. Note that the AssemblyLine Prolog script is run after any Global Prolog scripts have been evaluated.
B The "Data Flow" tab shows you the list of Connectors attached to this AssemblyLine.
C “Epilog” is for the script that is fired off at the end of AssemblyLine execution.
D Here you will find a number of general parameters that control how the AssemblyLine operates.

Below this tab bar are the buttons used to **Add**, **Remove** and **Rename** Connectors, as well as changing the order that they will be executed by moving them **Up** or **Down** in the list. There is also a button labeled **Run** for executing our AssemblyLine, and another for viewing the **Logfile** created when this AssemblyLine is run.

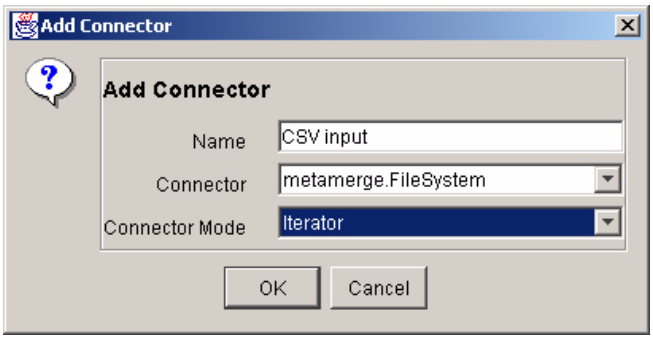
However, we are going to leave the “Prolog”, “Epilog” and “Settings” tabs alone, selecting instead with “Data Flow” tab where we’ll add the first Connector to our AssemblyLine.

Adding Connectors

As mentioned in the previous section, Connectors are the workhorses of an AssemblyLine, and we will need to add a Connector for each data source that we are reading from or writing to.

Our first Connector will be for reading in the CSV input file. That means we need a FileSystem type Connector. We’ll set it to *Iterator* mode and then pair it up with a CSV Parser

With the “Data Flow” tab selected, press the **Add** button to create the first Connector.



Enter “CSVinput” for the name. This is just a descriptive name, but for the sake of clarity and maintainability, you will want to be sure that you can identify your Connectors from their titles.

Then in the Connector drop-down list, select the connector type called **metamerge.FileSystem**. This drop-down listbox contains all the Base Connector Templates, as well as any Connectors that you've already added to this AssemblyLine[5] or to the Connector Library.

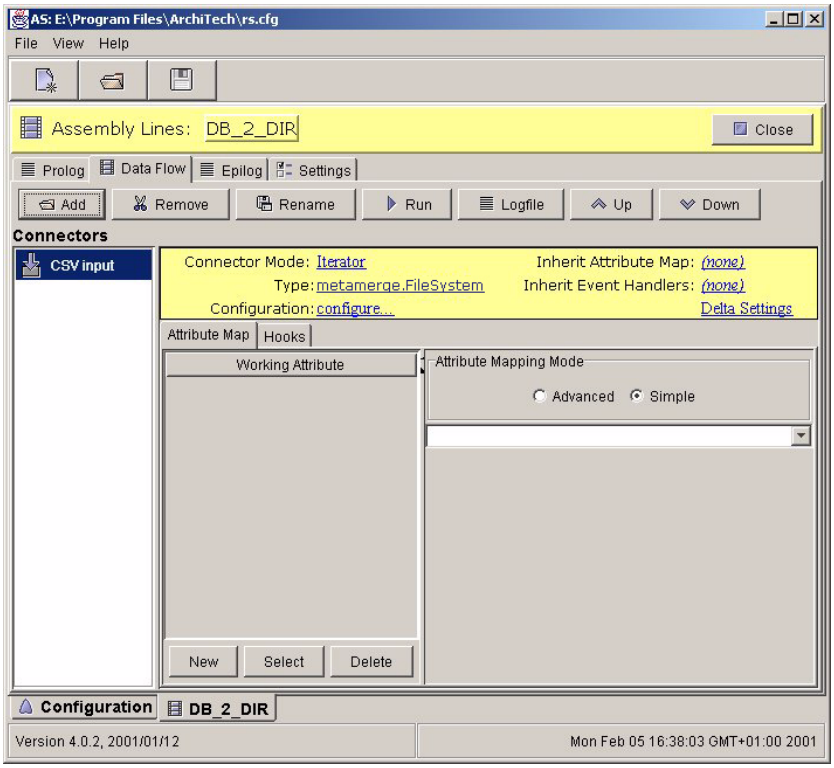
In the last drop-down listbox, labeled “Connector Mode”, select *Iterator*.

Once you've filled out the dialog, click **OK** to accept the new Connector.

5. You might be wondering why the other Connectors in this AssemblyLine should appear in this drop-down list. Why would you want to inherit from another Connector in the same AssemblyLine? One of the reasons might be to send your output to the same communicator that you are getting your input from.

   For example, an AssemblyLine for processing information requests coming over an IP port would receive a query, look up the information in other connected systems and then pipe the results back to the same port. Although this might seem pretty straightforward, problems arise when you consider the fact that each Connector opens and “owns” a connection to its data source. If that data source can only service a single process at a time (as in the IP port example above) then the output Connector would not be able to open its own connection to return the result data. By inheriting the output Connector from the one doing the input, both share the single connection and no conflicts arise.

The screen should now look like this:



As you create new Connectors, they appear in the list on the left side of the screen. Just above the list is a row of buttons for *adding*, *removing* and *renaming* Connectors, as well as one for *running* the AssemblyLine that contains this Connector.

There is also a button for opening the Logfile window that displays messages generated from the last run of the AssemblyLine.

Finally, there are two buttons for moving the selected Connector up or down in the AssemblyLine (Note: if you can't see all the buttons, simply resize the window until you can.)

The order in which Connectors appear in the Assembly Line is significant, in that they are executed from top to bottom. Furthermore, the flow of information is also from top to bottom, so that a Connector has access to the data handled by the all Connectors that appears before (above) it in the AssemblyLine.

Now take a look in the yellow band at the top of the Display area. This is the Connector header. Here you can find a number of parameters for this Connector: mode, type and configuration, plus three values in the column to the right.

The top two values are for controlling how this Connector inherits Attribute Mapping and Event Hooks (this is a little typo in the interface which will be corrected in the next release). The third value varies a bit for the various Connector modes:

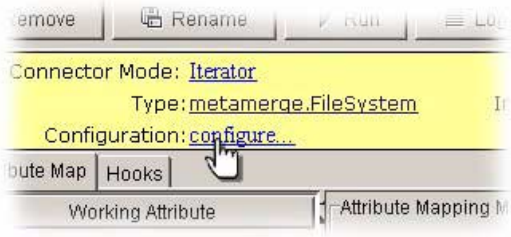
For *Iterator* mode then this value is **Delta Settings** (as it is now). If you click on this field then you get a dialog where you can tell the Integrator to keep track of records read from the data source and then quickly identify if new entries have been added or existing one have been deleted. This is a great feature if you are looking at a log from an HR system and want to know when employees have been added or deleted.

For *Lookup* mode, then this field is called **Allow Duplicates** which can be turned on or off.

Finally, for *Update* mode, it becomes **Compute Changes**, telling the Integrator to lookup data before writing to determine if there have been changes. The Integrator then will only write to the data source if changes are detected. This can be important when writing to systems that will immediately generate enterprise synchronization on an update, like Lotus Notes.

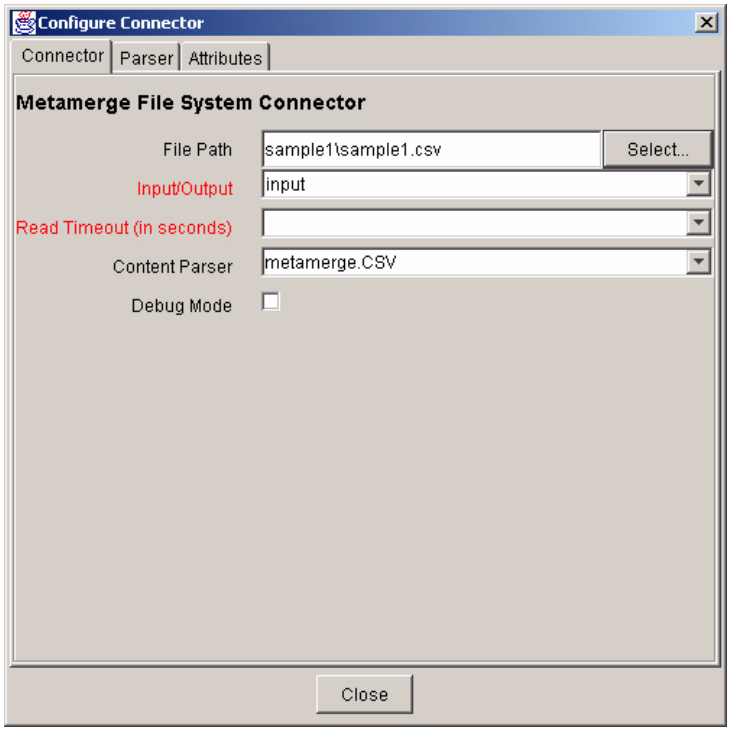
Configuring Connectors

If you move the mouse over one of these underlined values, you will see that the arrow turns into a hand (just like in a web browser). These fields are links that open dialogs for modifying the underlying values when you click them.



