Getting Started

with

tm

integrator

www.metamerge.com

Versions 4.6.6 or greater

Rev. 1.0.2 2002.02.06

Metamerge
Akersgaten 43
N-0158 Oslo, Norway

For further enquiries, contact info@metamerge.com
Or visit the web site at www.metamerge.com

metamerge

Contents

metamerge

Preface

About this manual

This book is a simple introduction to a simple system.

Make no mistake; the word ‘simple’ is used here 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 them one thread at a time; Studying their direction and content, as well as the part of the patchwork to which they belong. But this isn't much help if the integration tools we use don't let us express our understanding of the problem in these terms.

That's where Metamerge Integrator comes in, helping you preserve simplicity by giving you the tools to:

- Implement your integration solution in terms of individual threads of communication (the data flows) between systems and devices;
- Create, test and deploy each flow rapidly, and with immediate feedback;
- Easily maintain and evolve your solution as your organization and the world around it changes.

This means that integration projects become easier to estimate and plan, often simply a matter of counting and costing the individual data flows to be implemented. And since you are developing the solution flow-by-flow visually and interactively, you can report and demonstrate status to both project and corporate management at any time.

Furthermore, Integrator manages the technicalities of connecting to and interacting with the various data sources that you want to integrate, abstracting away the details of their APIs, transports, protocols and formats. Instead of focusing on data, Integrator lifts your view to the information level, allowing you to concentrate on the business and information management logic needed to perform each exchange.

Integrator also lets you build and maintain libraries of integration logic and components that can be reused to address new challenges. Development projects across your organization can all share Integrator assets, resulting in independent projects (even point solutions) that immediately fit into a coherent, integrated integration infrastructure.

Finally, the Metamerge approach results in a more rational and predictable use of resources, since you only bring in your data source and technology experts (Oracle, DB2, SecureWay, iPlanet, MQSeries...) at the very start of a project in order to set up your library of communication components. Once in place, these integration assets, although centrally managed, are available across the network, letting you leverage them to create new solutions and enhancing existing ones.

Although many of these topics are dealt with in the Customer Pages of the Metamerge website (www.metamerge.com), this document will give you an introduction to our approach, as well as how to tap into the radical and elegant simplicity of Metamerge Integrator.

Scripting Languages

Integrator allows you to choose from a wide selection of scripting languages, including JavaScript, VBScript and Perl. In fact, any script language that plugs into the Windows Scripting Host under Windows or the Beans Scripting Framework under Unix may be used.

If you want to use a particular scripting language which is not already available to you in Integrator, then please contact your Metamerge representative for more information on how to secure and install the desired interpreter, or check our online documentation.

In addition, please note that this manual (as well as other Metamerge documentation) does not expressly cover the use or applicability any of these languages. Although JavaScript is the language most used with Integrator — and many of the example scripts in this manual do show the usage of JavaScript for adding transformation and computational logic to an Integrator solution — you should refer to relevant JavaScript documentation for further details on this language.

Installing Metamerge Integrator

Before you can install Metamerge Integrator, you must first make sure that you have Java 2, version 1.3 or newer, installed on the machine. Note that if you are planning to run Integrator under Windows (NT/2000/XP), then the you can choose between two installation programs: one that includes the Java VM, and one without.

Otherwise, 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 to download and install is the run-time.

Once you have Java 2 in place, you can install Integrator using the automatic installation program. If for some reason the installation program does not operate on your system, please check our website for manual installation instructions, or contact your Metamerge representative.

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

Sometimes you will find Service Releases for the latest version. In this case it is only necessary to download and install the latest service release.

Finally, please make sure that you have the latest version of Integrator installed. Otherwise some of the screenshots in this and other documentation may differ from those presented by your system. The cover page of the manual indicates which versions it applies to.

Installing the Tutorial Files

In order to work with the examples in this manual, you should refer back to our website for both a link to download the necessary files, as well as instructions on where to install them.

You will still be able to complete the first part of the Getting Started tutorial without them. However, in order to add *Join* functionality to your AssemblyLine, it will be an advantage to have these example files in place.

metamerge

Simplify and Solve

How Do You Eat an Elephant?

The answer is: one bite at a time. The same approach applies equally well to an integration or systems deployment project.

The key to success is to reduce complexity by breaking the problem up into smaller, manageable pieces. This means starting with a portion of the overall solution, preferably one that can be completed in a week or two. Ideally, this should also be a piece that can be put independently into production. That way, it's already providing return on investment while you tackle the rest.

Once you have isolated the piece you are going to work with, simplify it further by isolating the basic units of communication, the data flows themselves, and you will be poised to start implementing them using Metamerge Integrator.

Integration development is done with Integrator through a series of try-test-refine cycles, making the process an exploratory one. This will not only help you to discover more about your own installation, but also let you evolve your integration solution as your understanding of the problem set grows.

A great way to get a good mental picture of the problem at hand is to make a picture of it. Grab a pencil and a piece of paper (and an eraser) and sketch out a flow diagram that maps out the solution in broad strokes. This exercise will not only help you to visual the scope of the task, it will serve as the blueprint for implementing it in Metamerge Integrator.

To Integrate Is To Communicate

Integration problems are all about communication, and as such can typically be broken down into four parts: The systems and devices that are to communicate, when they're sup-

posed to talk, what they should say to each other and how they should say it. These four constituent elements of a communications scenario can be described as follows:

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.

Data sources represent a wide variety of systems and repositories, like databases (Oracle, DB2, SQL Server), directories (iPlanet, SecureWay, ActiveDirectory), directory services (e.g. Exchange), files (XML, LDIF or SOAP documents), specially formatted email, or any number of interfacing mechanisms that internal systems and external business partners use to communicate with your information assets and services.

Events

Events can be described as the circumstances that dictate when one set of data sources is to talk to another. One example is whenever an employee is added to, updated or deleted from the HR system. Or when the access control system detects a keycard being used in a restricted area. An event can also be based on a calendar/clock based timer, e.g., starting communications at 12:00 midnight, every day except on Sundays. It could even be a one-off event like populating a directory.

Events are usually tied to a data source, and are related to the data flows that are triggered when the specified set of circumstances arise.

Data Flows

These are the threads of the conversations and their content, and are usually drawn as arrows which point in the direction of data movement.

Each data flow represents a unique message being passed from one set of data sources to another.

metamerge

Table 1: Integration elements (Continued)

Attribute Mapping and Transformation

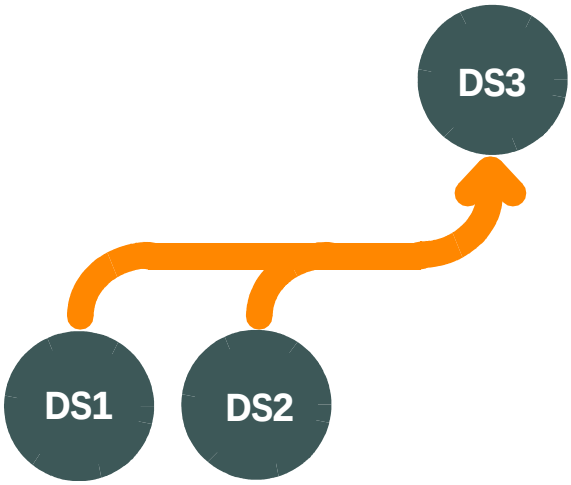
It goes without saying that for a conversation to be meaningful to all participants, everyone involved must understand what is being said. But we can probably count on our data sources representing their data content in a different ways. One system might represent a telephone number as textual information, including the dashes and parenthesis used to make the number easier to read. Another one 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 in one source may not be complete, and may need to be augmented with attributes from other data sources. Furthermore, only parts of the data in the flow may be relevant to some of our output sources.

Choosing which fields or attributes are to be handled in a data flow, or passed on to a data source, as well as how each connected system refers to and represents this information, is called Attribute Mapping.

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 (or when someone else does), it still makes sense.

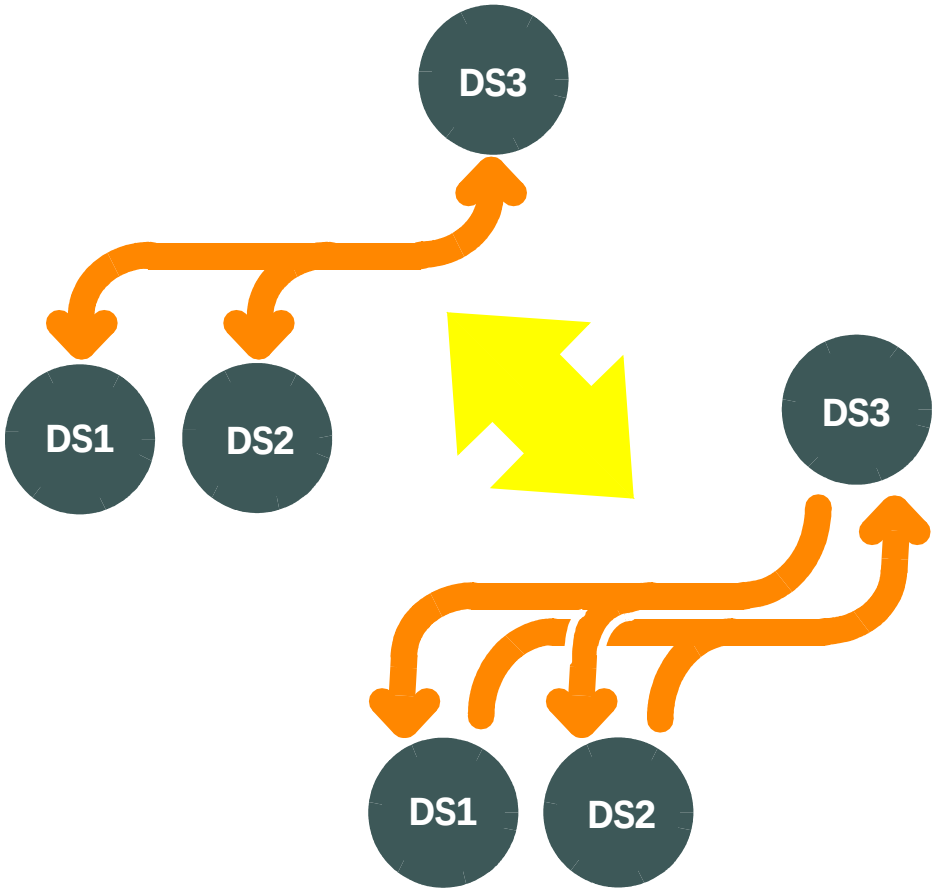
Let's start our first Integrator tutorial by doing a simple a flow diagram:



Here we see the third data source (DS3) getting data from DS1. Along the way, the dataflow also aggregates information from a second source (DS2).

First off, it's important to understand that each AssemblyLine implements a single uni-directional data flow. If we wish to do bi-directional synchronization between two or more

data sources, then we will have to use a separate AssemblyLine for handling the flow in each direction. The reason for this is that the form and content of the data, as well as the operations carried out on it, will most likely be different for each direction[1].

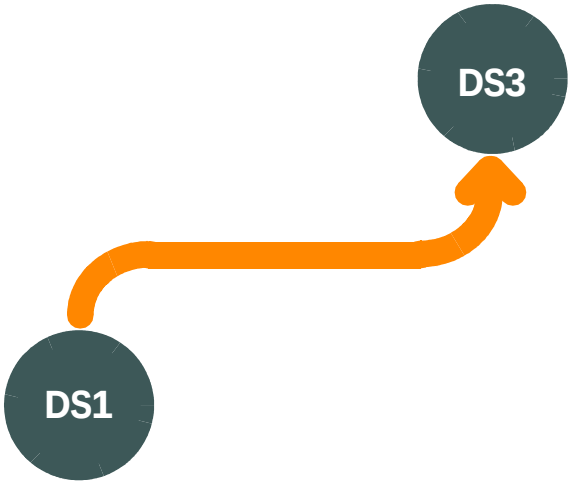


Although there are no limits to the number of Connectors that an AssemblyLines can contain, our AssemblyLines should be made up of as few Connectors as possible (e.g., one per data source participating in the flow), while at the same time including enough components and script logic to make the AssemblyLine as autonomous as possible. The reasoning behind this is primarily to make the AssemblyLine easy to understand and maintain. It will also result in simpler, faster and more scalable solutions.

The Metamerge philosophy is all about dealing with the flows one at a time, simplifying the problem set, so we'll zoom in on the flow going from DS1 to DS3.

1. Integrator is fully featured for creating request-response information solutions like Web Services. For more information, check out our website, under the *Docs & Resources* section of the Customer Pages.

This arrow could describe the operation of populating the directory, or migrating data from one type (or version) of data store to another.

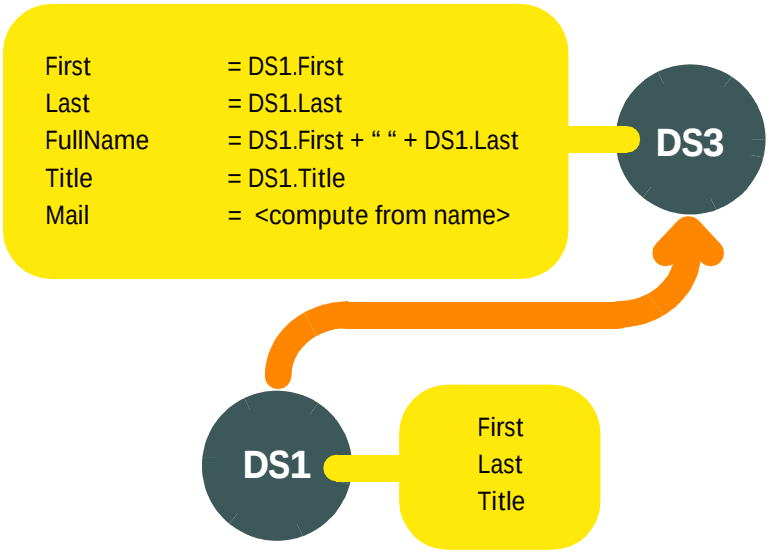


Our next step is to detail the data that will be sent.

