Chapter 4

Establishing the Framework

A multi-tiered application needs a solid framework. Since you're working at many different levels, it's crucial that you have a common framework that crosses the tiers. Such a framework is hard to develop when you have multiple languages in the mix, but EGL provides a very simple solution: the record.

Defining the Tiers

Earlier chapters introduced the concept of application tiers. Now it's time to define them very concretely. Figure 4.1 shows the tiers in an EGL Rich UI environment. (You might remember these tiers from Chapter 2.)

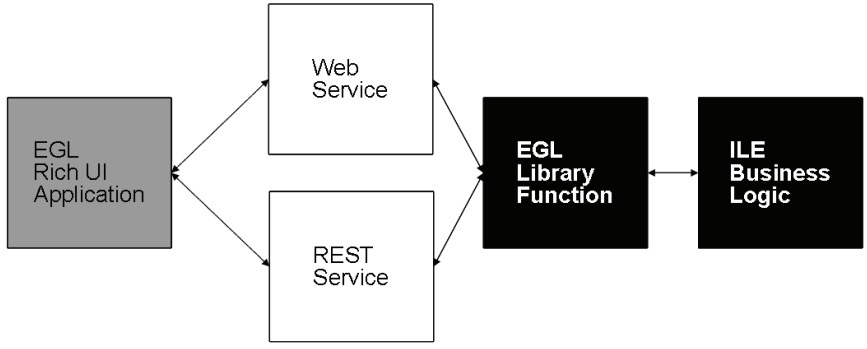

Figure 4.1: These are the components of an EGL Rich UI application.

While this is not a particularly complex design, it does need a little more detail. This is shown in Figure 4.2.

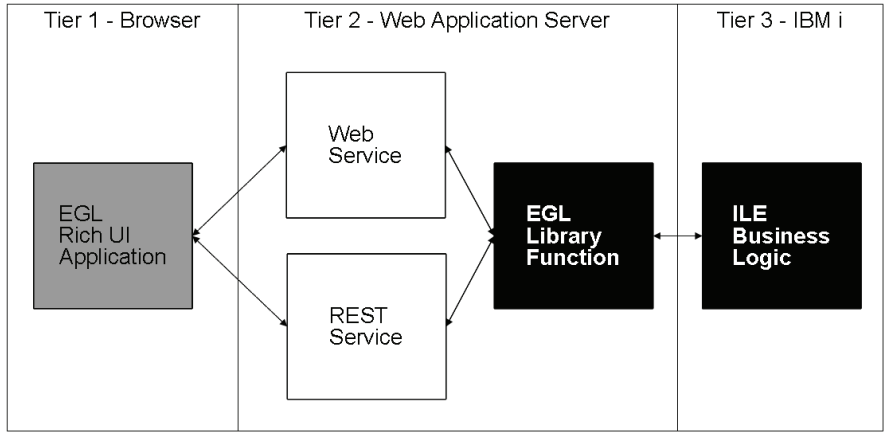

Figure 4.2: Here are the same components, segregated by tier.

As you can see, the EGL Rich UI component runs in Tier 1, which in turn runs in the browser, while the ILE runs in Tier 3 on the IBM i. Tier 2 acts as the bridge between those two very different worlds. This tier is really the anchor for the entire application.

First Steps

When I design an application like the one in Figure 4.2, I follow some very simple procedures. First, I try to get an idea of what I want to see in the user interface. Obviously, I need to get the users involved at some point, but I usually have enough information from the basic definition of the problem to make a first pass at the interface.

For example, consider a project to create a simple order-entry front end, something along the lines of an Internet storefront. I might have a design document that tells me the fields I need, or maybe just the request to "make it look like such-and-such.com." Whatever the input, I sketch out a list of fields needed.

You might remember the records described in Chapter 2. They are from the mental list of fields in Table 4.1.

Table 4.1: Two Records, Each with a List of Fields	
Order Header Fields	**Order Detail Fields**
Order Number	Item Number
Customer Number	Description
Customer Name	Quantity
Shipping Address	Price
Tax	Extended
Freight	
Total	

Is this an exhaustive list? Certainly not, but it's more than enough to get me started. You might have noticed that some of the fields could be calculated, such as "Total" in the order header. That's not important in the first phase of the design. The goal here is to get something visible, as quickly as possible. It's not hard to add or change fields later in the process.