We need to configure this Connector to specify where the input file is found, as well as which Parser to use. So, click on the underlined word **configure** in Connector header, or double-click on the “CSVinput” Connector in the list to the left.

The Configure Connector dialog will appear, showing you the parameters available for this type of Connector.



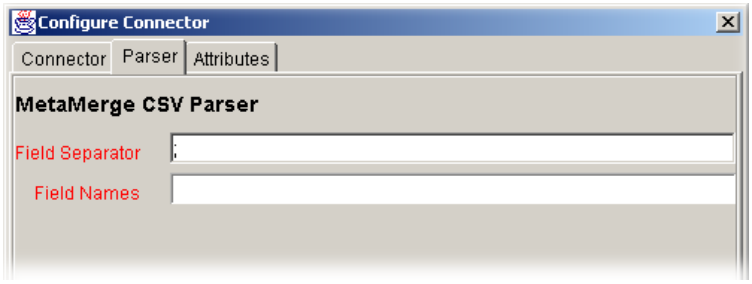
Type in the name and location of the input CSV file, as shown above. You will find the example data file in the `Sample1` sub-catalog of the directory where the Integrator was installed.

Notice how some of the fields in the dialog are red while others are black. Red text indicates that the value of this parameter has been inherited from another Connector; either one in this AssemblyLine, the Connector Library or from a Base Template Connector. Whenever you override an inherited parameter by entering a value in the edit field, then the text label turns black. This color coding applies to other integration objects as well, like EventHandlers and Parsers.

Finally, choose the CSV parser (**metamerge.CSV**) from the drop-down list over Content Parsers.

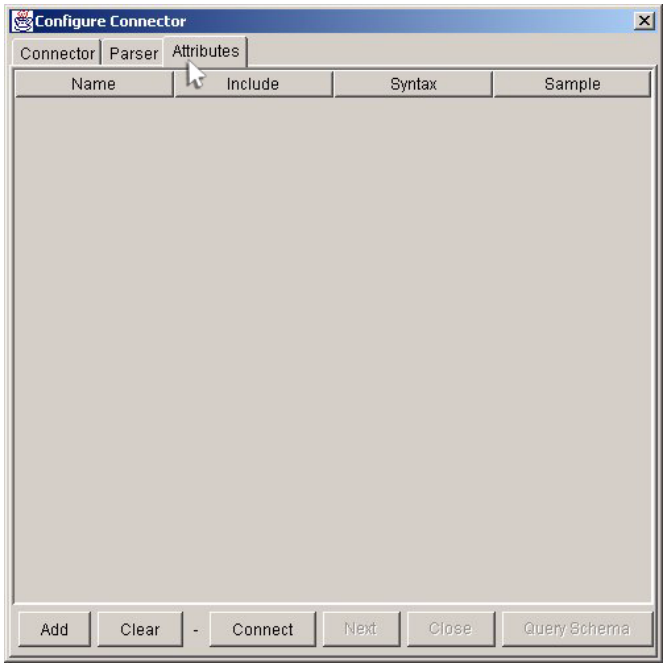
Configuring Parsers

In the Configure Connector dialog select the tab labeled “Parser” at the top of the dialog and you will get the parameters available for this type of Parser.



For a CSV Parser, this is the field separator (e.g. comma, semi-colon, etc.) and, optionally, the field names. If you remember back on page 9 where we looked at the CSV input data, the first line contained the field names. If the CSV file you are reading does not contain the field names in the first line, or if you want to override these, then you must enter the names in the Field Names field. The list should use the same separator as specified in the Field Separator field. Without some specification of the field names, then the Integrator will not be able to structure the data correctly.

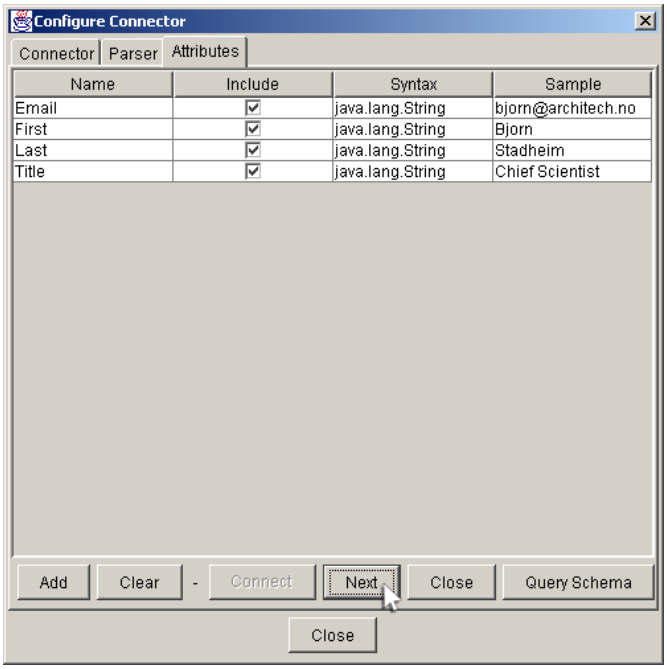
The last tab in this dialog is the "Attributes" tab, which gives us a screen for testing our connection and viewing the data coming in from the data source.



In order to test the connection, simply press the **Connect** button. This causes the Connector to make contact with the data source

and retrieve information on how the data is structured. If all goes well, then the **Next** button will be enabled.

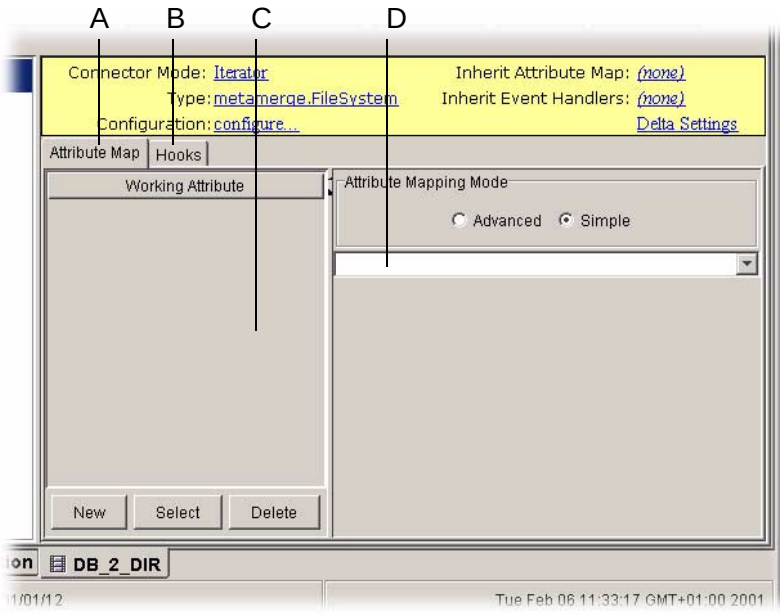
Each time you now press the **Next** button, the Connector reads a line from the input file, fills attributes with data and displays them in tablular form.



We can see from the results above that the Connector is working. These are the attributes that will be available later for mapping into the AssemblyLine, and you can remove any or all of them from list by clicking away the checkmark in the “Include” column.

Close the Configure Connector dialog when you’re finished.

It’s now time to take a closer look at the Connector Display area, and in particular, at how to set up attribute transformation and mapping.



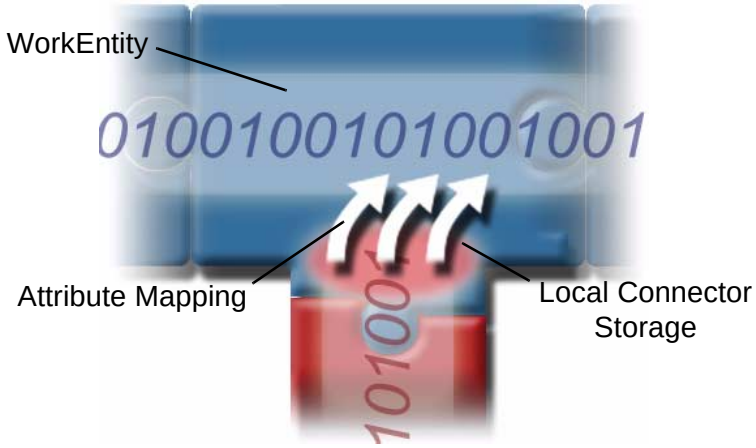
A "Attribute Map" is the tab for selecting which attributes to map.
B The "Hooks" tab (which we'll look at more closely on page 39) is for extending the Connector with scripts.
C The attributes we select for mapping appear in the "Working Attribute" list.
D This is where we specify the mapping for an attribute, plus any transformations (like converting numbers to strings, or vice versa).