Some data sources, like databases, 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 addition, these two types of sources use different sets of predefined data types.

But we don't really have to worry about how the data is stored in the sources: Integrator takes care of that for us. Everything that gets pulled into our data flow will be converted to a canonical format: Java objects, so that once inside the AssemblyLine, all data elements are called "*attributes*", share the same set of data types and can be dealt with in a generic fashion.

In order to complete our visualization of the data flow, we'll record how the attributes of the input data source are mapped (and possibly modified) to become those of the output source.



To keep our example simple and easy to install, we'll be using a comma-separated value file[2] as our DS1 that contains the fields **First**, **Last** and **Title**. Our output data source (DS3) will be an XML document.

Now that we have a good representation of our solution, let's take a look at how Metamerge Integrator handles data flows.

2. This input data file is available for download from the Customer pages of the Metamerge website.

AssemblyLines

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

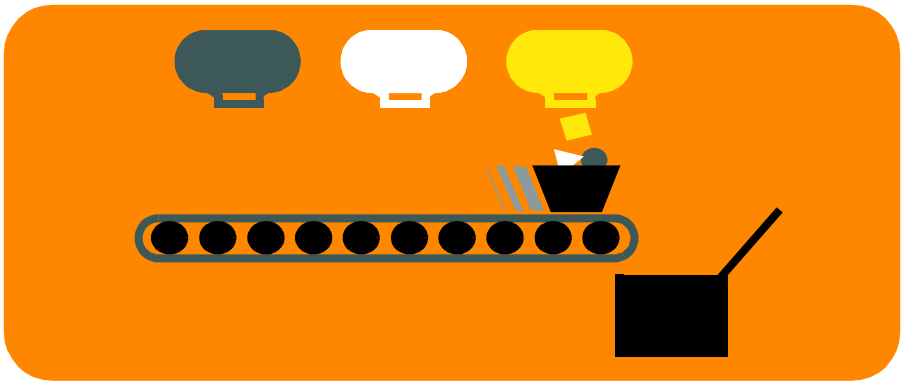
Real-world assembly lines are made up of a number of specialized machines that differ in both function and construction, but have one significant attribute in common: they can be linked together to form a continuous path from input source(s) to output.

An assembly line generally has one or more input units designed to accept fish fillets, cola syrup, car parts — whatever the raw materials needed for production are. These ingredients are processed and merged together. Sometimes by-products are extracted from the line along the way. At the end of the production line, the finished goods are delivered to waiting output units.

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.

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.

Integrator AssemblyLines receive information from various input units, perform operations on this input and then convey the finished "product" through output units. Furthermore, Integrator AssemblyLines work on only a single item at a time — i.e. one data record, directory entry, registry key, etc. Data from the connected input sources (e.g., fields, attributes,



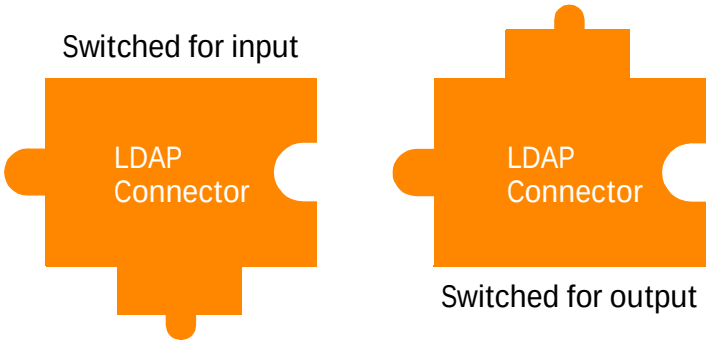
values...) are accumulated in a Java "bucket" (called the ***work*** object) and scripts are applied to this information, verifying data content, computing new values and changing existing ones, until it is ready for delivery from the line into one or more output sources.

The input and output units of an Integrator AssemblyLine are called *Connectors*, and each Connector is linked into a data source. Connectors tie the dataflow to the outside world, and are also the place where most data transformation and aggregation take place. They are also where you can layer in your business, security and identity management logic.

Connectors

Connectors are like puzzle pieces that click together while at the same time linking to a specific data source, like an SQL database, message queue or a text file.

Each time you select one of these puzzle pieces and add it to an AssemblyLine, you first chose the type of Connector. Then you assign the Connector its role in the data flow. This is called the Connector *mode*, and it tells Integrator how to use the Connector: either as in input unit, iterating through, or looking up information in its source — or as an output Connector, inserting, updating or deleting data in the connected system or device[3].

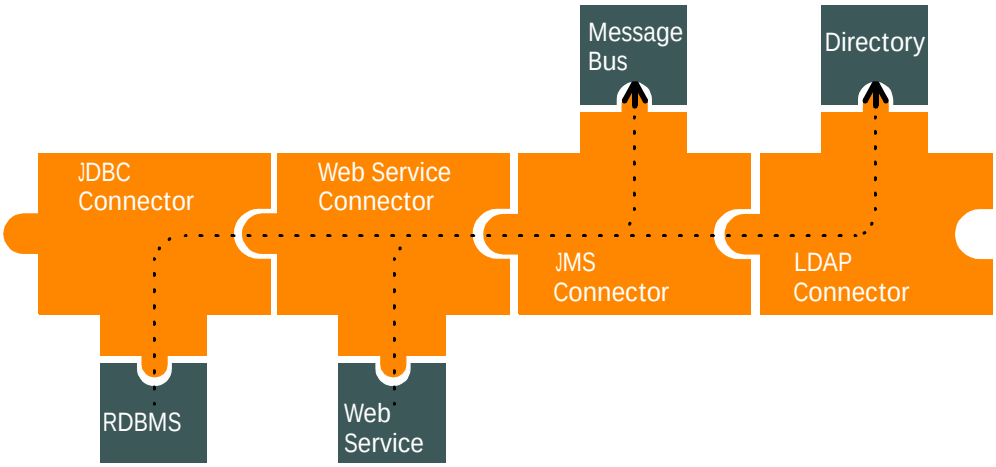


You can change both the type and mode of a Connector whenever you want, and if you've planned for this eventuality, then the rest of the AssemblyLine may not need to be changed at all. That's why it's important to treat each Connector as a "black box" that either delivers data into the mix, or extracts some of it to send to a data source. The more independent each Connector is, the easier your solution will be to augment and maintain. Not to mention making it possible to swap out data sources entirely!

By making your Connectors as autonomous as possible, you can also readily transfer them to your Connector Library and reuse them to create new solutions faster — even share them with others. Using Integrator's library feature also makes maintaining and enhancing your Connectors easier, since all you have to do is update the Connector *template* in your library, and any number of AssemblyLines using this template will inherit the enhancements. And when you are ready to put your solution to serious work, you can reconfigure your library Connectors to connect to the production data sources instead of those in your test environment, and move your solution from lab to live in minutes.

3. 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.

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



Metamerge Integrator gives you a library of Connectors to choose from, like LDAP, JDBC, MS NT4 Domain, Lotus Notes and JMS. And if you can't find the one you are looking for, you can extend an existing Connector by overriding any or all of its functions using one of the leading scripting languages, including JavaScript, Visual Basic and Perl. You can even roll your own, either with a script language inside our Script Connector "wrapper", or from scratch using Java or C/C++. But be sure to check the Customer Pages of our website to make sure that someone else hasn't already done this for you! Connectors, like all other Integrator components, are free to download and trade.

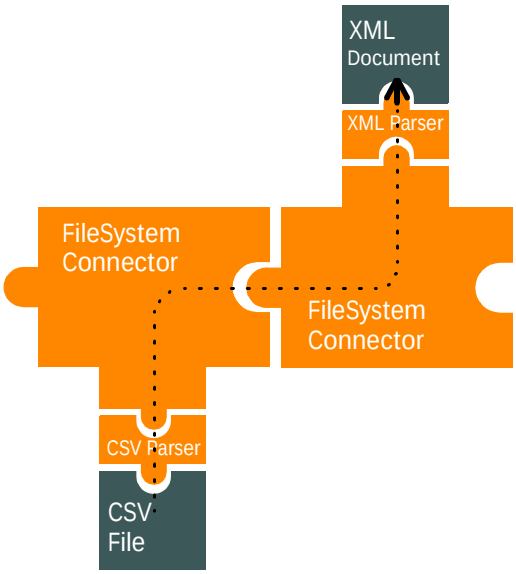
Furthermore, Integrator supports most transport protocols and mechanisms, like TCP/IP, HTTP and FTP — with or without SSL, or other encryption mechanisms to secure the information flow.

Parsers

Even unstructured data, like text files or bytestreams coming over an IP port, are handled quickly and simply with Integrator by passing the bytestream through one or more *Parsers*. Parsers are a second type of Integrator component, and the system is shipped with a variety of Parsers, including LDIF, DSML, XML, CSV and Fixed-length field. And just like Connectors, you can extend and modify these, as well as create your own.

Continuing with our example from page 10, the next step is to identify our data sources. Since our input data source is a text file in comma-separated value format, then we will be using the FileSystem Connector paired up with the CSV parser. We'll use a FileSystem Connector for output as well, but this time we will choose the XML parser in order to format our file as an XML document. The actual mechanics of doing this in Integrator will be dis-

cussed in a minute. First, we'll take a look at what our AssemblyLine looks like visually, using our puzzle pieces.



Now that we've identified which components to use, we are ready to build this AssemblyLine using Metamerge Integrator. However, before you continue, you will need an input file. You can either download this file from the Customer pages of the Metamerge website, or create it yourself in a text editor. The data looks like this:

```
First;Last;Title
Bill;Sanderman;Chief Scientist
Mick;Kamerun;CEO
Jill;Vox;CTO
Roger
Gregory;Highpeak;VP Product Development
Peter;Belamy;Business Support Manager
```

Call this file "People.csv" and place it in the examples/Tutorial[4] sub-directory of your Integrator installation. Once it's is in place, you are ready to build your solution using Metamerge Integrator.

4. Please note that the examples in this manual have been created on a Windows platform, and therefore use the Windows pathname conventions. In order for your solution to be platform independent, you should be using the forward slash (/) instead of the backslash character (\) in your pathnames, e.g., examples/Tutorial/Tutorial1.cfg)

metamerge

Introducing Integrator

Rapid Integration Development

Metamerge Integrator is actually two programs:

- **Admin Tool** This program gives you a graphical interface to create, test and debug your integration solutions. It's what we call the TIDE — Transformation and Integration Development Environment — and with it you create a configuration that will be executed by the run-time engine (next point). This executable is called MIADMIN.
- **Run-time Server** Using the configuration file you've created with the Admin Tool, the Run-time Server powers the integration solution. This program file is called MISERVER, and you can deploy your solution using as many or as few server instances as you want. There are no limitations imposed by the technology or the Metamerge Integrator license agreement.

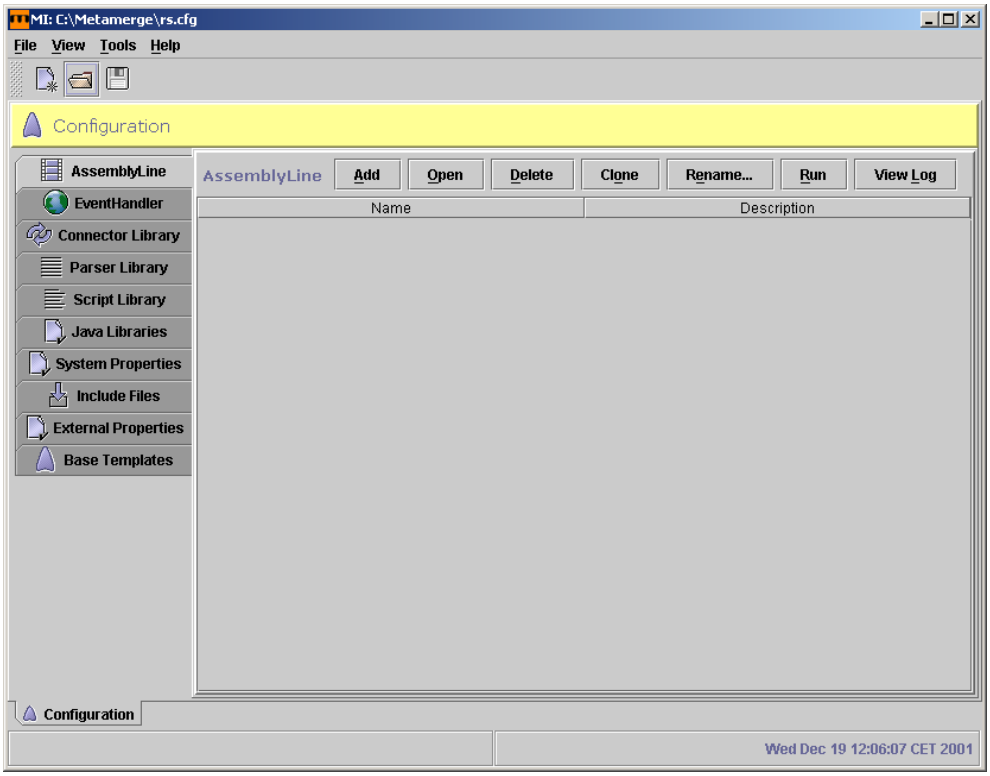
Both systems are written 100% in Java, and run in any environment that offers a Java 2 compliant Virtual Machine.