You might also have noticed that I don't include the order number in the order detail. That's because the order lines will be part of the larger order structure, which will also include an order header, which will in turn have an order number. Although you might feel more comfortable including the order number on every line, keep in mind that it introduces unwanted redundancy. I suppose the most persuasive argument against redundant data is that the goal is to minimize the data traffic, since these messages are used to communicate between tiers, potentially over relatively slow connection speeds.

Adding a New Project for Tier 2

The next thing to do is to create a project for Tier 2 of Figure 4.2. Eventually, I'll also create a project for Tier 1, the Rich UI browser component. I don't need an EGL project for Tier 3 because that is ILE code on the IBM i. I will be able to edit, compile, and test that code using the IBM i tools integrated into RDi-SOA. For now, let's concentrate on the Tier 2 code.

In Chapter 3, you saw how to create a simple EGL Rich UI project. Figure 4.3 uses the same menu option to create an EGL web project. (This is the New/Project option from the Project Explorer's context menu, or from the

File menu in the main menu bar.) In this example, though, Web Project is selected from the radio buttons to create a web project named "iEGL."

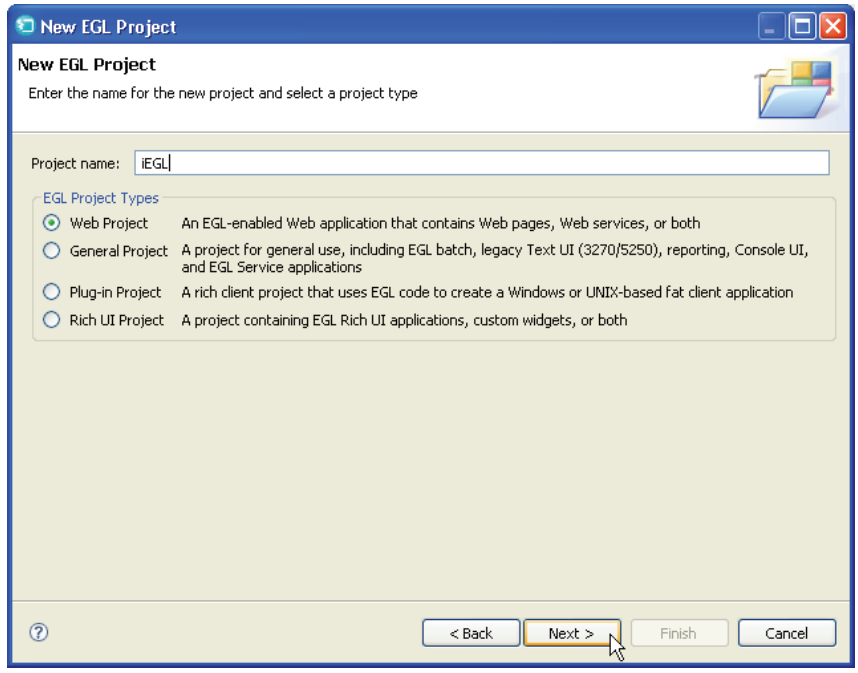

Figure 4.3: Create an EGL project, but be sure to select the Web Project option.

The only thing to worry about in a typical setup is which WebSphere runtime to use. Figure 4.4 shows the configuration for a WAS 7.0 project. It's important to select the runtime you'll be deploying to when you create an EGL project (or any web-based project). Otherwise, you won't be able to run your project in production.

Once the Finish button is clicked in Figure 4.4, RDi-SOA will create the iEGL web project. By default, it will also create an EAR (Enterprise Archive) project named "iEGLEAR." You can change the name of the EAR project, along with some other attributes, in the Advanced tab. However, I rarely find that necessary.