In order to understand how attribute mapping works in the Metamerge Integrator, it's first necessary to make the acquaintance of the container object that actually carries data down the AssemblyLine: the *WorkEntity*.

The WorkEntity starts off completely empty at the beginning of the AssemblyLine, and we have to create attributes and map over values from an input Connector in order for the data to be included in AssemblyLine processing. Conversely, we have to map that data back into any output Connectors so that it is available for making changes in the data source.

Looking under the hood; When an input Connector reads in data, this information is actually stored in attributes that are local to

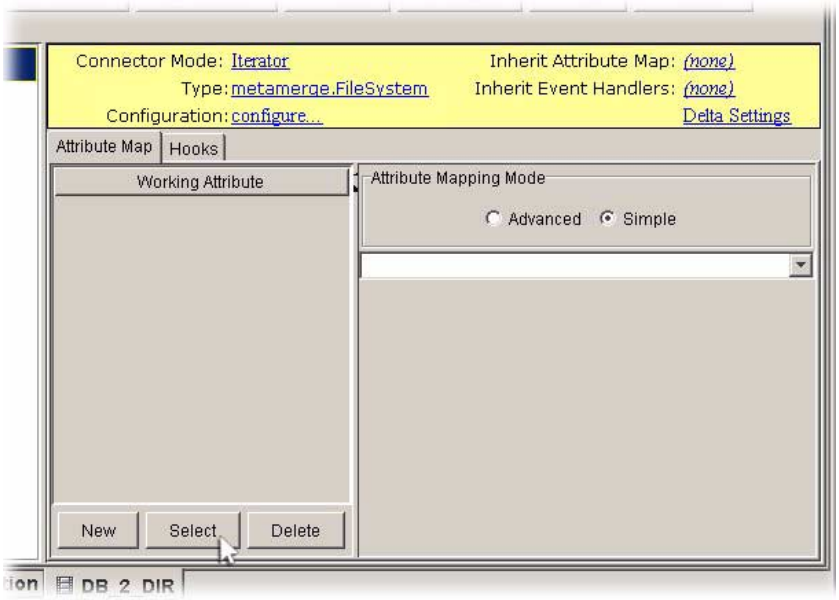
the Connector itself, and which disappear when the Connector is finished working.



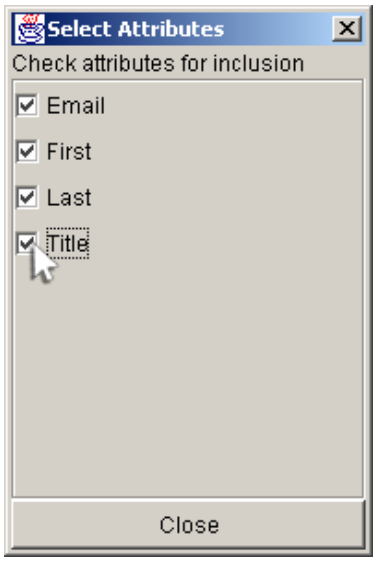
In order for this data to survive the Connector, and be passed on down the AssemblyLine, it must first be copied from the Connector's local storage to attributes that are created in the WorkEntity. All scripts and functions that work with the data, must necessarily reference these attributes in the WorkEntity.

And finally, when the dataflow reaches an output Connector, the data must first be mapped back from the WorkEntity to the output Connector's attributes before it can be written to the data source.

The simplest way to create new attributes for the WorkEntity, and get their mapping automatically in place, is by pressing the Select button at the bottom of the Working Attribute list.



If the Connector is able to open a connection, and if the data source can return the names of the fields being read (or you have specified them yourself in the Connector Configuration dialog), then you will be presented with a list of available attributes.



Simply click the checkbox next to the fields you want mapped from the Connector to the WorkEntity.

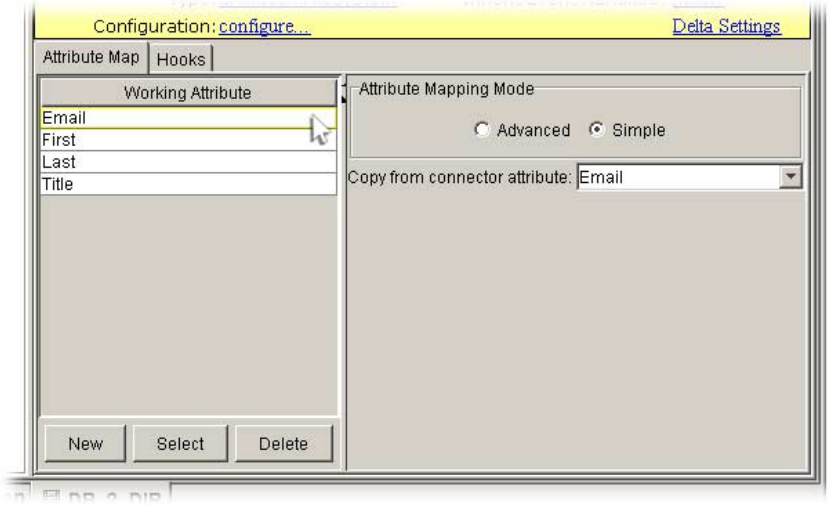
If for some reason the Integrator cannot read the attribute names from the data source, or if you want to create new attributes that will contain computed values, simply use the **New** button at the bottom of the Working Attribute list, and type in the names of the attributes you want to add.

Attribute Mapping

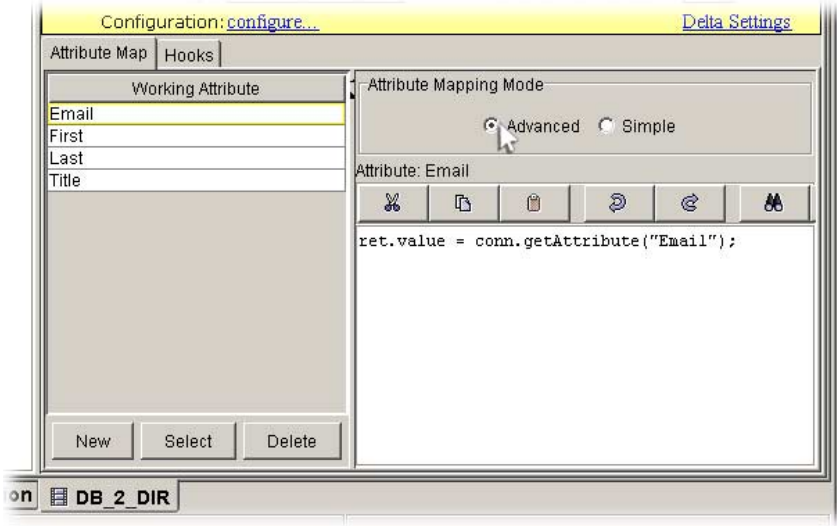
Attribute mapping can be *simple* or *advanced*.

Simple mapping means that the value in the Connector attribute is copied unmodified over to the corresponding attribute in the Work Entry.

Just click on one of the entries in the Working Attribute list and you will see that simple mapping is the default for attributes selected from the input Connector's list.



If you want to see the actual script which is created and run by the Integrator to handle the mapping of this variable, click on the **Advanced** radio button.

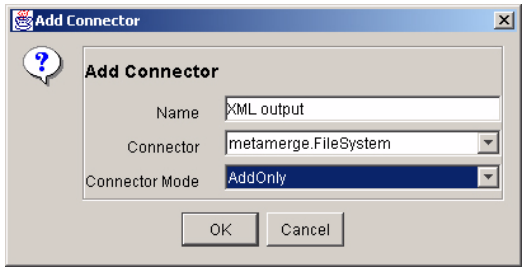


As you can see from the screenshot above, the value returned for the selected attribute is read from the **conn** object (which gives you access to the Connector's attributes and functions).