If you haven't already started the Admin Tool, do so now. After a moment you should be presented with the Main Screen. You may be asked if you wish to create a new configuration file called rs.cfg. Answer **Yes** and you will be presented with the main screen[1].

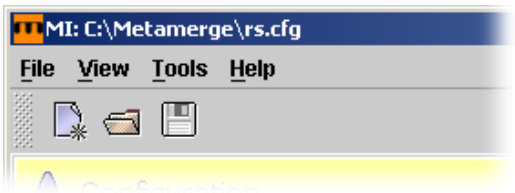
1. If the screen you see is different from the image above, your system may have a different display setting selected. You can change this with the menu selection **View | Look & Feel**. Please note also that the Integrator window is resizeable, so again, if the screenshots in this manual do not match your own experience, try changing the size of the window on your screen.

   The **View | Preferences** selection opens a dialog where you can set a number of other user interface parameters, like whether the Button toolbar is visible or not, or if you want Integrator to use the Status Bar at the bottom of the window.

   If this does not help, check the version number of Integrator, found by selecting **Help | About** from the Main Menu, and then reading the resulting dialog box. This version number should match up with the product version information on the cover of this manual.



At the top of the screen is the Main Menu and the Button Bar.



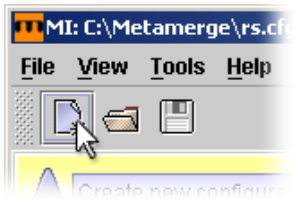
The Button Bar provides commands for creating new configuration files, open existing ones and saving your current work. These same commands are also available under the **File** menu, where you will also find the **Save As** selection for saving your configuration to a new filename. Note that the path and filename of the current configuration file appear at all times in the title bar of the Integrator window.

Creating A New Configuration File

Integrator configurations are stored in files that are created and maintained in the Admin Tool and run by the Server. Each configuration file contains the AssemblyLines that a Server is to run, as well as the Integrator components that make up these lines. It also holds user preferences like colors and the GUI interface style.

Although it's not within the scope of this text, configurations can also be spread over several files, and stored at several locations. Integrator will then assemble its configuration dynamically at startup, using include URLs/filepaths that you have specified. This means that you can create and maintain corporate settings and components that can be shared by many server configurations.

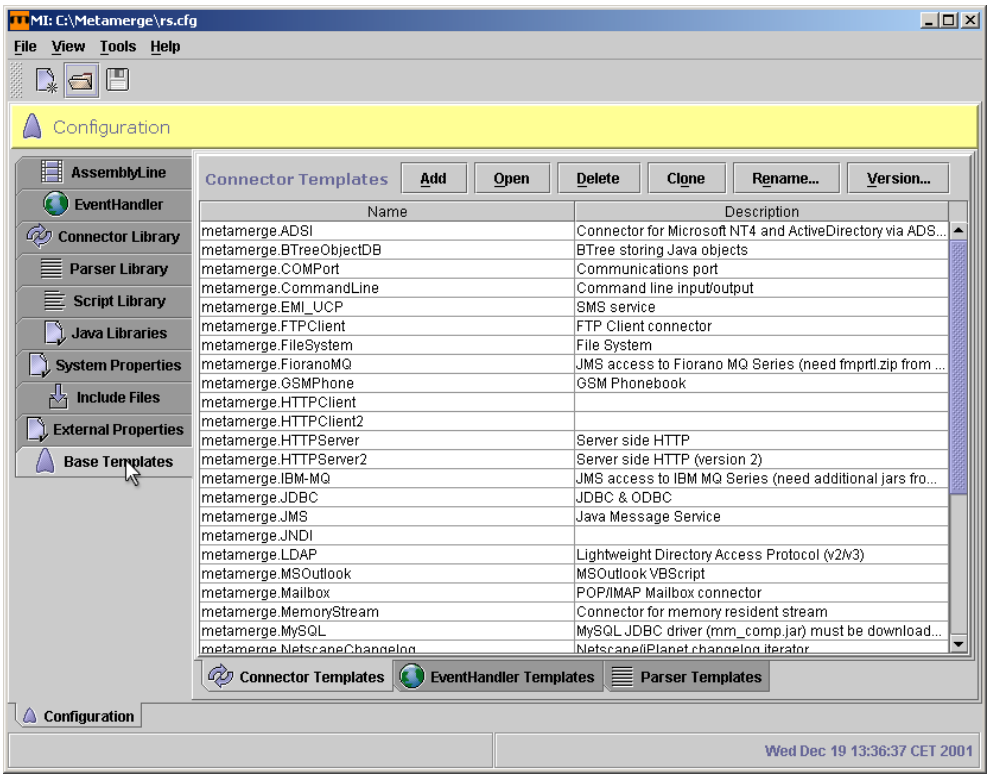
When you start the system for the first time, Integrator will ask if you want to create the default configuration file, called `rs.cfg`. Instead of using the default configuration file, we're going to create a new one. Do so by either pressing the **New Configuration** button, or by using the **File | New** menu selection to create a configuration file called "`Tutorial1`" (please note that the ".cfg" extension will be added for you). This file should be located in the `examples/Tutorial` directory[2]



Looking along the left side of the window, you can see a number of tabs. These are the **Configuration Tabs**, and they give you access to your AssemblyLines and EventHandlers (which we'll get into later). There are also tabs here for building and maintaining libraries of Integrator components, as well as for setting system and configuration properties.

2. Note that pathnames can be written as relative to the directory that you specified when installing Integrator.

At the very bottom is the tab where you will find the set of components that are installed with your system[3].

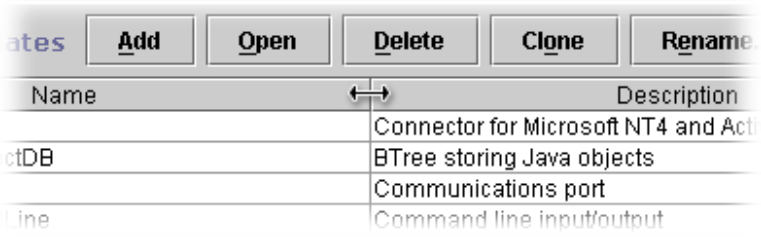


Without going into the Admin Tool interface in great detail, we'll go over the general layout of these configuration screens:

First, each one shows you an Element List of selected configuration objects. You select an item by clicking on it, and in most cases you can open or edit it by double-clicking. You can also change the space allocated for any column in the list by moving the mouse cursor over

3. The three lists under the Base Templates tab show you which components are currently installed. If you have just installed Integrator, then these are the components that were bundled with this release. Additional Connectors, EventHandlers and Parsers are also available for download from the Metamerge website as part of our *Early Technology Access* program. Check our website for more details.

the boundary between columns, and then clicking and dragging the boundary to its new location.



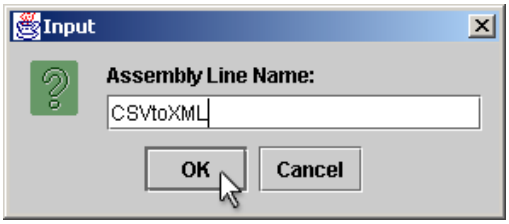
At the top of each Element List is a row of buttons providing the set of operations that are available for that type of object. The list of available operations will vary from screen to screen, but the general behavior is the same: You select an entry in the list and then press the button — with the obvious exception of the **Add** button, where you do not need to select anything first.

Finally, you can select several elements at once by clicking on one and dragging up or down in the list while holding the mouse button down. This is particularly handy if you want to delete more than one element.

That will do for now. Let's get started with creating our first integration solution.

Creating an AssemblyLine

The first thing we have to do is to create a new AssemblyLine. We do this by first selecting the **AssemblyLine** Object Tab and then pressing the **Add** button. C this AssemblyLine "CSVtoXML".



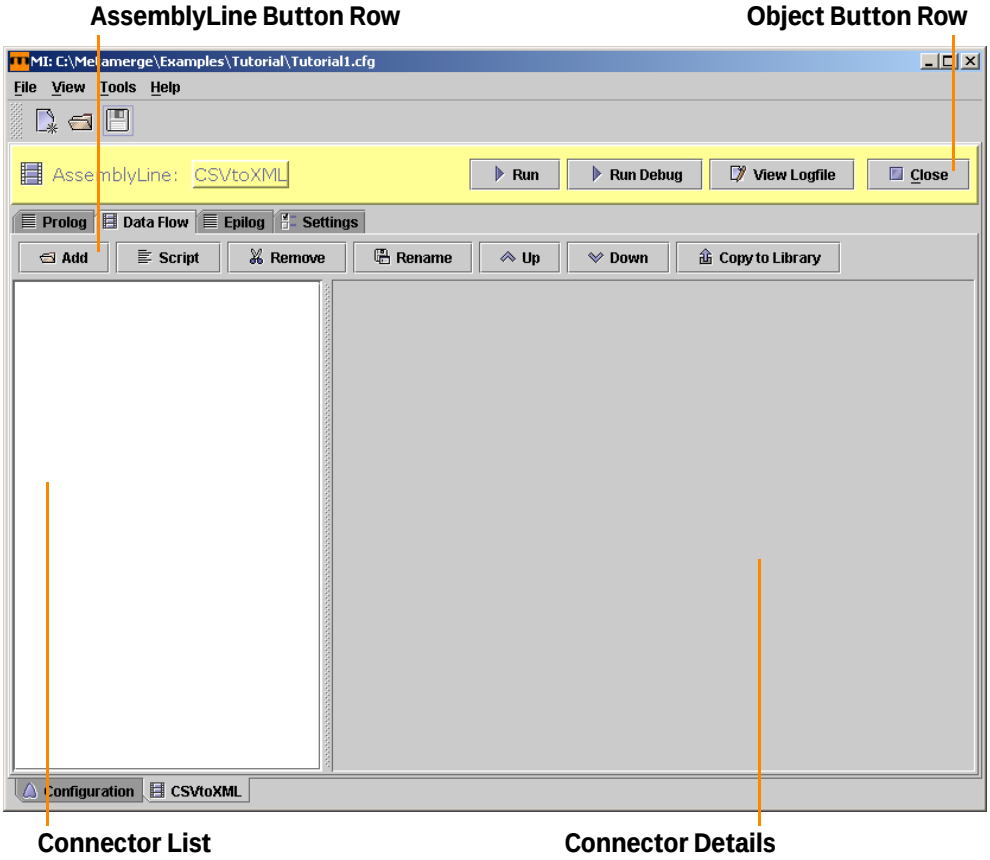
You can call an AssemblyLine whatever you want, but it is important to use a naming convention that will help to document your solution.

Use of special characters and spaces in naming AssemblyLines or Integrator components (like Connectors and EventHandlers) is not good idea, as it may cause problems later when you want to start Integrator Server from a command prompt to run your solution.

Integrator will now take you to the AssemblyLine screen. The first thing you will notice is that this new screen fills the entire window. But don't panic, you can go back to the Config-

uration screen (or any other open object) by selecting one of the *Context Tabs* at the bottom of the screen.

Before we start adding our Connectors, let's take a quick look at the layout of the AssemblyLine screen.



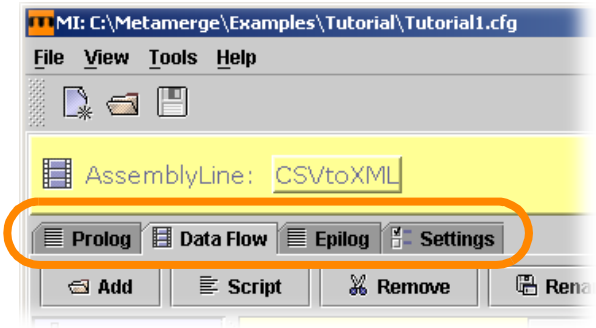
At the top of the details screens associated with each of the Object Tabs (in this case, our new AssemblyLine Details display), you'll find a colored title bar. In addition to the name of the currently selected object, this bar holds a button row that, at the very least, has a **Close** button for leaving this screen.

In the AssemblyLine Details screen we have the following buttons as well:

- **Run** This button lets us execute the current AssemblyLine;
- **Run Debug** Integrator includes a data flow debugger which allows us to step through our AssemblyLine, watching the data being transported and transformed inside the line, as well as our own script variables, as control is passed from Connector to Connector;
- **View Logfile** Each AssemblyLine gets its own set of logfiles, and we use this button to open a display window for the most recent copy.
- **Close** Closes this Object Details display.

The white box on the left side of the screen is the Connector List where new Connectors appear as we add them. To the right of the Connector List is a large gray area where the details of the currently selected Connector are displayed.

At the top of this details area are the AssemblyLine Tabs. These give you access to various aspects of this data flow:



- **Prolog** The Prolog tab let's you set up scripts to be evaluated/executed when the AssemblyLine starts up (and before control is passed to the first Connector).
- **Data Flow** Here is where the Connectors are created and maintained.
- **Epilog** The Epilog tab is for scripts that are to be run when the AssemblyLine is complete and is about to exit.
- **Settings** which gives you a number of configuration parameters for this AssemblyLine.