Packaging the Application

Now it's time to actually describe the data. Before doing that, however, I have to make some decisions about how to package the application. That

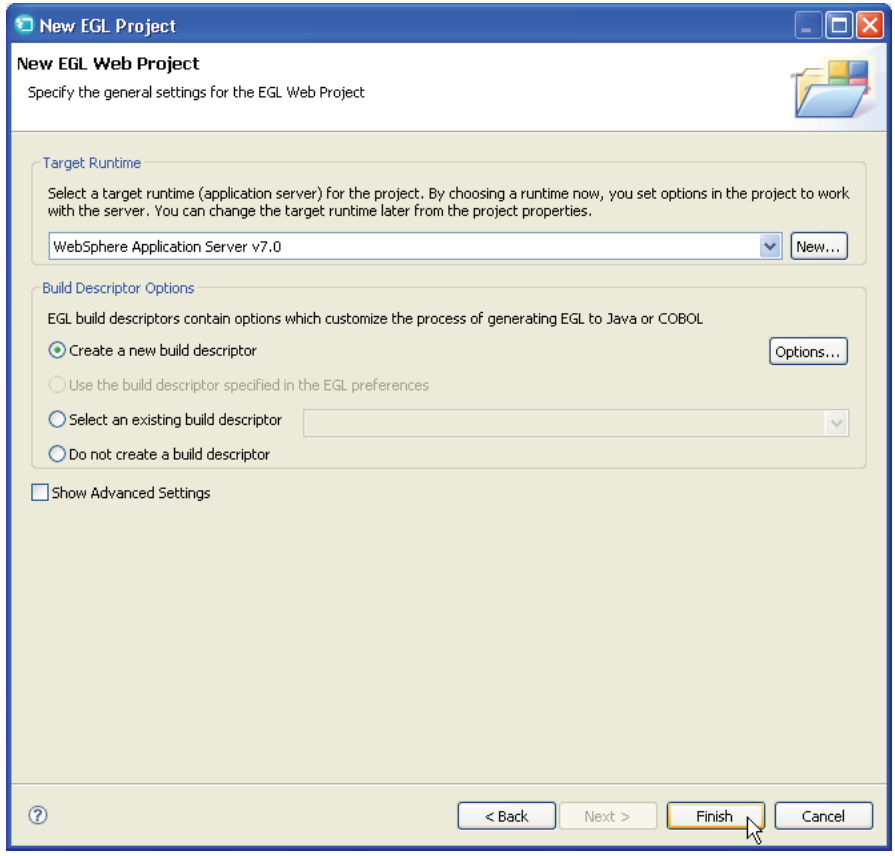

Figure 4.4: Target Runtime is the important option on the second panel.

is, how will I define the packages in my EGL source code? The IBM i side is relatively simple; I create a library and store my programs there. I might have several libraries for different applications, and I probably have a source library separate from my program library, but that's all second nature for an ILE developer. The package concept is a little more off the beaten path, so let me take a moment to tell you how I like to do things.

First, I need to create the overall package name. EGL uses the same dot-notation for naming that Java uses, so it makes sense to use Java-style conventions. In Java, you always use "reverse domain" naming for your packages. That is, if your corporate domain is xyzcorp.com, you start any of your custom packages with "com.xyzcorp." The trick, though, is to come up with a good convention for the parts after the reverse domain name. If you've already got good standards for Java code, you might

be able to extend them. If you're coming straight from the green-screen world, though, you're unlikely to have such a naming system in place. Then what do you do?

Well, a lot depends on your expected application design. Let's go through the naming logic I use for the packages in this particular book.

I use the basic packages listed in Table 4.2. There are two things to keep in mind: first, the com.pbd.app package is typically only used in Tier 1, the EGL Rich UI browser component. The com.pbd.data and com.pbd.bl packages reside only in Tier 2, the EGL service portion. The two other packages live in both packages, though for slightly different reasons.

Table 4.2: Package Names

Package	Contents
com.pbd.bl	Business Logic
com.pbd.data	Data Definitions
com.pbd.svc	Exposed Services
com.pbd.util	Utility Functions
com.pbd.app	Application Programs

The com.pbd.util package is simple enough. It contains utility functions, some of which are specific to the tier, and some of which are shared. For example, it's a good place to put extensions to the EGL language, but extensions to Tier 1 are made in JavaScript, while extensions to Tier 2 use Java. Tier 1 extensions don't make sense in Tier 2, and vice versa.