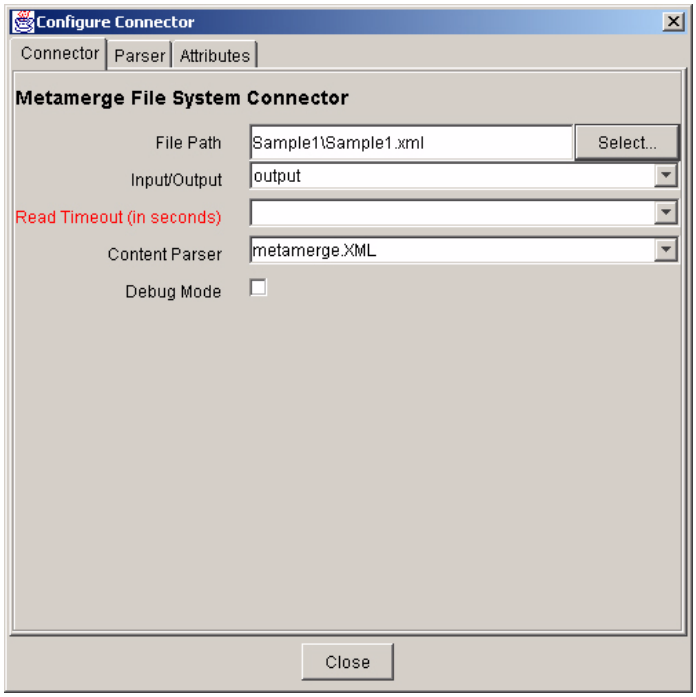
The Integrator Admin tool gives you a simple editor window in which to write your scripts. Here you'll find buttons for *cut*, *copy* and *paste* operations, as well as *undo* and *redo*. The last two buttons (you may have to resize the window to see them) are for *searching* and *repeating the last search*.

With the input Connector in place, all we have to do now is add an output Connector, and then our first AssemblyLine will be ready to test.

You do this in the same way that you added the input Connector. Choose a FileSystem Connector type here as well, but this time coupled to an XML Parser.



As with the input Connector, you can open the Configure Connector dialog by either double-clicking on the new entry in the Connector list, or selecting the desired Connector and then clicking on the underlined word **configure** in the Connector header.

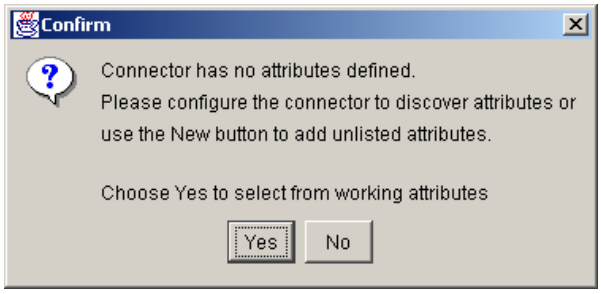


Again, choose the `Sample1` directory for the file path of the `Sample1.xml` output file, and choose “output” in the second drop-down. This will cause the Integrator to create a new XML file each time, while “append” would make the system append to the end of an existing one.

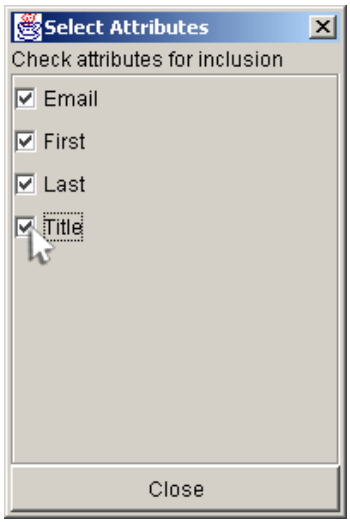
Now we have to map the variables from the WorkEntity into attributes in the output Connector so they can be written to the

XML document. Once again we will use the **Select** button at the bottom of the. Did you notice how the list is now labelled “Connector Attribute List” instead of “Working Attribute List”? This is because we are mapping the other way this time: into the Connector.

But when we press **Select**, we get the following message:



By clicking on Yes, we are asking the Integrator Admin to present us with a list of WorkEntity attributes so that we can select from these. In return, we get a dialog that looks very similar to the one we used to choose attributes for mapping to the input Connector.



Make sure all the fields are checked for inclusion and press the **Close** button.

Now we know that the output Connector will take all the attributes mapped over from the WorkEntity and write them to the XML file. But our job is done yet.

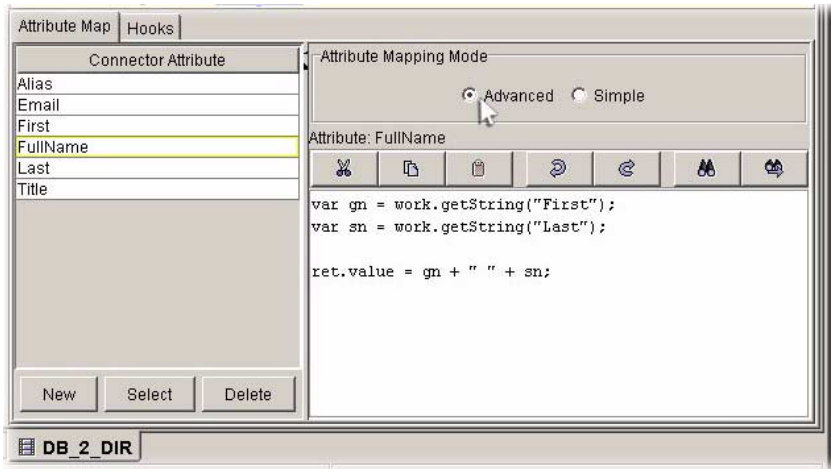
You may recall that back on page page 6 we specified two additional attributes: “FullName” and “Alias”. Since these attributes are not available directly from the WorkEntity, we must create them ourselves. This is done by pressing the **New** button at the

bottom of the Connector Attribute list. You will be presented with the following dialog.

Simply enter the name of the new attribute and press **OK**. Repeat this procedure for “Alias” as well.

Once we have created the attributes “FullName” and “Alias” we have to set up the mapping and transformation necessary to create their values.

Selecting “FullName” first, choose advanced mapping mode and enter the script shown below.

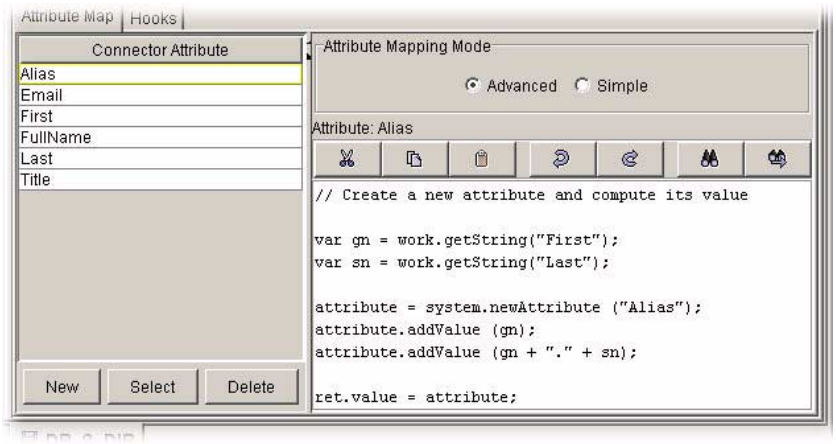


The script in the screenshot is a bit tiny, so here is a more legible copy:

```
var gn = work.getString(”First”);
var sn = work.getString(”Last”);

ret.value = gn + ” ” + sn;
```

Without going into the mechanics of JavaScript here, this snippet defines two variables called “gn” and “sn”, assigns to them the values of the Working Entity’s attributes “First” and “Last” (converted to string variables). The script then sets the return value—which will be placed in the “FullName” attribute in the example above—as a single string value equal to the value of the two attributes read from the WorkEntity joined together with a space character between them.

Now click on the Alias attribute and enter a slightly longer, but just as simple a script.



This JavaScript code snippet starts off similar to the previous one used for the “FullName” attribute:

```
var gn = work.getString(”First”);
var sn = work.getString(”Last”);

attribute = system.newAttribute(”Alias”);
attribute.addValue(gn);
attribute.addValue(gn + ”.” + sn);

ret.value = attribute;
```

Notice how it then creates a new attribute (which is local to this script) and assigns it two values. The script then returns the new attribute itself, instead of simply returning a string value as we did for “FullName”.

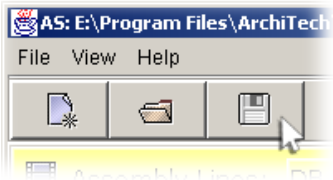
An attribute can hold any number of values if you need it to—and you probably will. A typical example is the “Email” attribute, which could be made to hold several address values. Then we could connect into the data source(s) where the other email addresses were stored and aggregate them into the “Email” attribute in our AssemblyLine.

Running an AssemblyLine

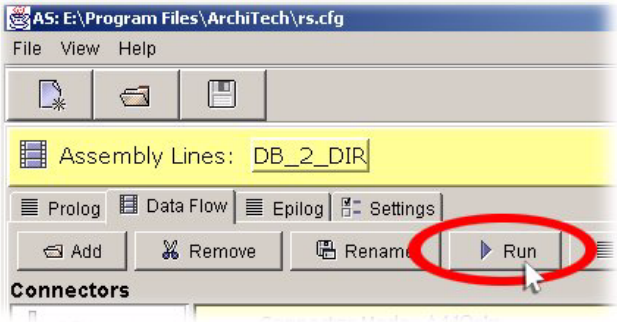
Once the attribute mapping is completed, we are ready to test our new AssemblyLine. This can be done directly from the Admin Tool.