If you click on the **Settings** tab, you will see that you can do things here like selecting the script language you want to use in this AssemblyLine[4], as well as limiting the number of

4. Regardless of which script language you select to use in your AssemblyLine, Integrator still lets you use components that have been scripted using other languages. So you can write your AssemblyLine logic in JavaScript, but still use that Connector you created with VBScript, as well as your Perl-based Parser.

iterations (which is handy when developing and testing your AssemblyLine on large data sets).

Just below the AssemblyLine Tabs is the AssemblyLine Button Row, offering you a number of AssemblyLine operations:

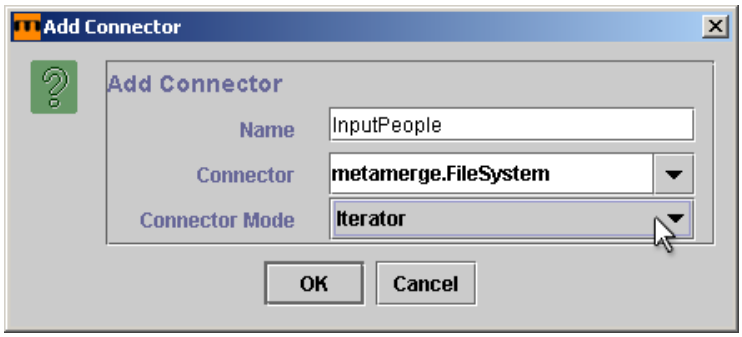
Going from left to right in this row, these buttons perform the following actions:

- **Add** Adds a new Connector to the AssemblyLine;
- **Script** Creates a "data-source free" Connector, called a *Script Component*, that can be used to hold script logic for manipulating data in the AssemblyLine;
- **Remove** Deletes the currently selected Connector;
- **Rename** Allows you to change the visual name of the current Connector;
- **Up** Move the selected Connector up one spot towards the start of the AssemblyLine. This position is significant since the AssemblyLine calls the Connectors in order from top to bottom;
- **Down** Takes the current Connector and pushes it down in the list of Connectors;
- **Copy to Library** Makes a copy of the selected Connector and drops it into your Connector Library.

We're going to make use of this button row in order to add our first Connector to the AssemblyLine.

Adding The Input Connector

Click on the **Add** button to create the first Connector.



We'll call it "InputPeople" and then choose the metamerge.FileSystem *template* from the list of available Connectors. This is the same list that you saw when you selected the Base Templates tab in the main Configuration screen back on page 18, plus any Connectors that you have in your Connector Library which show up at the top of the drop-down.

The last parameter to set here is the Connector *mode*, and this tells the AssemblyLine how this Connector is to be used. Integrator has six Connector modes[5]:

Table 1: Connector Modes

AddOnly

This mode is for Connectors that will only be adding new information to the data source, e.g., writing to files, populating a database or directory for the first time, etc.

Delete

Delete mode causes the Connector to search for a specific record (or records) to delete.

Iterator

A Connector in Iterator mode will run through the data source (or the desired part of it, like a view of a database, or the sub-tree of a directory) and then return these data objects (records, entries, etc.) one at a time for processing in the AssemblyLine. Connectors in Iterator mode are called *Iterators* for short.

An AssemblyLine can contain more than one Iterator, and these will be executed in succession, the second Iterator starting up once the first one reaches the end of its data set.

Lookup

This mode will cause the Connector to find and return one (or more) records, and is the mode used to *join* information into a data flow.

Table 1: Connector Modes (Continued)

Passive	Passive mode tells the AssemblyLine that this Connector is ***not to be executed during normal operation***. What's the point then? Well, let's say that you want to log all error messages to a DB2 database. You would set up a JDBC Connector to point to this data source and put it in Passive mode. Then wherever you have error handling code (and we'll take a closer look at this later) you can invoke your Passive-mode Connector to write the error-log to the database.
Update	In Update mode, a Connector will first try to find the specified record (or records). If it succeeds, then existing entries will be *modified* with the information you pass to the Connector. If the lookup fails, then the Connector will *add* the information instead.

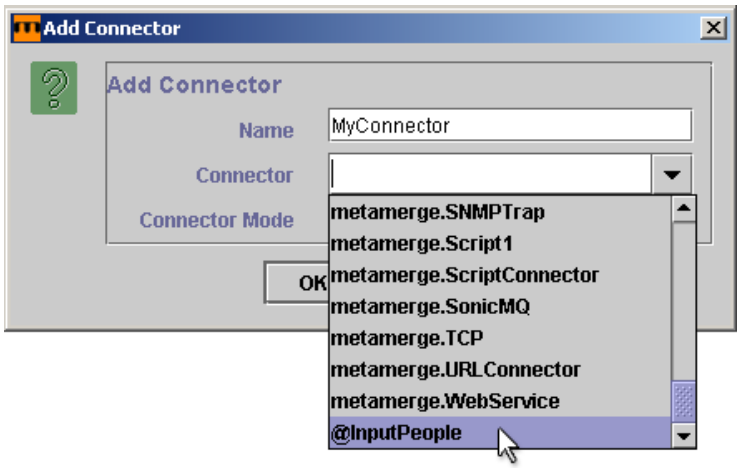
There is no limit to the number of Connectors that an AssemblyLine can hold, and you can also have as many of the same type as well — even connected to the same data source. In some cases, this is even necessary.

For example, if you want to create an AssemblyLine to delete all the records in a data source, you will first need an Iterator (e.g., a Connector in **Iterator** mode) which will run through the data set and return each entry, one and a time, for processing. These objects are passed on to your second Connector, which is of the same type, but this time set to **Delete** mode. This AssemblyLine will loop through the input source and delete each entry that it finds.

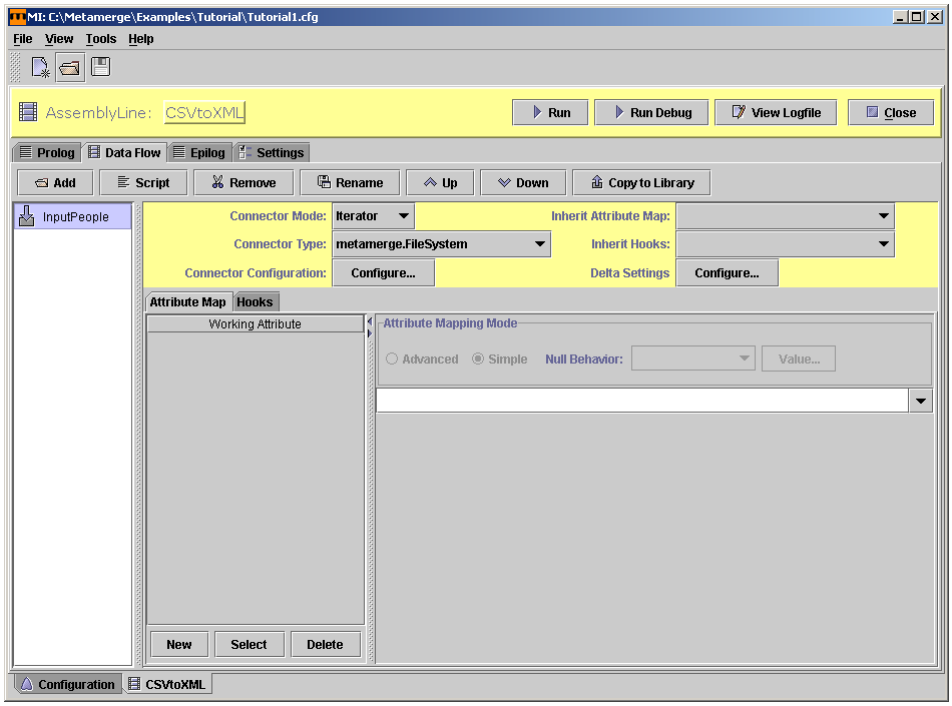
Sometimes it is undesirable to have multiple simultaneous connections to the same data source (it can even be impossible due to resource locking or limits imposed by software licenses). In this case, you can tell Integrator to *reuse* an existing connection when you set up a new Connector. This is done by scrolling all the way to the bottom of the **Connector Type** drop-down list. Here you will find the names of all the Connectors that appear in the

5. Please note that not all modes are available for all Connectors. For example, if you are working with a flat file, then Lookup, Delete and Update will not be supported. Unless of course you write your own Connector, or enhance an existing one to handle this.

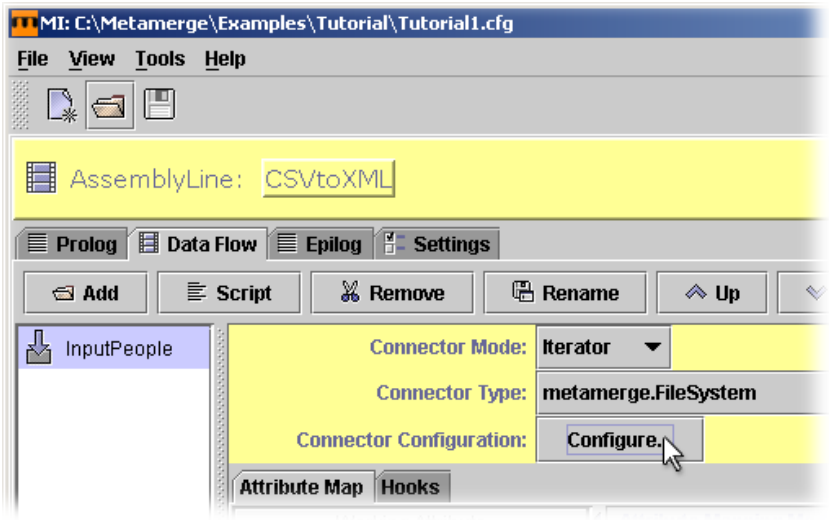
AssemblyLine before this one. These reusable Connectors appear in the list with an "***at***" (@) before their names:



Returning to our example once more, as soon you have given your Connector its name, type and mode (as we just discussed on page 23), press **OK** to confirm your choices. This new Connector will appear in the AssemblyLine Connector List at the left hand side of the window. Notice how the details for the currently selected Connector are now shown in the Details display area to the right of the list.

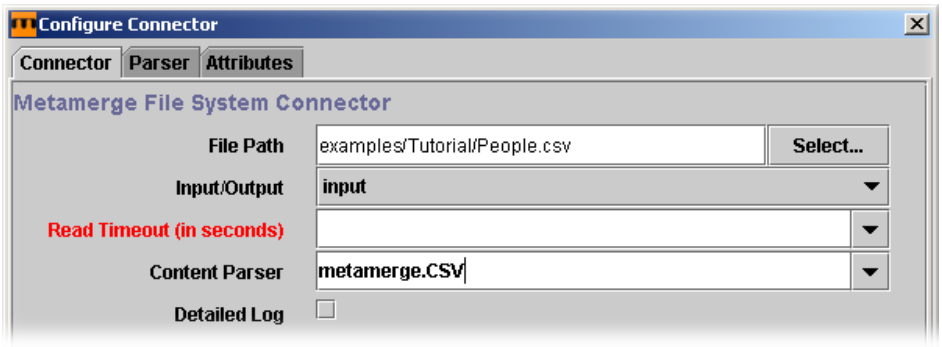


The first thing we have to do with any Connector that we add is configure it. You do this by pressing the **Configure...** button at the top area of the Connector details display.



This will bring up the Connector Configuration dialog. This dialog is closely tied to the data source we are connecting to, and will be different for each type of Connector.

The **FileSystem** Connector that we just added does not require any authentication parameters. However, it does need the pathname of the file to use, the I/O mode and, since we are now working with unstructured data, a parser to pass the bytestream through in order to format it.

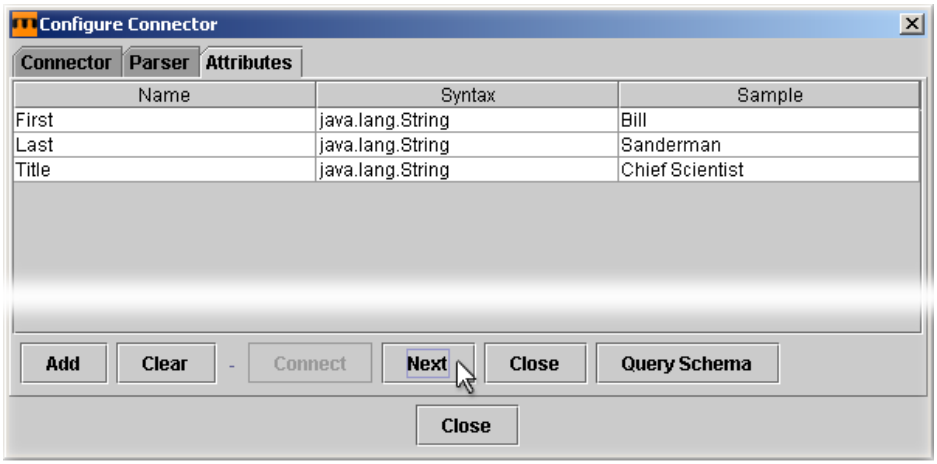


Select the People.csv file in the Tutorial directory, set the Connector to **input** mode and then choose the **CSV** parser.

Once the configuration parameters are in place, it's time to check to see if we have a live connection to the data source by selecting the **Attributes** tab at the top of this dialog.

From this tab we can press the **Connect** button at the bottom of the window and see if our Connector can reach the data source. If all goes well, then the **Next** button will become enabled. Each time we press the Next button, we are telling the system to read the next

entry in the data source, analyze the schema and then convert the source-specific data types to their relevant Java objects, e.g. strings, date/time, integers etc.