The com.pbd.svc package is a little more specific to my own programming style. In Tier 2, I use com.pbd.svc to expose EGL business logic from the com.pbd.bl package. In Tier 1, however, I use the com.pbd.svc package to hold the proxy libraries that use the services in Tier 2's com.pbd.svc to communicate with the libraries in Tier 2's com.pbd.bl. The proxy libraries have the same name as the business logic libraries in Tier 2, but they use services in Tier 2 to call the library functions. As an example, OrderLib in Tier 1's com.pbd.svc calls OrderService in Tier 2's com.pbd.svc, which in turn calls **OrderLib** in Tier 2.

This means that if you need to get an order, you call getOrder, regardless of your tier. In Tier 2 (e.g., the thin client), you call getOrder from OrderLib in com.pbd.bl. In Tier 1, you call getOrder from OrderLib in com.pbd.svc, which in turn calls getOrder in OrderService in com.pbd.svc in Tier 2, which finally calls getOrder from Orderlib in com.pbd.bl. All very simple, right?

Certainly, this is not the only way to do things, but for me, it ties the components together nicely. I suppose an equally valid option might be to name the package "com.pbd.ifc" in Tier 1, but I like it the way it is. You'll see the whole hierarchy in use when the tiers are tied together.

Creating Placeholders in Tier 2

The first thing to do to tie the tiers together is create placeholders in Tier 2. To do this, I create data definitions in com.pbd.data and a simple "get" function in com.pbd.bl that will return hardcoded information. In minutes, I can have a complete test environment up and running. (It really takes me longer to describe it than to do it!) The code for this chapter contains four simple source files, one each in three of the standard packages, and a fourth one in a special test package, which you'll learn about a little later.

Let's take a look at the new parts in the three standard packages. First, a member called Order is added to the package com.pbd.data. Typically, I'll add one part (one source file) for each set of related records. Sometimes this will only be a single record, such as a customer record. It's unlikely that a customer will have a very complex structure, so it's probably represented by a single database record. An order, on the other hand, is almost always made up of multiple records. At the very least, you'll have an order header and order detail.

Let's take a look the parts for an order. Here is the part Order.egl in the package com.pbd.data:

```
package com.pbd.data;
record Order{}
  Header OrderHeader;
  Lines OrderLine[]
end
```

```
record OrderHeader{}
  OrderNumber string;
  CustomerNumber decimal(6,0);
  CustomerName string;
  ShippingAddr string;
  Tax money(9,2);
  Freight money(9,2);
  Total money(9,2);
end
record OrderLine{}
  ItemNumber string;
  Description string;
  Quantity decimal(9,2);
  Price money(9,2);
  Extended money(11,2);
end
```

You can see how easy it is to represent complex data types in EGL. After the **package** statement, you immediately see the definitions of the data, starting with the definition of the complex Order structure. Order is made up of an OrderHeader record and an array of OrderLine records. One nice thing about EGL is that arrays are self-sizing; you don't have to muck around with determining an optimum maximum value for an array.

You might notice that the code uses **String** for all character fields. That's because EGL, especially in the Rich UI component, isn't really built on fixed-length data. The whole idea of a fixed-length string has positive and negative connotations. It's obviously easier to define data when you don't have to worry about the length, but it's harder to set up your user interface when you don't know the width of a given field. Tradeoffs exist, as always.

The next thing is to create what I call the "placeholder" function. This is a simple data-access function, designed to support the actions that will eventually be required by the application. Typically, the first thing you need is a fetch function (or a "getter," if you prefer object-oriented terms). In this example, I will create a part that has a function to get an order.