You must first save your configuration before you can test it. Remember: you are in the Admin Tool. The changes you are making must be saved to the configuration file before the Integrator Server can get access to them.

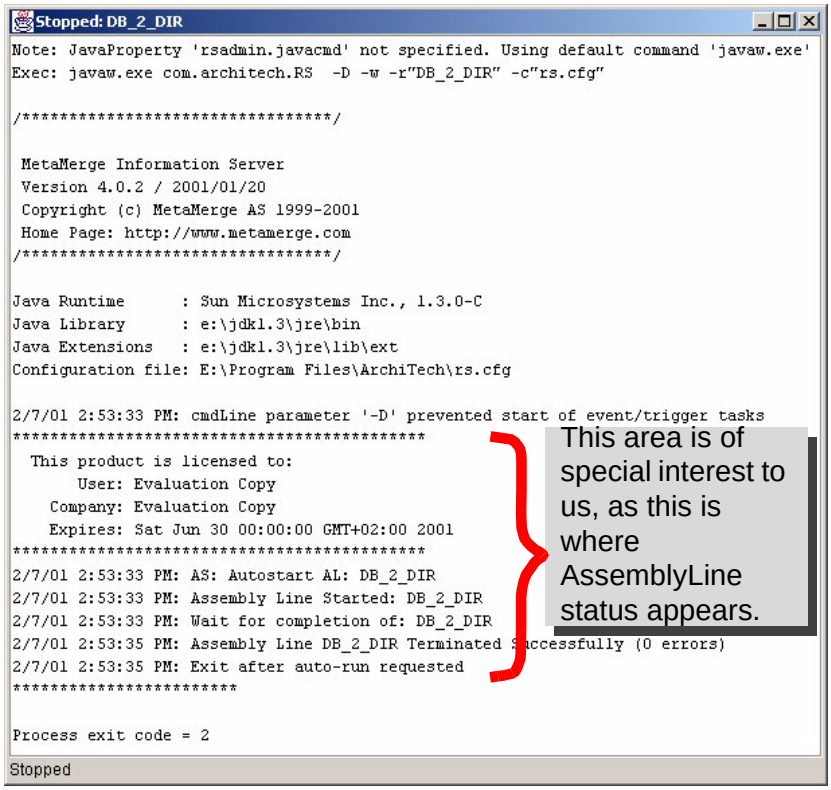
Saving your configuration can be done by either selecting **File|Save** from the Main menu, or by pressing the **Save** button on the Button bar.



Once the configuration file is saved, click on the **Run** button to execute the currently selected AssemblyLine.



You will get a monitor window that displays messages from both the Java VM and from the Integrator Server.



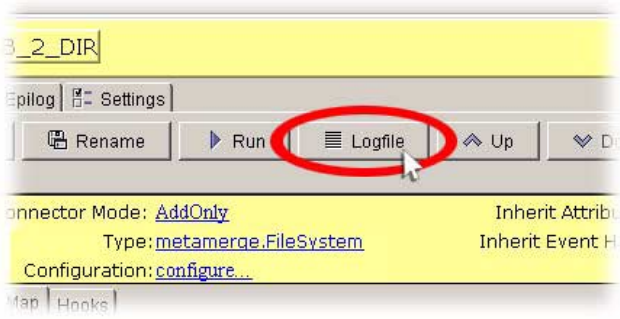
At the top of the screen are the details of how the Integrator was started, including which command line parameters were used.

These specify both the configuration file and the AssemblyLine to run. Then there is a brief banner that displays the version number of the Integrator. This is followed by a couple of lines that show where the Java libraries and extensions are being loaded from.

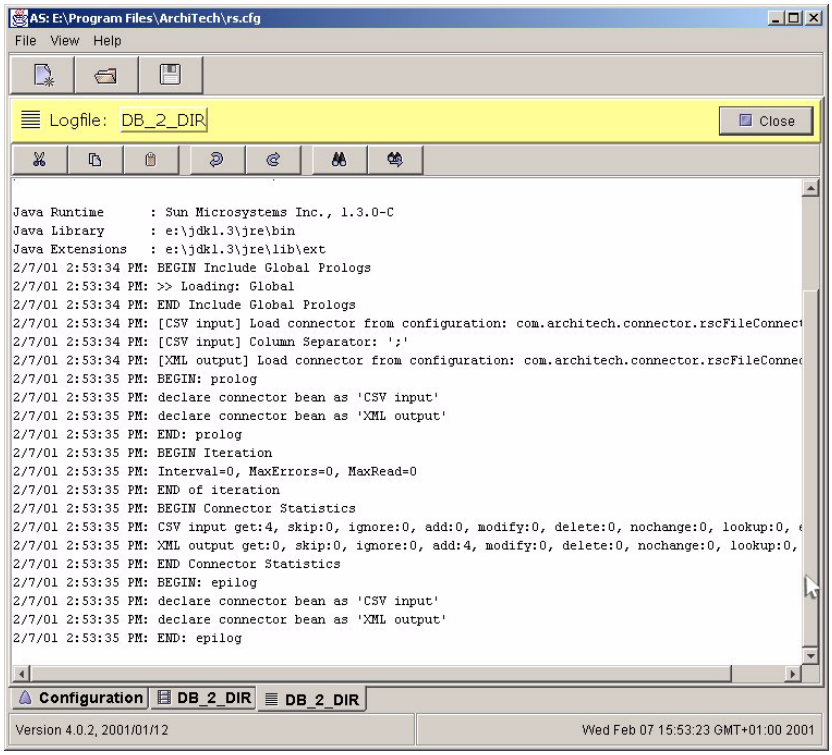
From the above monitor output we can also see that this is an evaluation copy of the Integrator, as well as the date when this copy expires.

In the second line from the bottom of the AssemblyLine status area we can see that our “DB_2_DIR” AssemblyLine ran with no errors.

If you now press the Logfile button, then the system will display the logfile generated by our AssemblyLine.



A new Context tab opens with the contents of the logfile:



The logfile gives us a view of what goes on behind the curtain when the Integrator is running an AssemblyLine. Let's walk quickly through it.

- First Global Prologs[6] are loaded and evaluated.
- Next the connectors are instantiated and initialized, making all connections to data sources. That way, if one of them fails, then we know about it before we start processing any data.
- After the Connectors are ready, the AssemblyLine's own Prolog script is executed.

Here is where fun starts:

- The Integrator passes control to the first Connector, which is in *Iterator* mode. We can see from the logfile that we have not entered any values for the parameters controlling the timeout interval, maximum number of errors before aborting, or the maximum number of reads for Connectors in *Iterator* mode in this AssemblyLine[7].
- After that we get some summary statistics showing that four entries were read by the CSV input Connector, and that four entries were written (added) by the XML output Connector. That jibes nicely with our expectations.
- Finally the Epilog script is executed and the AssemblyLine ends.

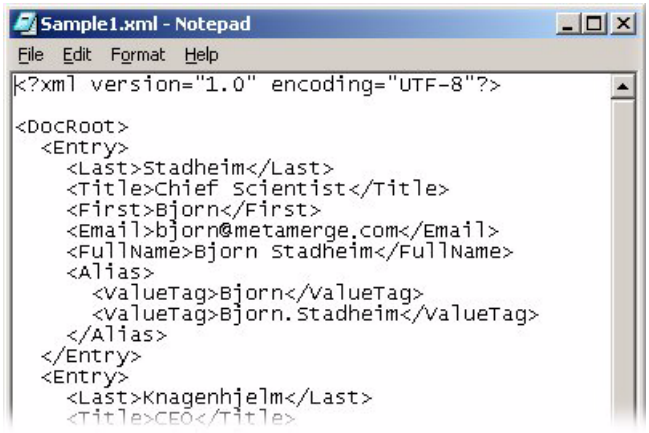
So far everything looks good. The only place we haven't checked yet is the output XML document itself.

Back on page page 32 we told the XML output Connector to call the file `Sample1.xml` and to write it to the `Sample1` sub-directory. If we open this file, preferrably in a browser that can interpret XML, like Internet Explorer or Netscape Navigator, we can

6. There can be any number of Global Prolog scripts, and these are defined under the “Global Prolog” screen accessed through the Object tabs. These are evaluated before AssemblyLines are started. That way any variables or functions are available to the scripts in the AssemblyLine.

7. The read timeout interval is set in the Connector Configuration dialog, while the other two parameters belong to the AssemblyLine and can be accessed from the “Settings” tab from the AssemblyLine screen. This is also where you specify Global Prologs to evaluate as well as which script language is to be used for coding hooks in AssemblyLine Connectors.

see that it does indeed contain the XML data that our AssemblyLine just created. Mission accomplished.