The data retrieved from the source is displayed in the grid, with the attribute names, the Java object types that Integrator is converting them to and the actual values found in the data source. Not only do we confirm that we are live with the connected system or file, but we can also do a visual control of the data being read. And don't panic if Integrator is not converting the underlying data types as you wish — you can always override this behavior when you *map* these attributes into the data flow, as you'll see in the next section.

Mapping Attributes Into The AssemblyLine

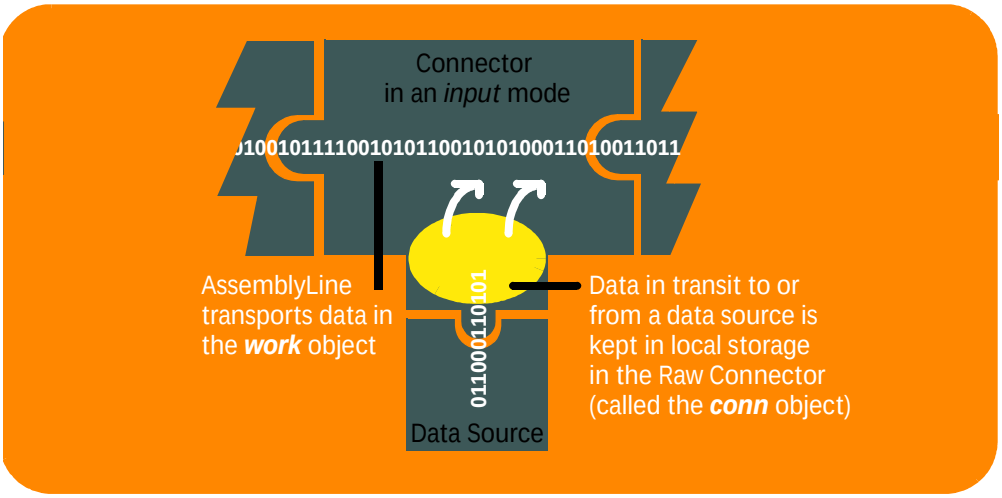
Attribute Mapping is the operation of moving information between the data source and our data flow. We just saw in the previous step how Integrator not only discovers the schema for us, it is also automatically converting the data to Java objects. So why is Attribute Mapping necessary?

There are a couple of reasons. First off, although Integrator has made the data available to you, the system has no preconceptions of how you intend to use it. So at the very least, you need to select which attributes you want to use.

In addition, some of the attributes may have to be computed, combined or created, or converted to a different format or type than Integrator has chosen. All this is all typically handled through scripting in your Attribute Map.

Connectors are actually made up of two parts: the *Raw Connector*, which knows how to talk to, and interpret responses, from a particular data source; and the generic Connector "wrapper" that allows Connectors to plug into and operate in the Integrator AssemblyLine framework.

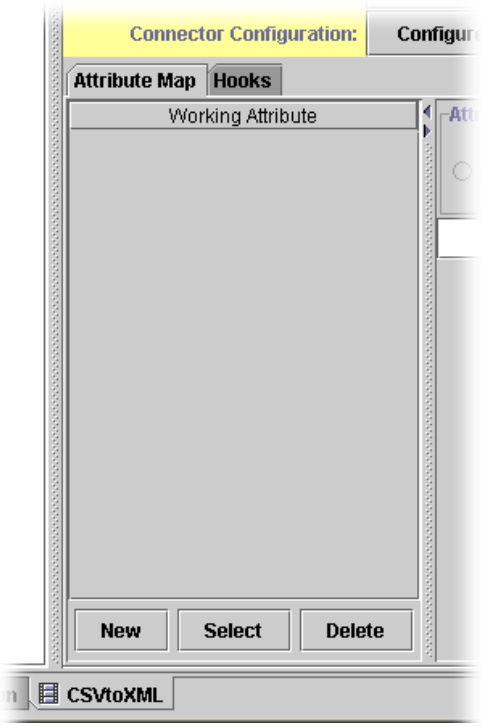
Information passed to (or from) a data source is kept in a temporary local storage object inside the Raw Connector. It is from here that we map attributes into the AssemblyLine, or in the case of an output Connector, map them out to the data source.



You can change the Raw Connector part of a Connector at any time by simply changing its type. However, if you do, you probably also need to change the Attribute Map, since scripts in your AssemblyLine, as well as other Connectors may be impacted by the change. Particularly if the schema of the new data source is different from that of the old one.

Your Attribute Map may also be affected if you change the mode of your Connector, since Connectors in an input mode map attributes from local storage in the Raw Connector (the script object called ***conn***) to the object used to store and transport data in the AssemblyLine (the ***work*** object). Output Connectors map attributes the other way: from ***work*** to ***conn***.

If we return to our example again, cast your eyes at the lower left part of the Connector display window where you'll find the Attribute Map list.

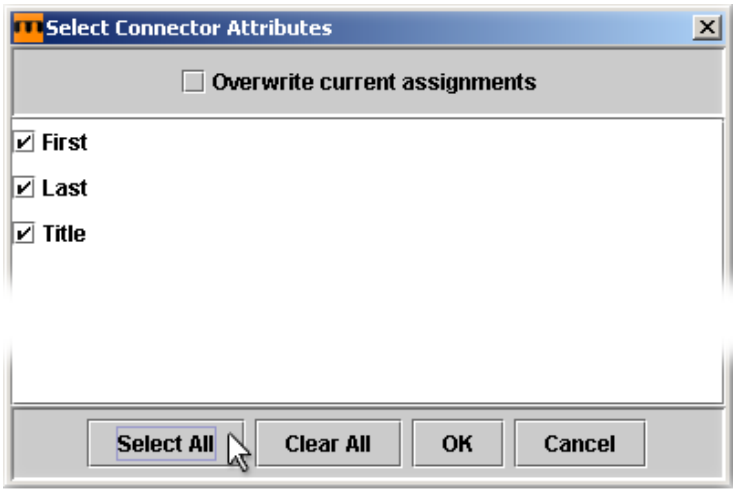


Below this list you will find three buttons:

- **New** This button you add new attributes to the list. Although the **Select** button (below) allows you to choose from a list of existing attributes, the **New** button lets you add attributes by hand that may not be available (e.g., they may need to be computed, or perhaps optional attributes that the Connector did not discover in the source).
- **Select** Pressing Select opens up a dialog with a list of attributes to select from. For in input Connector then this is the schema that Integrator discovered in the data source. In the case of an output Connector, this list will contain those attributes that are already in our data flow;
- **Delete** Removes one or more attributes.

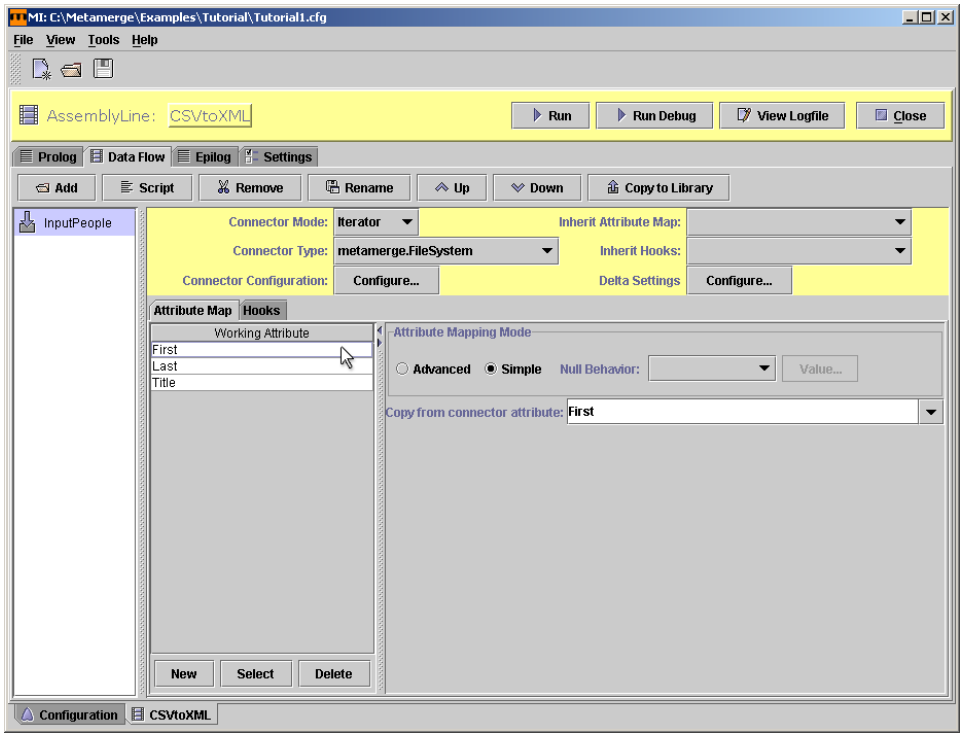
Press the **Select** button, and we'll set up the Attribute Map for our Connector. You will be presented with the Select Connector Attributes dialog, displaying the list of attributes that the system discovered when you pressed **Connect** and **Next** in the Connector Configuration dialog in the previous section.

This dialog can be used at any time to add or remove attributes from the schema that Integrator has retrieved for you.

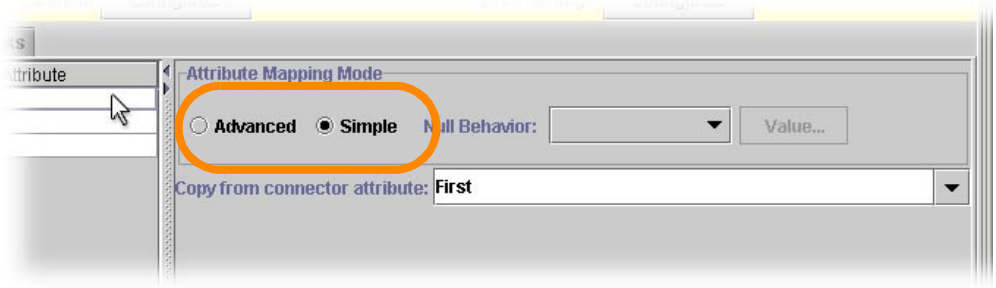


Once the window is up, you can either select the attributes that you want individually, or use the **Select All** button at the bottom of the dialog. Note that if we wanted additional

Click on **Select All** to tag all the attributes for inclusion, and then **OK** to confirm your choice and close the dialog. Now the selected attributes will appear in the Attribute Map part of the Connector Details display area.

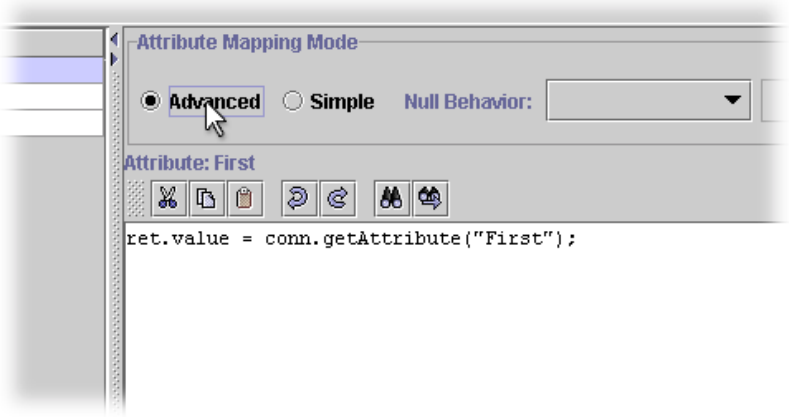


As you select attributes in this list, the details of how the mapping is performed are displayed off to the right. If you look to the top of these mapping details then you will notice two radio buttons: **Advanced** and **Simple**. As you can see, we are using simple mapping



for our attributes, which means that Integrator is copying the value(s) from the Raw Connector's attributes with the same name.

However, if we want to convert the incoming data, or otherwise manipulate these values, then we can script this ourselves in **Advanced** mode. Select Advanced mode and you'll notice something interesting: even in Simple mode, Integrator has actually written a little code snippet for us.



This line of script returns the attribute called "First" from local storage in the Raw Connector (the **conn** object).

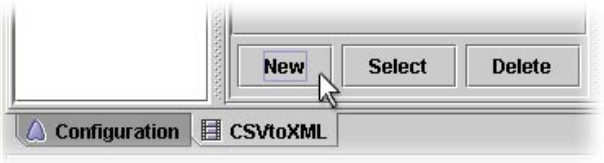
We'll take a moment to look at Integrator's script editor window, which appears wherever you need to write scripts. While in this window, you have a number of typical editor features (some of them appearing as buttons in the row above the editor window).

- **Arrow keys** move the cursor around in the editor window. If you hold the **SHIFT** key down at the same time, then you select text. Pressing **CTRL** and the left and right arrow keys moves the cursor around a word at a time. And, of course, you can combine **SHIFT** and **CTRL** to select whole words or lines of script.

- **Cut** — cuts the selection out of the text. This function is available as both the first button in the row above the editor window, as well as through the **CTRL-X** keyboard shortcut.
- **Copy** — copies the current selection. Copy can be done by selecting the second button in the button row, or pressing **CTRL-C**.
- **Paste** — pastes text (that you've copied or cut) into the script at the current position of the cursor. You can either use the third button or press **CTRL-V**.
- **Undo** — rolls back the last editing operation. This is the fourth button.
- **Redo** — reapplies the change that you just undid, Redo appears in the button row just after the Undo button
- **Find** — allows you to search for text in your script. You can either click on the sixth button (second from the right) or press **CTRL+F**.
- **Find Again** — repeats the last search, appears at the right end of the button row, and has its own keyboard shortcut: **CTRL+G**.