While I put the data definitions in my data package, the placeholders go into the business logic package. (If you remember, that's the one named com.pbd.bl.) The part that has the functions directly related to a given record has a name derived from that record; in this case, the part for Order functions is OrderLib. OrderLib looks like this:

```
package com.pbd.bl;
import com.pbd.data.*;
import com.pbd.util.*;
library OrderLib type BasicLibrary {}
   // Get an order
   function getOrder(orderNumber string in, order Order inOut)
     returns (Error)
     order = new Order { Header = new OrderHeader {
       OrderNumber = orderNumber,
       CustomerNumber = 789,
       CustomerName = "Pluta Brothers Design, Inc.",
       ShippingAddr = "542 E. Cunningham, Palatine, IL, 60074",
       Tax = 17.19,
       Freight = 14.95,
       Total = (17.19 + 14.95 + 4.32 + 23.95 + 9.45)
    }, Lines = [
       new OrderLine {
          ItemNumber = "AS-1445", Description = "Squirt Guns",
          Quantity = 36, Price = .12, Extended = 4.32 },
       new OrderLine {
          ItemNumber = "IIR-7728", Description = "Wading Pool",
          Quantity = 1, Price = 23.95, Extended = 23.95 },
       new OrderLine {
          ItemNumber = "IIR-7243", Description = "Metal Ladder",
          Quantity = 1, Price = 9.45, Extended = 9.45 }
    ]};
    return (null);
  end
end
```

The function is very simple, although it shows off a number of very interesting features of EGL. For example, EGL supports bidirectional parameters, and even allows you to control the directionality of those parameters. In this case, the getOrder function has two parameters: the key, orderNumber, is an input-only string, and the order itself is a bidirectional parameter of type Order. Many of these capabilities become even more important in the world of SOA, because they allow you very fine-grained control over the amount of data sent back and forth.

Initializing Data in EGL

This function also shows how easy it is to create and fully initialize complex data structures with EGL. Although it might take you a few moments to work your way through the syntax, you'll find that the entire function is

made up of only two statements. The first statement creates the order and encompasses the first 19 lines of the function. The second line returns a null value (something I'll get back to momentarily). For now, focus on the first line, if you will.

Even though the Order record is complex, with both a nested record *and* a nested array of records, it is still relatively easy to create a fully initialized order. That's because of EGL's simple, keyword-based approach to initialization. For example, if I already had a record of type OrderHeader named orderHeader, and an array of lines named orderLines, I could have done this:

```
order = new Order { Header = orderHeader, lines = orderLines };
```

That's all it would take. Note that the field names defined in the record (in the Order.egl listing earlier in this chapter) can be used as keywords to assign values when creating a new variable of that record type. By using curly braces, { }, after the **new Order** syntax, you can now specify one or more fields using their names as keywords. This is quite spectacular, actually; the EGL editor is smart enough to make use of the code in other EGL source to edit the current source. You can also use *auto-completion*. If you press Ctrl-Space within the braces, you will get a list of the fields that (still) need to be initialized, as shown in Figure 4.5.

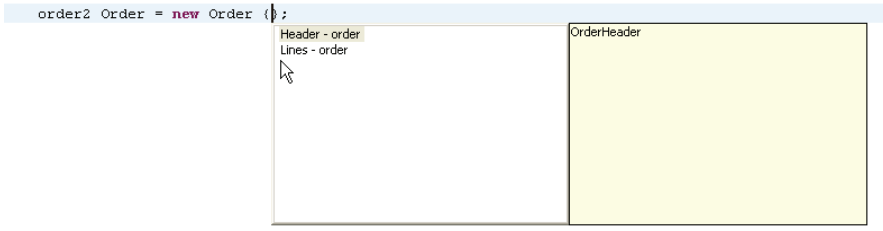

Figure 4.5: Auto-complete works very nicely when initializing complex structures.

What really makes your job easier is that, for complex structures, the initialization code can be nested. An example of that is shown in the code for the OrderLib.egl part. First, the Order record is initialized using an instance of new Order, with the two keywords **Header** and **Lines** (similar to the one-line snippet in the previous paragraph). Then, each

of those fields is initialized by further use of the new keyword. **Header** is initialized by creating a new OrderHeader record and then setting each of the fields in the OrderHeader (OrderNumber, CustomerNumber, and so on). Even more fascinating is that you can initialize the array by simply using square brackets and then defining a list of OrderDetail records (each using new and then setting the values for its own inner fields). What might have taken a whole bunch of initialization lines and a number of work variables can all be done quickly and easily using this syntax.