But wait! That last entry looks a bit strange. Obviously ;) we were supposed to skip that one.

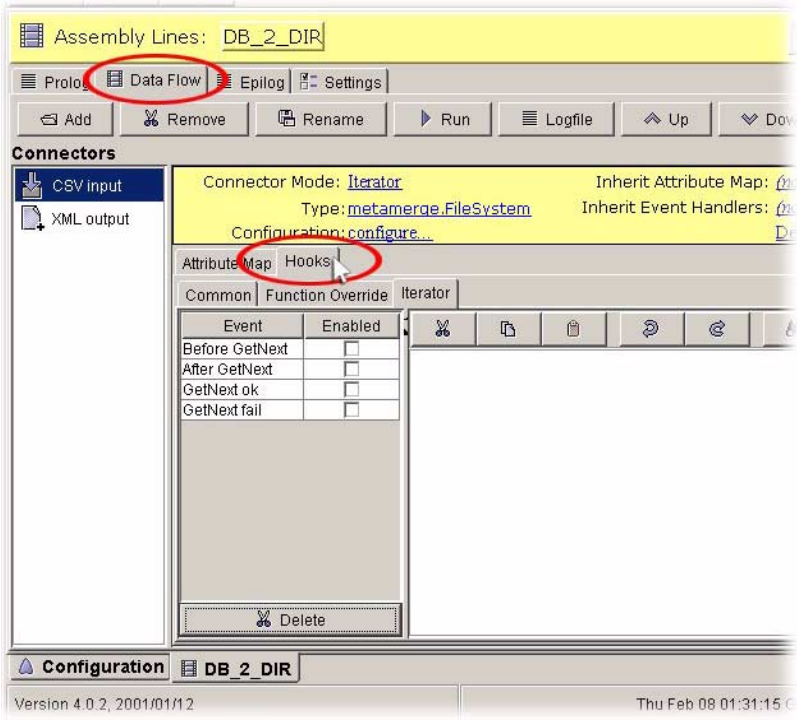
That means we'll have to add some intelligence to our input Connector to weed out unwanted data. We'll do this by *hooking* one or more scripts to the Connector.

Connector Hooks

Hooks are triggers in a Connector's work cycle where you can add a script which is to be executed each time the Integrator reaches that point; Like before a *Delete*, or after an *Update*. The hooked script will run before the Connector continues.

Each Connector mode offers its own set of Hooks in addition to those that are common to all modes. To access a Connector's Hooks, you have to select the desired Connector in the AssemblyLine “Data FLow” list, make sure you are in the “Data-

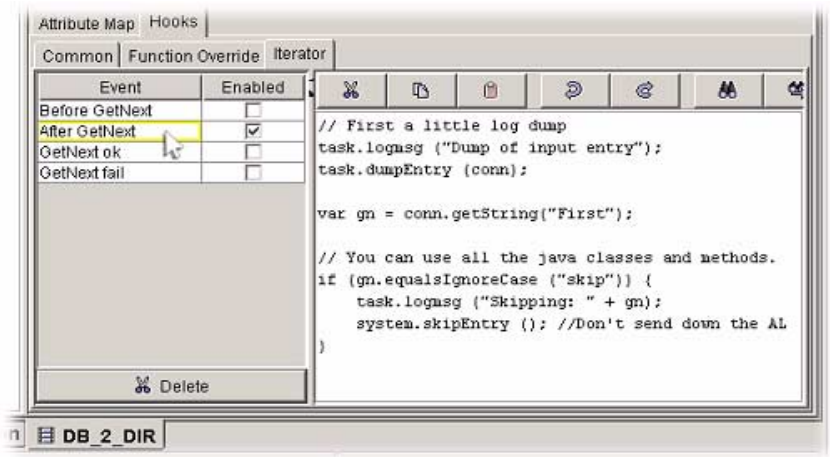
Flow" tab of the AssemblyLine and then select the Connector tab labeled "Hooks".



As you can see, the Connector Hooks that are available in *Iterator* mode are: **Before GetNext**, **After GetNext**, **GetNext ok** and **GetNext fail**.

In addition to the sub-tab which is labeled the same as the Connector's mode, there are two other sub-tabs under "Hooks": "Common", for those Hooks that all Connector modes offer, and "Function Override" for replacing the Connector's data access functions, like **GetNext**, **Lookup** *and* **Update**, *with your own scripted versions.*

We'll drop our script (again in JavaScript) for filtering the input data into the **After GetNext** Hook.



A more legible copy of the script—without the comment lines—follows:

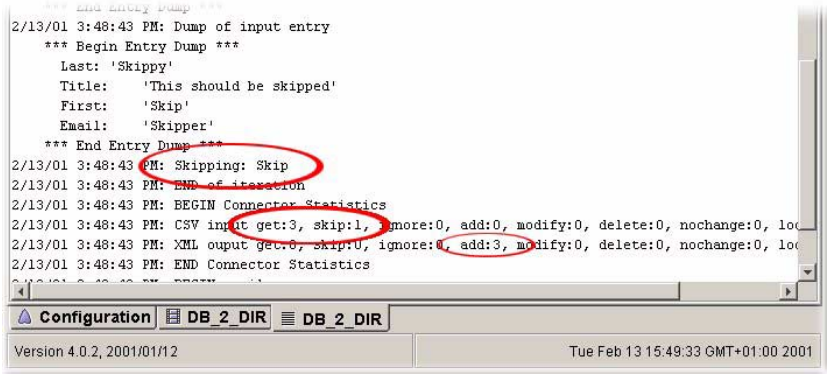
```
task.logmsg("Dump of input entry");
task.dumpEntry(conn);

var gn = conn.getString("First");

if (gn.equalsIgnoreCase("skip")) {
    task.logmsg("Skipping: " + gn);
    system.skipEntry();
}
```

The first thing we do here is to write the input data to the log file[8]. Then we check for the "skip" value, and if we find it, we ignore this entry, returning instead to the top of the *Iterator* loop to get the next one.

8. Although writing scripts is not within the scope of this manual, we'll take a quick look at some of the commonly used objects available for scripting in the Integrator:

system	This object provides access to system variables and functions, like **skipEntry**() in the example above.
task	The **task** object represents the AssemblyLine itself and offers functions like **logmsg**() and **dumpEntry**().
work	Used to access the **WorkEntity's** data and functions.
conn	The Connector itself, providing access to local storage attributes and operations.
entry	This object is a base type that other objects, like **work** and **conn** are inherited from. As a result, it may be used from within a Connector script to reference variables and functions in the Connector itself, instead of using the **conn** object.

When we run this AssemblyLine again, we now get the following output to our logfile, showing us that our script is working as planned.



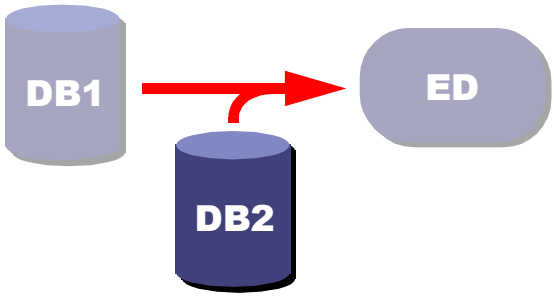
Each record is being dumped to the logfile as it is read, and our script is causing the input Connector to skip the specified entry. The other three entries were passed into the AssemblyLine and then all three were written to the XML document.

Checking our output file, we find that the last record has been omitted.

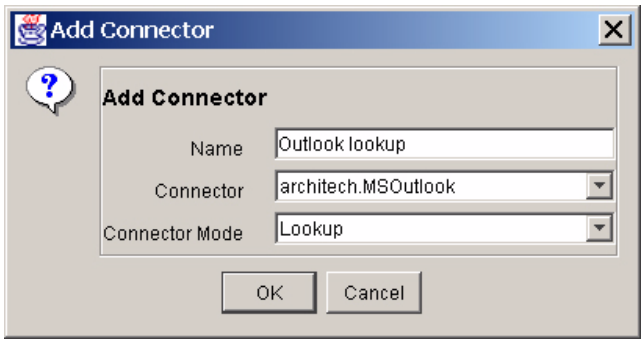
Aggregating data

And now an apology to our Unix users, because the next part of this example will involve Microsoft Outlook. We are going to use the **Email** attribute from the input Connector to get each person's mobile phone number from the Outlook Contacts database and write it to the XML document with the other data[9].

This will complete our implementation of the data flow diagram we drew back on page 5, with the Outlook Contacts database assuming the role of our “DB2” data source.