Time to try our hand at scripting by adding an attribute called "FullName" which we'll compute using values from other attributes read in from the input source.

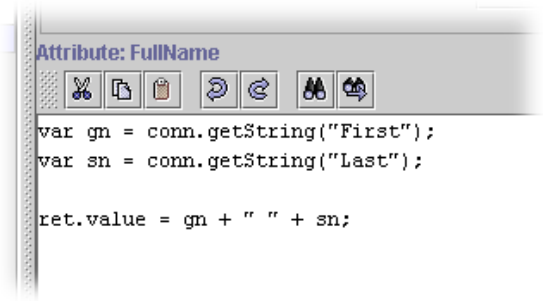
So press the **New** button at the bottom of the Attribute Map list.



In the dialog box that appears, enter the name "FullName" and press **OK**.

Integrator will automatically try to use simple mapping to retrieve the value from a data source attribute called "FullName". This won't work since this information is not available in input file.

Instead, you need to select this new entry in the Attribute Map list and click on the **Advanced** mapping radio button. This gives us access to the script editor window. Here is the script that we'll use to create the value for this attribute:



This short script does the following for us:

```
var gn = conn.getString("First");
var sn = conn.getString("Last");
```

These first two lines get the string value of the "First" and "Last" attributes and store them in local variables called `gn` and `sn` respectively.

```
ret.value = gn + " " + sn;
```

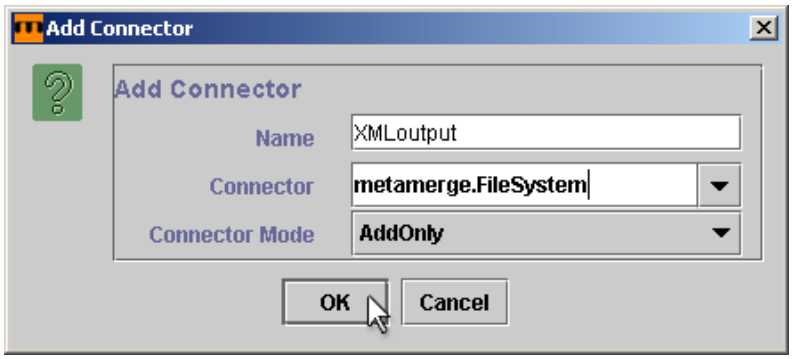
Our last statement returns the value of these two local variables concatenated together with a single space between them[6].

The input feed to our data flow is now finished: we are connecting to our input source, passing the bytestream through the CSV parser one line at a time, converting these fields to Java objects and moving this data into the AssemblyLine. It's time to add our output Connector.

6. We could have created this attribute in our output Connector. However, since we're going to need it in our AssemblyLine later, we'll put it here in the input Connector.

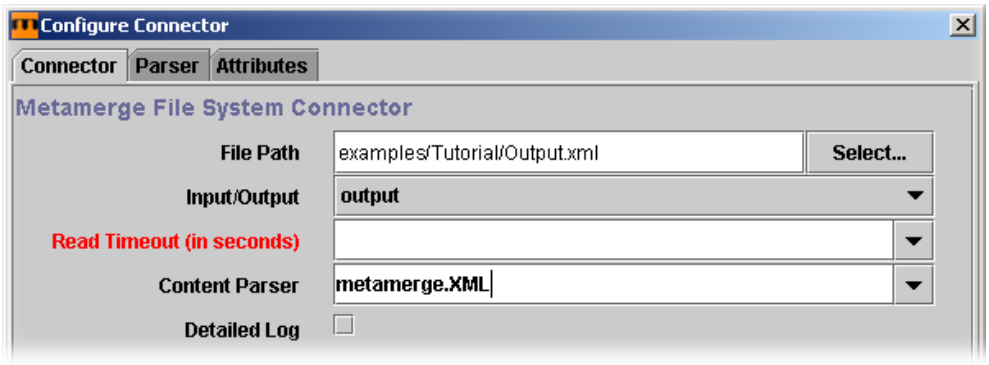
Adding The Output Connector

From the AssemblyLine Details screen, press the **Add** button again. This time we'll name our Connector "XMLoutput", we'll choose the **FileSystem** type once more, but this time set the Connector to **AddOnly** mode since we will be writing to a file.



As always, we have to configure this Connector using the **Configure** button, as we did back on page 26.

Call the output file "Output.xml" and write it to the same directory where our input file was located. Then select **Output** mode and choose the metamerge.XML parser. Once



you are finished, you can press **OK** to close the Configure Connector dialog.

Once we are back in the AssemblyLine screen, our last step is to tell the "XMLoutput" Connector which of our AssemblyLine attributes we want to write to our XML document. To do this, we use the **Select** button at the bottom of the Attribute Map list.

Integrator will now give you a warning dialog stating that you are *not* selecting attributes from the data source itself. Instead, you are about to be presented with a list over all the attributes that are currently available in our Java "bucket" (the **work** object) in our AssemblyLine.

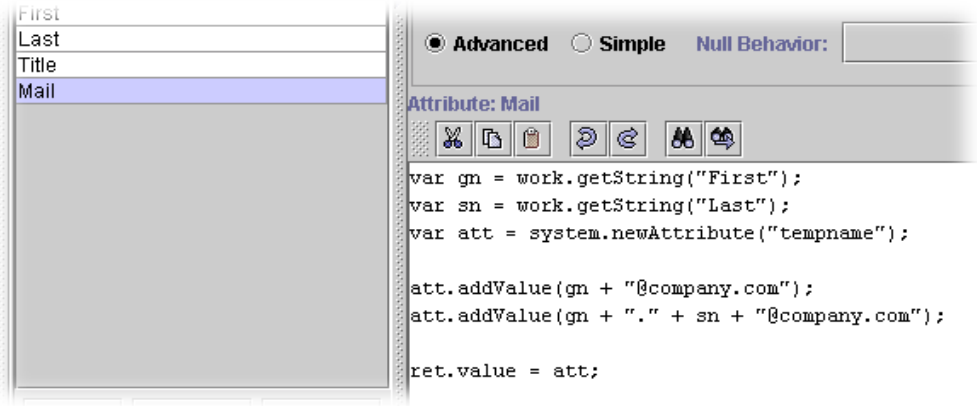
Once you press **Yes** to acknowledge the warning, you will get a familiar selection list. However, if you look at the title of this dialog, you'll see that you are now selecting from the

Work Entry (our Java "bucket"). You can use the **Select All** button here to choose all the attributes and then close the dialog.

All we have left to do now is to add the "Mail" attribute that we specified in our diagram on page 10. This will be a computed field, and we will construct this value on-the-fly (just as we did for "FullName" in the input Connector). In addition, since people often have more than on email address, we'll make this a *multi-value* attribute.

In order to do this, press the **New** button at the bottom of the Attribute Map list. This will open a dialog asking you to name the new attribute. Call it "Mail" and press **OK**.

Now press the **Advanced** Attribute Mapping radio button and enter the script shown below.



Let's walk through this script:

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

These first two lines store the values of the "First" and "Last" attributes in local variables. Notice how we are using the work object to access data inside the AssemblyLine.

```
var att = system.newAttribute("tempname");
```

This next line uses a system call to create a new attribute. We have to give it a temporary name, although this will not be transferred when we return the value at the end of the script.

```
att.addValue(gn + "@company.com");
att.addValue(gn + "." + sn + "@company.com");
```

These two lines compute and add two values to this attribute, making it a *multi-value* attribute.

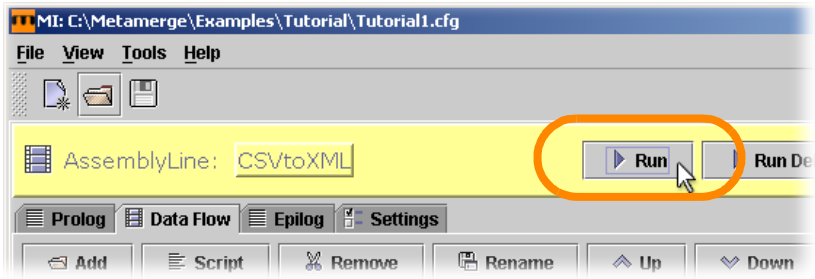
```
ret.value = att;
```

Finally, we return the newly created attribute. The Integrator framework will deal with this for us, converting the complex object to the format of the output source.

At last we are ready to watch our first data flow implementation in action.

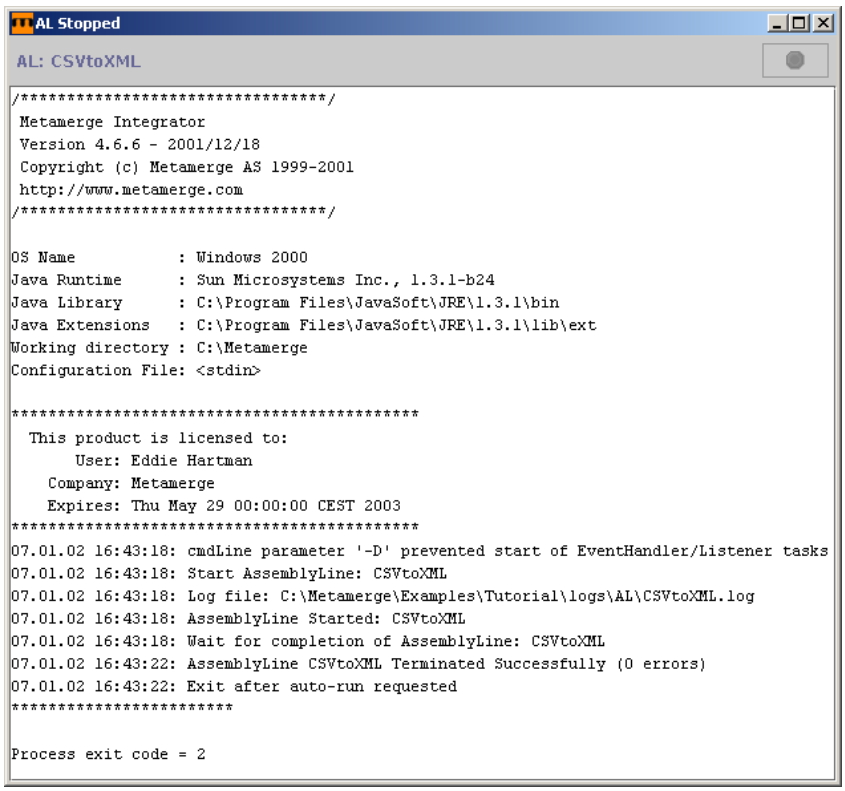
Running Your AssemblyLine

Your AssemblyLine is now complete and ready to test. To do this, press the **Run** button in the Object Button Row at the top of the AssemblyLine Details area[7].



7. Before you run your AssemblyLine, you may want to go to **View | Preferences** and check the box entitled "**reuse command window**". This will tell Integrator not to open a new window each time your run your AssemblyLine.

When you tell Integrator to run an AssemblyLine, the system starts up an instance of the server and then pipes the current configuration to it[8]. Integrator will also create a monitor window that the server writes its status output to.



Not counting the "Process exit code" line at the bottom that simply tells us that the server stopped after executing the specified AssemblyLine, the output in this window is divided into four main parts:

- First comes some information about the version of the server that you are running.
- After this is a section that describes the environment that Integrator is running in, including which VM it is configured to use, and the working directory.
- The third part shows our license information.
- The fourth and last area the monitor output is usually the biggest, and tells you a number of things: which parameters where used to start the server; the configuration file that is being used; and finally messages generated during the execution of the AssemblyLine and its Connectors. You can send messages to this screen as well, using special Integrator objects and functions available when writing your scripts.

8. Of course, you can start up the server from outside the Admin Tool and instruct it which configuration file to use. For more information, check out the online Customer Pages of our website.

At the bottom of this last section is the message that our AssemblyLine (CSVtoXML) ran without errors. That means that we should be able to open the output file that we specified back on page 34.

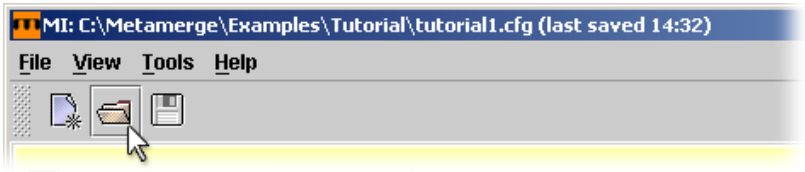
Opening this file (e.g., in a browser) will allow you to confirm that the AssemblyLine has actually converted the CSV input data to the XML document.

We can see that our input data was correctly read in from the CSV file, and then passed to our output Connector which wrote it to the XML document. Even our "Mail" attribute is there, computed on-the-fly for us by our script.

However, one of the entries ("*Roger*") appears to be incomplete. This entry is lacking both the "Last" and "Title" attributes. If we check our input data file (listed on page 14) then we can see that these fields are actually missing from our input CSV file.

The easiest solution would be to edit the CSV file and add the missing fields; However, few data sources will give us this much control. So instead, we are going to try filtering our input by scripting a *Hook*.