Suppose you tried to do the same thing for Java records. You'd have to create a class for each component, and each class would need a constructor that takes all of the variables. When initializing each level, you'd have to remember the order of the constructor's parameters and specify every one (or at least null). The keyword syntax in EGL lets you easily change the order of initialization and leave out fields you don't need to initialize. To do the same thing in Java, you'd have to have multiple constructors.

The point is that with EGL, it's very, very easy to create dummy data. That's crucial to testing. It's easy to set up even the most complex data, including data that tests boundary conditions or bad data, and use that to rigorously test your code.

Handling Errors

I glossed over the issue of bad data earlier, simply because this topic needed its own section. One of the only things I'm not thrilled about with EGL is the lack of good, custom error-handling. I'd like to have the try/catch capabilities of Java, but since they're not available, I had to come up with my own technique.

I've created a standard error record, named "Error." This record is the return value for every function that can fail, or at least whose failure I need to be able to recognize, handle, or at least report cleanly. The Error record is contained in com.pbd.util, in the Error part. It looks like this:

```
package com.pbd.util;
record Error
  severity int;
  message string;
end
```

Right now, there's almost no support; it's just a bare-bones record with a couple of fields, the severity and the message. However, neither one of these is really defined to any degree. My primary purpose for creating this part is to define my functions. The Error record will get more capabilities and support later in the project.

So, now I can go on creating functions. As you can see in Table 4.3, all are variations on the same basic structure.

Table 4.3: Functions

Function	Input Parameters	Output Parameters
Get (single)	Unique key	Instance of record
Get (multiple)	Selection criteria	Array of record
Put/Update	Record to write	n/a
Delete	Unique key	n/a

You can see that the getOrder function follows this structure. It has an input parameter of the orderNumber and a bidirectional Order record to return the result, and it returns an Error record to indicate the outcome of the operation.

Note that none of these routines has a completion code or status parameter. They are all assumed to complete successfully. Instead, they all return an Error record. If a function returns a null value, then the function completed successfully. Otherwise, the error information is in the Error record.

It can get a little more complex for editing; instead of a simple error message, you might have to return more information. That's not a problem, however; you can easily extend the Error record if needed. For the purposes of this exercise, though, a single error string will suffice.

Testing

Now that I have defined the basic framework for my functions, I can return to the task at hand, which in this case is testing the function. It's really easy. I can test using a simple EGL program:

```
package test;
import com.pbd.bl.*;
import com.pbd.util.*;
import com.pbd.data.*;
program Test1 type BasicProgram {}

  function main()
    order Order;
    error Error = OrderLib.getOrder("ABC654", order);
    writeStdout(
      "Order: " :: order.Header.OrderNumber ::
      ", lines: " :: order.Lines.getSize());
  end

end
```

I usually create test programs in a separate package that isn't part of the standard reverse-domain hierarchy. The test programs are internal components that should never escape the lab, so they can be defined using a different, simple package name; such as "test." That's what I've done here.

The program itself is very simple, which makes sense given how little actual logic I've actually written thus far. This code creates an empty order, and then invokes the getOrder function from OrderLib. Upon completion, it writes out the order number from the newly retrieved order. This sort of test program takes almost no time to create and is really easy to run using the context menu. The **writeStdout** command sends data straight to the console.

If you're new to EGL, you might be wondering about the syntax of the **writeStdout** command. Basically, **writeStdout** will output any sort of data. You use the double-colon operator, ::, to concatenate values. (This is similar to the two vertical bars, ||, in CL or SQL.)

The context menu for an EGL program part (one that has the type **BasicProgram**) has the Debug EGL Program option enabled, as shown in

Figure 4.6. Select it, and the workbench will run your program, sending any output to the console.