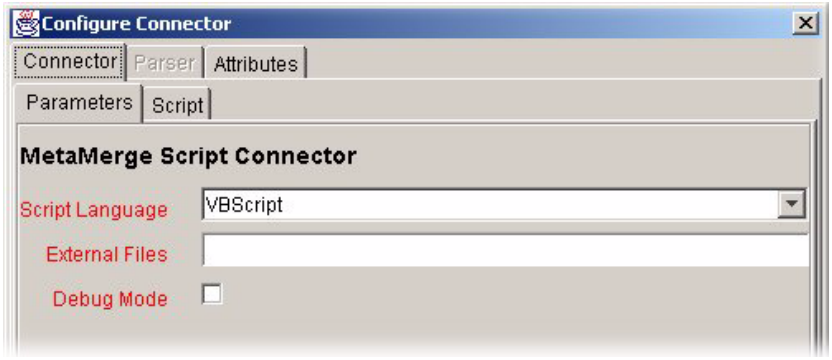
The first thing we have to do is to add a new Connector to our AssemblyLine. This time we will call it "OutlookLookup", and choose the MSOutlook type and *Lookup* mode.



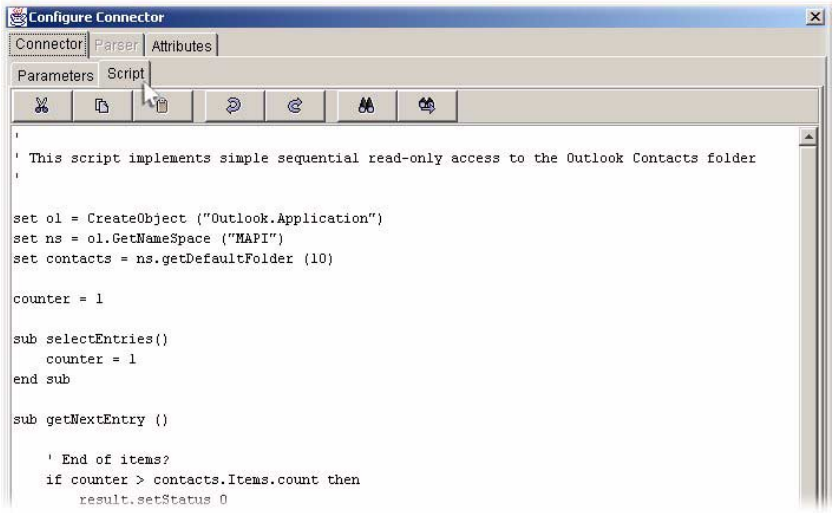
Now this may look like an ordinary Connector, but in fact, it is actually a *Script Connector*, in that all the functionality has been implemented through a script.

9. You are going to have to add the founders of Metamerge to your list of Contacts in Outlook, in order to get this demo to work. Pretty sneaky, eh? Remember to add a value for the **mobile telephone number**.

Bring up the Connector Configuration dialog in order to view the script.



As you can see, this Connector has been created using Visual Basic Script. If we click on the "Script" tab, we get the script itself.

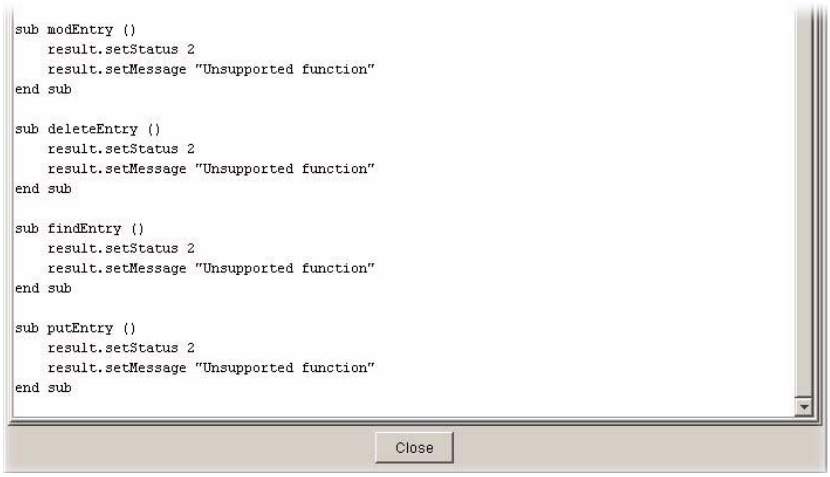


Again, without delving into details here, we can see that this script starts out by setting up a connection with Outlook. It then implements a number of functions that have to be in place for a Script Connector, although many of them will not be used for a Connector in Lookup mode.

For example, since the **getNextEntry** function has been implemented, then this Connector can be used in *Iterator* mode[10].

10. That means that instead of reading from the CSV input file, we could have spooled all the contacts in Outlook to our XML document.

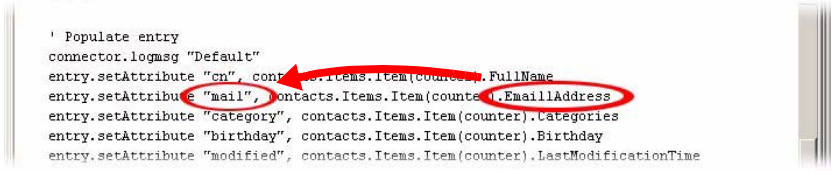
However, if we look further down in the script, we see that most of other functions are simply dummies that return an error message if used.



Since we need to be able to use this Connector in *Lookup* mode, we will have to make our own **findEntry** function.

Actually, we are going to make several changes.

First off, we might as well complete this Connector by implementing all the functionality needed to use this Connector in other modes as well. But most importantly, the script above maps the Outlook data into Connector attributes with different names than those used internally in Outlook[11].



In order to script functions that access features and variables in Outlook, we will need to maintain a translation table over attribute names in our Connector and what they're called internally in Outlook. Not a very pretty, or maintainable solution in the long run.

Instead, we will simply rewrite this script to use Outlook's own attribute names, as well as adding the functions needed to support all Connector modes. You can download the enhanced script from the Metamerge Customer pages using the link found on the *Docs and References* page.

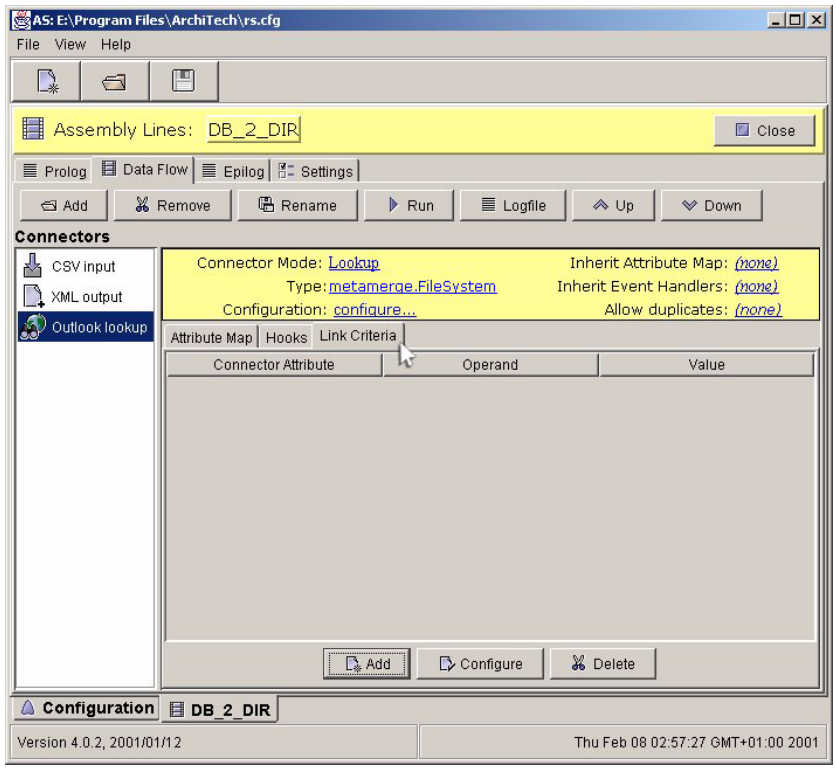
Just copy the contents of this file into the Connector.

11. Here you can see how the **entry** object is used to access local Connector attributes, as mentioned in footnote 8. on page 41.

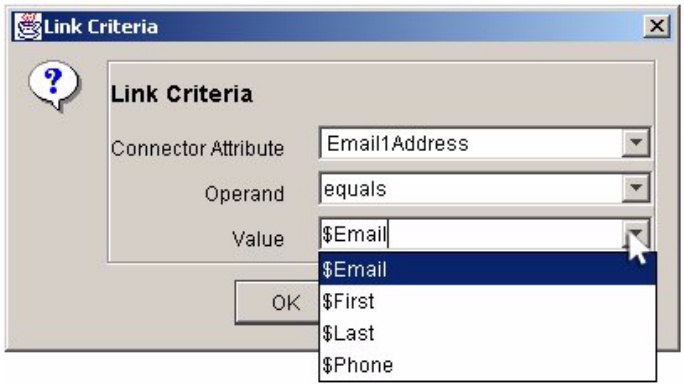
Once the script is in place, we can continue to configure this Connector like any other.