Before we start evolving our AssemblyLine, let's save our work first by either pressing the Save button in the Main Button bar, by selecting ***File | Save*** *from the Main Menu, or by*

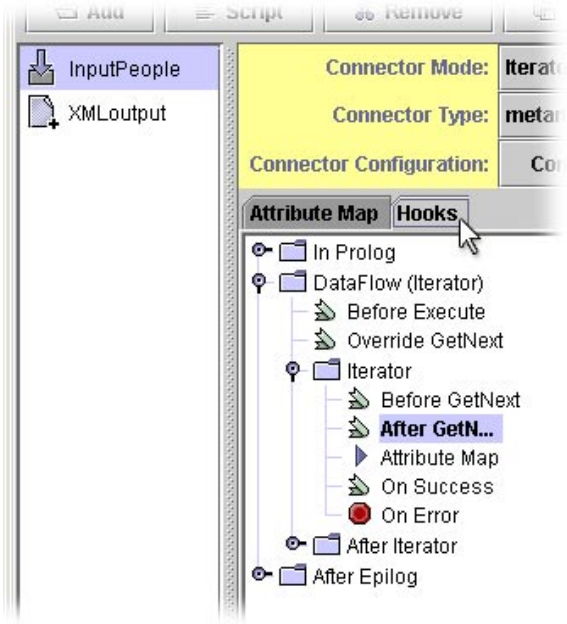


pressing ***Ctrl+S****. Each time you save your configuration file, the last save time value is updated in the title bar.*

Working With Hooks

Hooks are waypoints in a Connector's work cycle where you can add logic to be executed each time Integrator reaches that point; Like before a Delete, or after an Add operation. The hooked script will run until completion before the Connector continues its work.

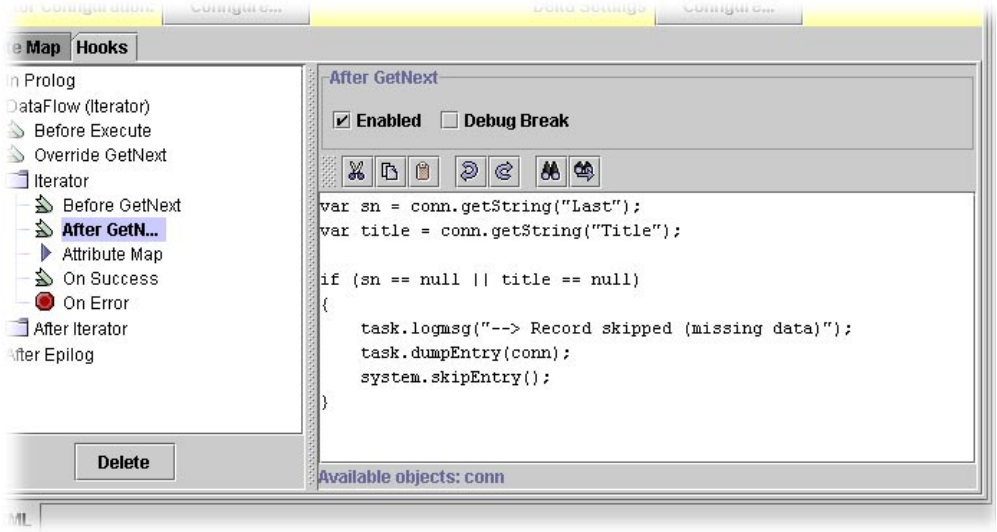
There are a number of Hooks that are common to all Connectors, plus a few that are mode-specific. In order to work with the Hooks of the "InputPeople" Connector, first select it in the Connector List and then click on tab labeled **Hooks** (right next to **Attribute Map**).



There are three sets of Hooks for every Connector, represented by folders in the Hooks tree-list:

- **In Prolog**, These scripts are fired up when the AssemblyLine first starts, meaning they are run only once;
- **DataFlow** Here is where you'll find the Hooks that are executed at every iteration of the AssemblyLine, each time this Connector is run. In the above example, since the "InputPeople" Connector is in **Iterator** mode then this means that it will be doing a number of *GetNext* operations to retrieve the input data. As a result, Integrator gives us Hooks like **Before GetNext** and **After GetNext** so that we can wrap this read operation in our own logic;
- **After Epilog** Hooks are executed once at the end of the AssemblyLine's life-cycle.

So, with the "InputPeople" Connector selected, click on the **After GetNext** Hook in the list. Now enter the following script in the edit window to the right of the Hooks list:



Stepping through the script, let's see what our filtering code is doing:

```
var sn = conn.getString("Last");
var title = conn.getString("Title");
```

These first two lines are retrieving the values (as strings) of two attributes that are available in the Raw Connector (the **conn** object).

```
if (sn == null)
{
```

Then we check to see if the value returned for the "Last" attribute is *null*[9], meaning that it does not exist in the input data source. If this is the case, then the next three lines are executed.

```
    task.logmsg("--> Record skipped (missing data)");
```

The **task** object gives us access to AssemblyLine functions, like logmsg() which lets us write to the AssemblyLine's logfile.

```
    task.dumpEntry(conn);
```

This time we use the AssemblyLine's *dumpEntry()* function to write the contents of the Raw Connector's local storage object (**conn**) to the logfile. Note that this function works equally well for output Connectors, except that we would be passing it the **work** object instead.

9. We will probably want to test for the existence of more fields in the production version of our AssemblyLine.

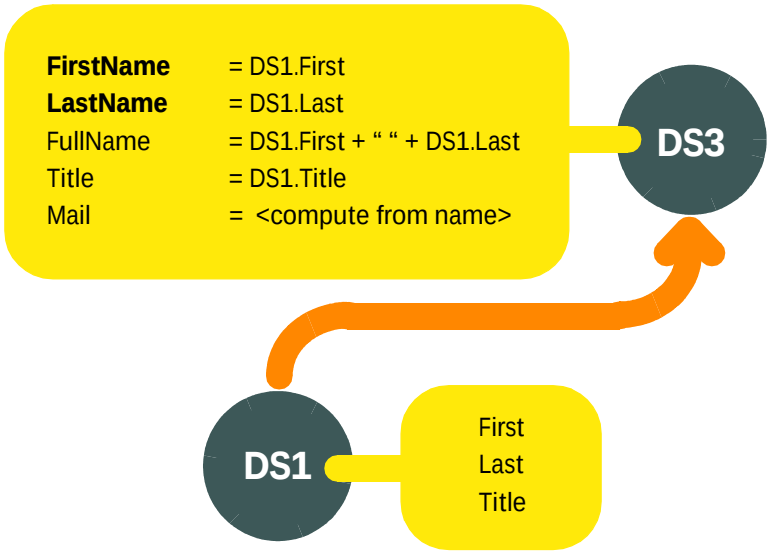
```
        system.skipEntry();
    }
```

Finally, we use the **system** object to signal to Integrator that we want to skip this input entry, and start back at the top of the AssemblyLine loop and read the next one.

Before we test our line again, we're going to make a slight change to the output Attribute Map.

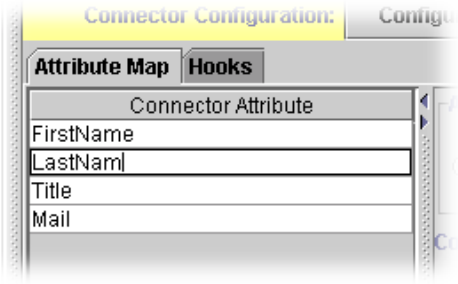
Schema Conversion

In our example, the output attributes happened to have the same names as those in the input source. But let's imagine for a moment that our specification had called for the output attributes to be called "FirstName" and "LastName".



Integrator makes mapping schemas easy, and all we need to do is update the names of these attributes directly in the Attribute Map of our output Connector.

So, select the "XMLoutput" Connector, click on the attribute that you want to change and start typing.



And don't worry about this affecting scripts in your AssemblyLine, since we are only changing these names locally for the Attribute Mapping phase of the output Connector; The "First" and "Last" attributes are still being read in correctly, and are available inside the AssemblyLine[10].

We'll leave these changes in (even though it's not actually part of our original specification) and then run our AssemblyLine again. When Integrator completes, go back to the output browser window and hit the refresh button If you closed that window, you'll have to repeat the steps you used to open the output file after the first run.

10. Because Integrator keeps focus in the field that you are entering, even if you switch to a different Connector or AssemblyLine, you may need to press **ENTER** (or otherwise shift focus to another Attribute) so that Integrator knows that you are finished with your changes before you trying running the AssemblyLine again.

Once our output file is visible again, we can confirm that "Roger" is no longer there (he used to be between "Jill" and "Gregory"). Furthermore, we can see the changes that we made to the names of two of our attributes.

We'll save our work again (**CTRL+S**), and then go to the next step: Aggregating data from a third data source.

Adding The Join Connector

Included with the Tutorial files is a simple database over people who owe us money (important information, this!). We'll use this source to do get information about our debtors into the output source.

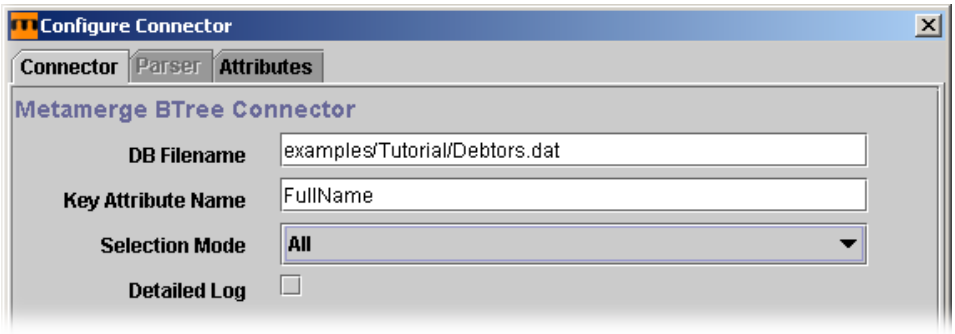
Our first step is to add a third Connector by pressing the **Add** button in the AssemblyLine Button Bar. Call this one "Debtors" and choose the metamerge.BTreeObjectDB Connector type. This Connector needs to be in **Lookup** mode since we'll be searching for records that match the data inside our AssemblyLine.

We were working with the "XMLoutput" Connector last, so when we add one to the list, Integrator drops our new Connector right after the one currently selected. But this won't work, because we need to do the aggregation between the input and output Connectors.

To fix this, press the Up button in the AssemblyLine button row.

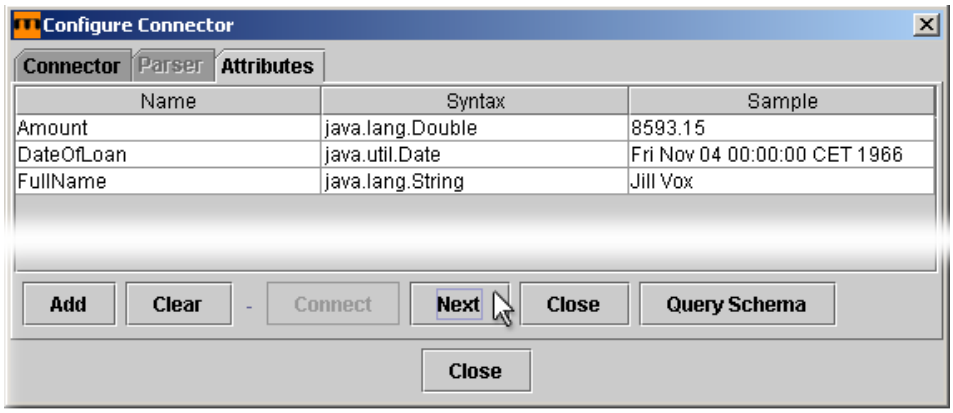
This moves our Connector up one slot so that it will be executed after "InputPeople", but before "XMLoutput".

Once again we press the **Configure...** button to set up this Connector.



Enter the pathname of the data file, which may be different from the one above, depending on where you installed Integrator (note that the database file itself is called "Debtors.dat"). In the **Key Attribute Name** field you need to specify the name of the attribute that differentiates these records. In our tutorial database this is the "FullName" attribute (which should give you a hint as to why we constructed a similarly named attribute in our input Connector).

To test our Connector, we select the **Attributes** tab, press **Connect** and then **Next**. You should now be able to see the data in this source.



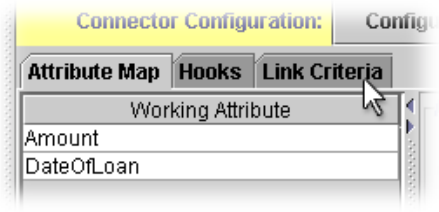
Again, Integrator is reading the schema and converting the data accordingly to Java objects. You can close this dialog now.

Now we set up the Attribute Map by pressing the **Select** button under the Attribute Map list, and then pick only "Amount" and "DateOfLoan" from the list presented[11]. That's all we need, plus we already have an attribute named "FullName" so we don't need to aggregate this data into our AssemblyLine.

Setting Up Link Criteria

Because it's in **Lookup** mode, our new "Debtors" Connector will be searching for specific entries in its data source, trying to find a match for the entry that is already inside the AssemblyLine. Exactly how this match will be made is specified by us in what is called the Connector's *Link Criteria*.

If you take a look at the AssemblyLine screen (still in the DataFlow tab for our AssemblyLine), you'll see that because our Connector is in Lookup mode, we've got a new tab next to **Attribute Map** and **Hooks**.