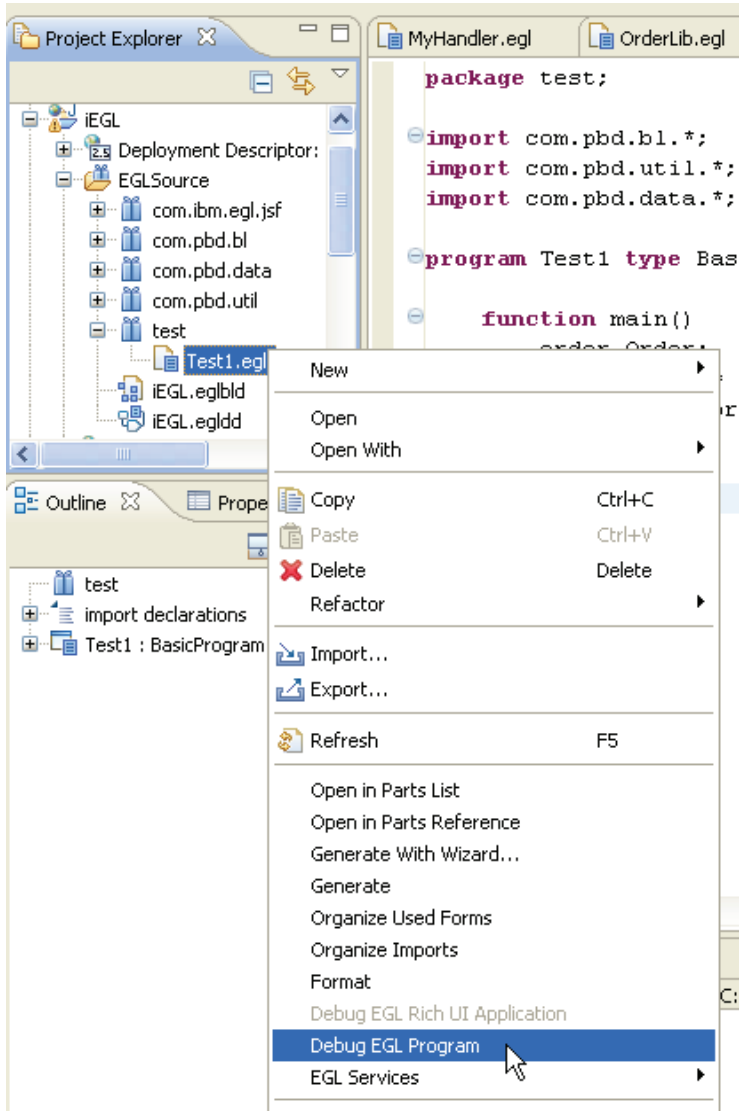

Figure 4.6: Right-clicking the test program gives you the option to debug an EGL program.

As you can see in Figure 4.7, the output shows the order number and the number of lines.

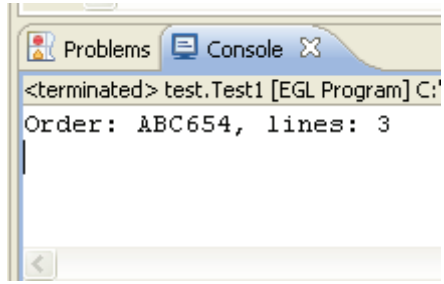

Figure 4.7: The output of the program is shown here.

The order number is actually the one passed into the program. If you review the code for the getOrder function, you'll see that I initialize the order number from the input parameter:

```
function getOrder(orderNumber string in, order Order inOut)
  returns (Error)    order = new Order { Header = new OrderHeader {
    OrderNumber = orderNumber,
```

The last line sets the OrderNumber field in the OrderHeader record from the orderNumber parameter on the function.

The number of lines is calculated by getting the size of the Lines array.

Summary

That's all there is to it. Just like that, your program is running, and your library function is tested. What I want you to take away from this chapter is that you don't need a user interface to create your business logic. While at first this seems almost counterintuitive, it's actually the underlying premise of this entire book: your user interface and your business logic should be fundamentally independent of one another.

Achieving complete independence is of course, impossible; your tiers must have some knowledge of one another. But you can strive for it with the only binding between the layers being your messages. In EGL, those messages are Records which are passed between the tiers. By writing your tiers to those Records, they can remain as independent as possible. And as I'll demonstrate throughout the rest of the book, that also means that you can build and evolve your user interface as needed without having to rewrite

your business logic, and that's the very definition of leveraging your legacy assets.

Now you've got a test environment that provides a foundation for your business logic and that you can test without having to commit to any particular user interface strategy. Of course, a hardcoded test program that dumps to the console isn't always the best or even the easiest way to test a routine. The next chapter shows you another option.