Since we are doing a Lookup, we will have to tell the Connector what to look for. This can be quite a problem when you're trying to aggregate information from two or more sources that have little or no data overlap; no common fields or attributes that you can search with to find a match. Sometimes you're even forced to build a cross-reference table.

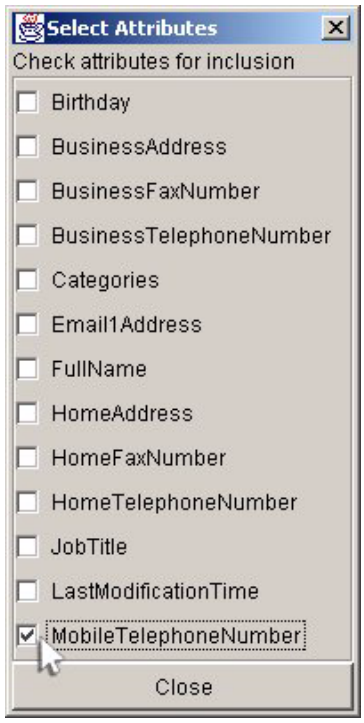
As luck would have it, there is an attribute that is stored in both data sources: the Email attribute. Click on the "Link Criteria" Connector tab and we'll add this Link Criteria.



Press the **Add** button and specify that a match is when the “Email1Address” field read from Outlook and the “Email” attribute of the WorkEntity are equal.



Open the configuration dialog for the "OutlookLookup" Connector. Bring up the “Attributes” tab and press the **Connect** button, and then **Next**. This makes the Connector read in both the first entry in the Outlook Contacts register, but also the structure of the data. Close this dialog and then click on **Select** at the bottom of the Connector's Working Attribute list. Choose to include the “MobileTelephoneNumber”[12] attribute.

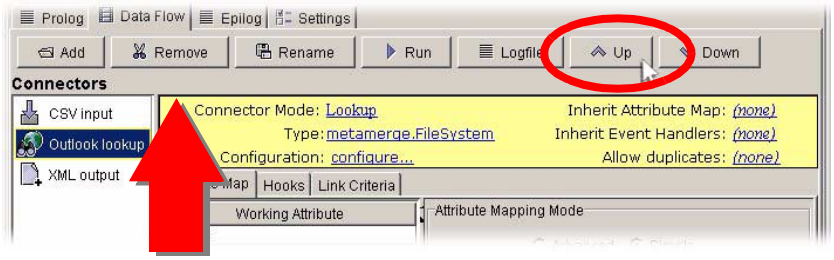


Now we have to map this attribute into the output Connector as well. However, before we can do that, we must move the

12. Just drag the window larger if you can't see the “MobileTelephoneNumber” attribute.

"OutlookLookup" Connector ahead of "XML output" in the AssemblyLine. Otherwise the new attribute will not be ‘visible’ to the XML Connector.

Select the "OutlookLookup" Connector and press the **Up** button.



Now you can select the "XML output" Connector and press the **Select** button at the bottom of the Working Attribute list.

You should now get a new entry in the selection dialog that you didn't get the last time.

Select “MobileTelephoneNumber” for inclusion and close the selection dialog.

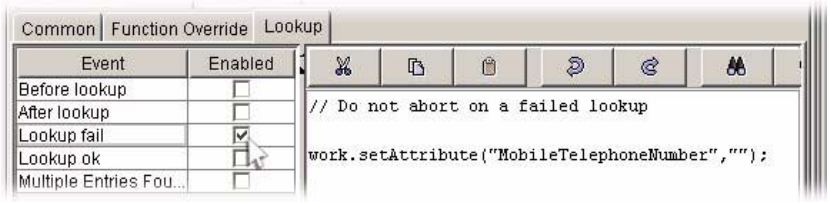
Now if you take a look at the following excerpt from the improved Outlook Connector script, you will see that if the

findEntry function does not find a match then it returns a status of 2, along with an error message: “Not found”.

```
sub findEntry ()
    flt = "[" &
        search.getFirstCriteriaName() &
        "] = '" &
        search.getFirstCriteriaValue() &
        "'"
    set item = contacts.Items.Find ( flt )
    if item is nothing then
        result.setStatus 2
        result.setMessage "Not found"
    else
        populateEntry entry, item
    end if
end sub
```

Since we have not configured this AssemblyLine to tolerate any errors, then it will halt because of this return value.

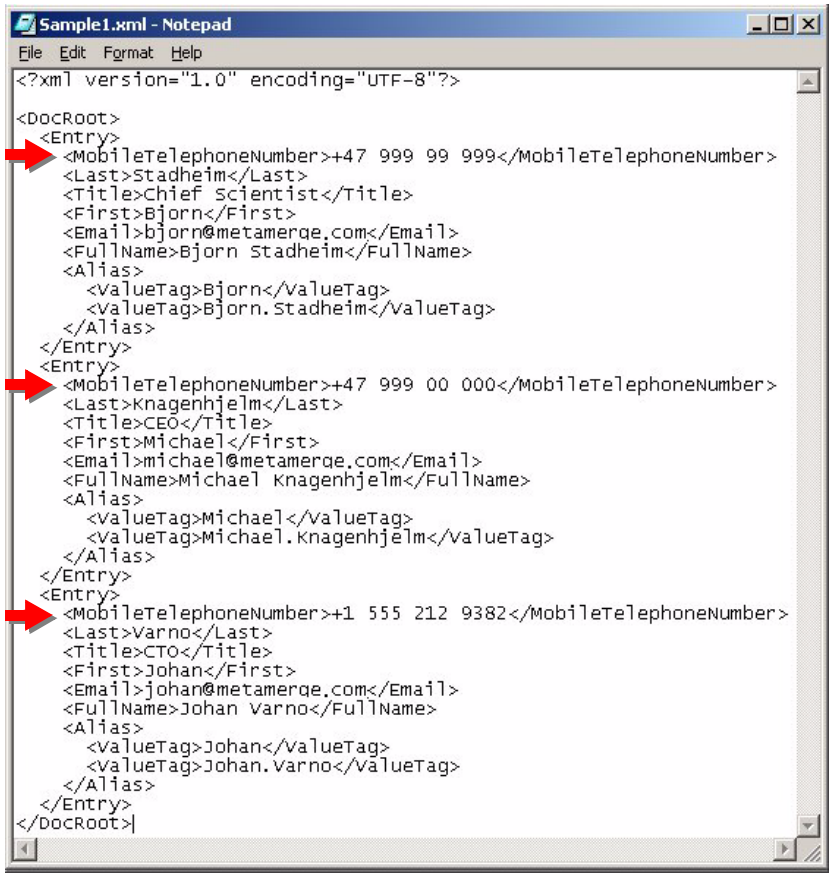
We have two options here: either rewrite the Connector script to not treat a failed search as an error, or we can trap that error by putting a script into the **Lookup fail** Hook for this Connector.



An empty script would have been enough to cause the error to be swallowed. We are going to add the following line to assign the attribute a default value:

```
work.setAttribute(”MobileTelephoneNumber,””);
```

Finally, our enhanced AssemblyLine is ready. *Save the configuration file* and run the AssemblyLine again, and then let's look for the new attribute in the output XML document.



There it is: the “MobileTelephoneNumber” attribute has been retrieved from the Contact register in Outlook and written to the XML file. It's a wrap[13].

And we can continue to enhance and improve our AssemblyLine, but then that's the beauty of the Integrator: *incremental implementation*. It means that you can grow your integration solution to fit your needs.

With the Integrator, you get an infrastructure that fits perfectly in its environment because it's been grown and evolved there.

And when you approach an integration problem by first atomizing the individual dataflows, you reduce complexity. This gives you gains across the board: in deployment speed, accuracy of the solution, robustness, maintainability... The list goes on.

13. Ok, so the mobile telephone numbers are not legit. But if you need to get hold of any of these gentlemen, the place to start is www.metamerge.com (unless you've already managed to get hold of a business card).

Of course, the examples in this text have been reasonably simple. But you will soon find that practically any integration solution can be implemented quickly and efficiently with the Integrator. In fact, as you start to think in terms of Metamerge's **atomize and implement** mantra, you will see your installation, and your integration needs, from a whole new perspective.

Perception is reality, and our perception is formed (and limited) by the toolset we have at our disposal. So you can continue to accept reality as you perceive it, whittling away at the vision of your integration infrastructure in order to make it fit the tools you are using.

Or you can switch tools.

But be warned—it may be hard to go back.