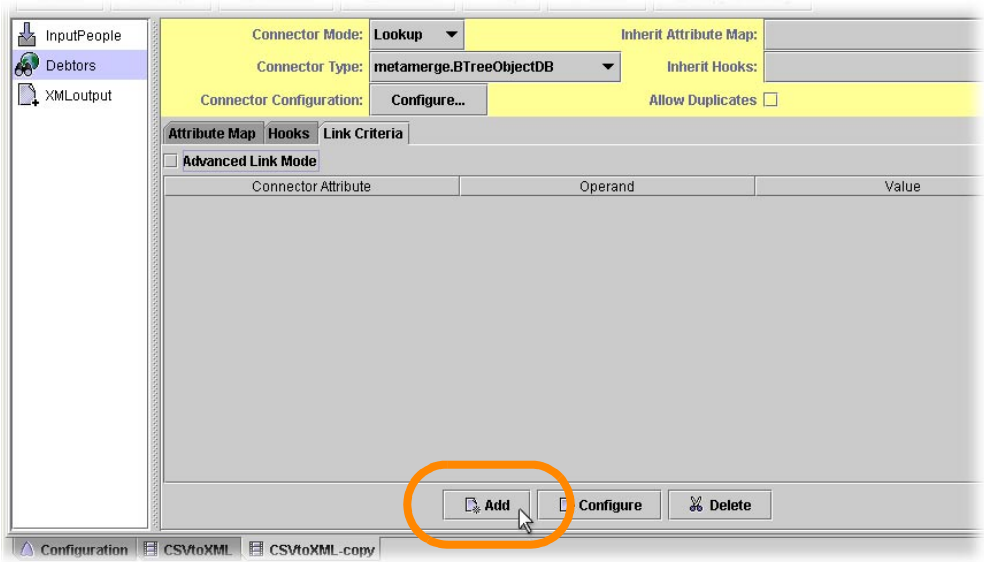


Selecting this tab brings up the **Link Criteria** display where we can specify how this Connector is to do its lookup.

11. As you'll see in the following section, we are about to use this data source's "FullName" field to set up our Link Criteria. However, an attribute does does not need to be included in our Attribute Map in order for us to use it as part of a Link Criteria.

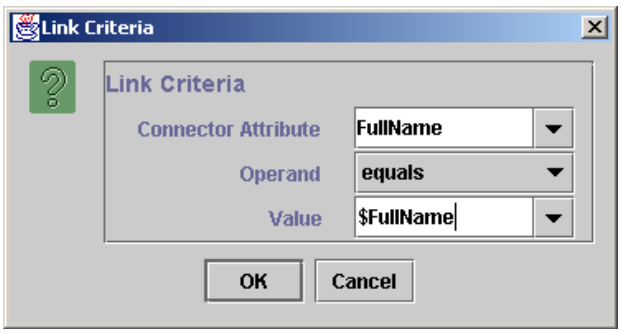
Now, do you recall how we looked at Integrator's feature for scripting an Attribute Map directly ourselves? The same applies here, so by clicking the **Advanced Link Mode** checkbox (just under the **Link Criteria** tab itself) we get an Editor window where we can write the data source specific *lookup* call ourselves. This could be an SQL SELECT statement for a JDBC Connector, or an LDAP search call if we where connected to directory.

However, just as with Attribute Mapping, Integrator can handle the details of this for us automatically, creating the relevant command for the underlying data source, and keeping our solution a bit more data source independent.



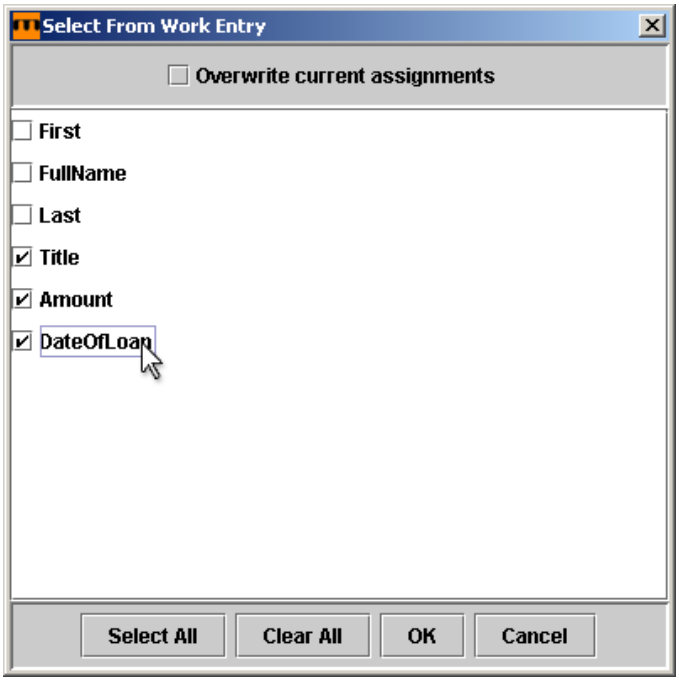
We'll use this feature and simply press the **Add** button at the bottom of the screen (shown in the screenshot above).

When the Link Criteria dialog box appears, we first choose an attribute from the schema that Integrator discovered in the data source. Then we select a comparison operation (like **equals** or **contains**). The last field lets us specify the attribute inside our AssemblyLine to be compared. to.



For our example, we'll choose the "FullName" attribute from our Debtors data source, the **equals** operation, and then the "FullName"[12] attribute that we scripted in our input Connector. Press the **OK** button when you're done.

Once the dialog is closed and you are back in the AssemblyLine screen, select the "XMLoutput" Connector so that we can update its Attribute Map to include the data that we're aggregating into the line. Do this by once again pressing the **Select** button under the **Attribute Map** list, clicking past the warning again, and then choosing the new "Amount" and "DateOfLoan" attributes.

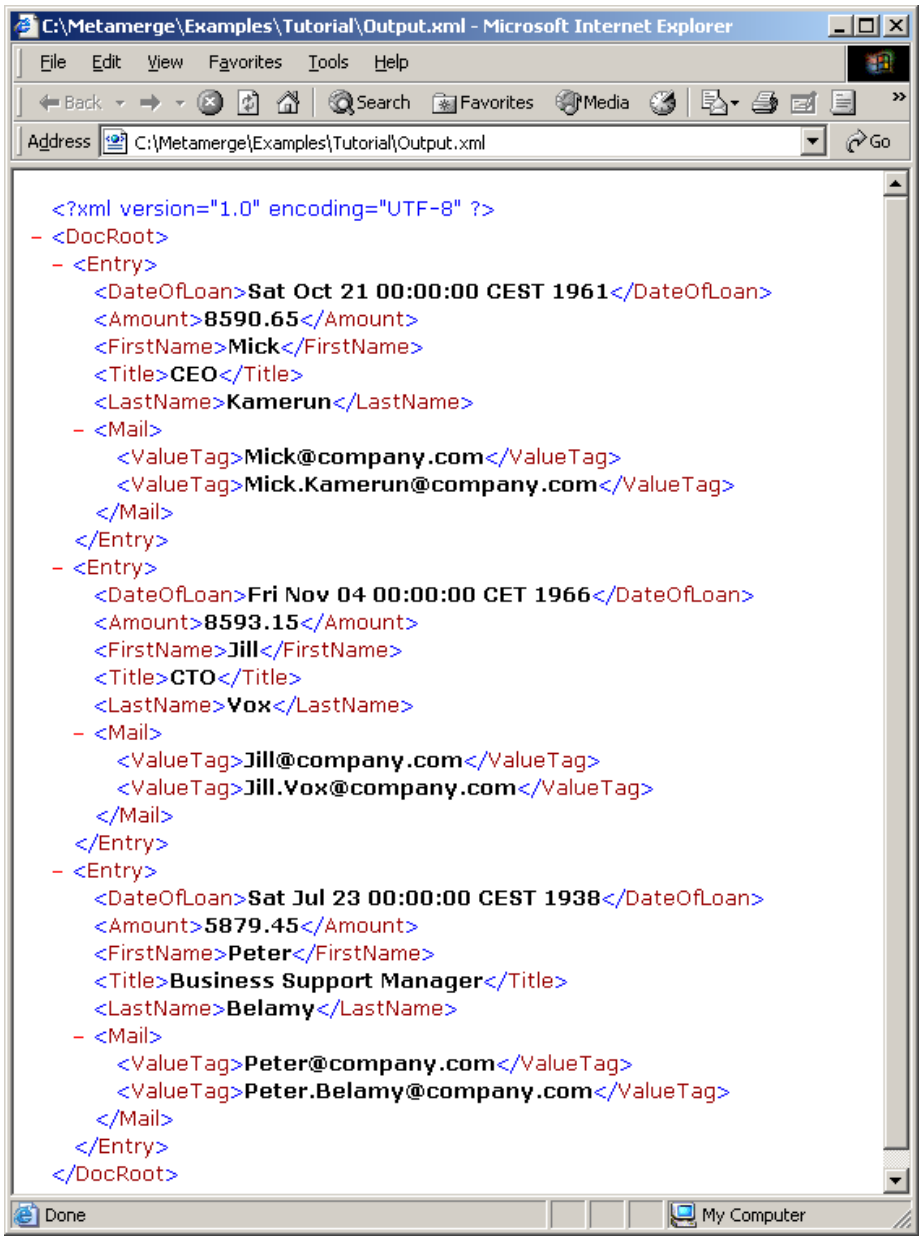


You may be wondering why the "First" and "Last" attributes no longer appear to be selected. That's because when Integrator builds this list, it compares the current Attribute Map with the list of available attributes to see what's already been included. Since we've changed the names of the "First" and "Last" attributes for our output map, Integrator can't find them in our Attribute Map anymore, and un-checks them in the selection dialog. If you were to click on them, then these attributes would be added to the mapping list with their original names again.

Now you can close this dialog box, save your work and run the AssemblyLine again.

12. You probably noticed the **dollar sign ($)** in front of the "FullName" attribute coming from our AssemblyLine. This special character causes Integrator to retrieve the first value of this attribute (it might have any number of values) to use in building the Link Criteria. If we wanted to match any one of the values of a multi-value attribute, we would use the **at sign (@)** instead.

Our XML document should now look like this[13]:

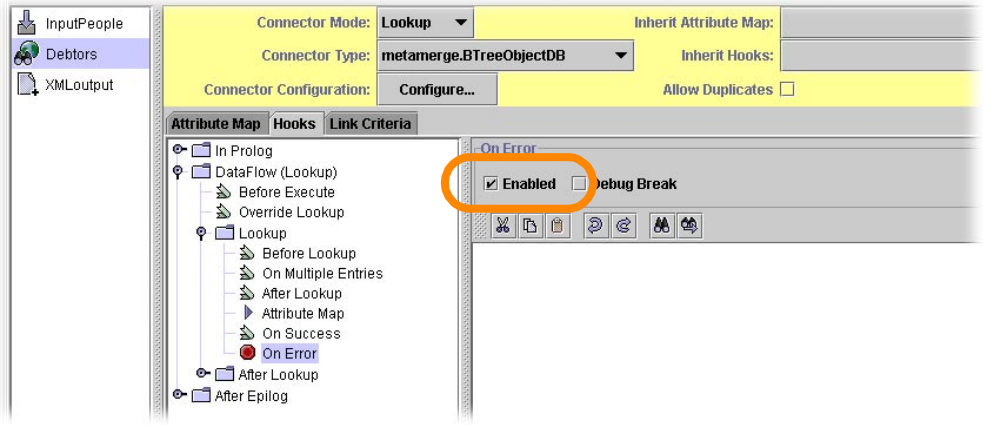


We can see a couple of important changes here: First, we can see the two new "Debtors" fields that we included in the output. Secondly, and possibly a bit of a surprise, we notice that our XML file now includes only three entries.

13. Please note that the order in which the attributes for each entry appear is not relevant, and may not be the same on your system as it appears in the screenshots of this manual.

That's because these are the only entries in our input source ("InputPeople") that Integrator managed to match with data in our aggregation source ("Debtors"). When our **Lookup** Connector did not find matching information in the database, it faulted to the AssemblyLine error handler which then skipped this entry — like when we used the system.skipEntry() function in our input filter script.

If this is not what we want, then all we have to do is enable the **On Error** Hook of our "Debtors" Connector.



Instead of sending the *Lookup Failed* error to the AssemblyLine error handler, Integrator detects that we have error handling code of our own and uses it instead. Of course, we will probably want to do something a bit more intelligent than just enabling an empty script Hook.

In addition, remember how we filtered out "Roger" during our initial input? This could cause a situation where the person or people filtered during input actually owed us money, but our AssemblyLine never gets that far with these records.

As an alternative, we could have created default values for missing attributes in our first Connector. Integrator gives you a number of ways to handle missing values.

However, this won't help much in our case, since we want real values for both "First" and "Last" in order to create a "FullName" attribute which we later need in the "Debtors" Link Criteria.

Hmmm. Back to the drawing board.

Soapbox Once More

As you can see, although Integrator makes building our data flows fast and easy, the quality of the resulting solution is dependent on how good our specification is. But Integrator actually helps us here as well by removing the platform and vendor technology blinders that block our vision and limit our imagination.

And when you approach an integration problem at the data flow level, you reduce complexity. This gives you gains across the board: in deployment speed, accuracy of the solution, robustness, maintainability... The list goes on. In fact, as you start to think in terms of Metamerge's *simplify and solve* mantra, you will see your installation, and its integration possibilities, from a whole new perspective.

Integrator also keeps making a difference long after your solution is finished and deployed; Because as your business and technical requirements change, Integrator lets you enhance and evolve your solution to meet these new challenges. That's the beauty of Integrator: incremental implementation. It means that you can grow your integration solution (and your infrastructure) to fit your needs, as well as the environment where it's going to live.

Perception is reality, and our perception is formed — and limited — by the toolset we use. The choice is simple: 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.