Chapter 9
Debugging NOI Agents

Those of you who have used LotusScript to create Agents or event handlers in the Notes user interface are by now familiar with the LotusScript Integrated Development Environment that comes with the Notes Client. You get a LotusScript editor and debugger all in one. Unfortunately, Domino 4.6 does not include an IDE for the new Java interface. As you saw in Chapter 7, creating Java applications that use NOI is not really a problem: You're writing a standalone program, and you can use whatever development tools you like, so long as the Notes libraries and jar files that you need are available. In Chapter 8 you saw that Java Agents for Domino are developed outside the Notes Client and imported into a database using the Import Class Files button in the Agent Builder UI.

You can certainly use a third-party development tool to create your Java Agent: You do the type-compile-fix cycle until you get a clean compile, then you're ready to import the class(es) into Domino. But how do you debug it? Your Agent has no main() function, so you can't run it from the command line, and no third-party tool will have a clue as to how to deal with an Agent either. Only Domino can launch a Java Agent with the proper context all set up, but Domino doesn't (yet) provide a debugger. What's a frustrated developer to do? There are some obvious choices:

1. Use lots of System.out.println calls to trace the execution of the Agent. This is fine for small programs, but if you're going to use this technique (which harks back to the early days of C programming, when the only debuggers available were at the assembly language level, if you were lucky), you have to put in a compile time "isDebug()" call around each println so that later, when you switch over to production mode, your users aren't staring at all your debug messages. Then you have to

recompile the Agent before deploying it, to reset your internal debug flag. This is definitely a suboptimal solution.

2. Write your Agent so that it can be used two different ways: as a real Agent, or as a standalone program. The advantage of this approach is that you can use a real Java debugger. The problem with it is that you can't debug the Agent the way it will be really working: complete with Agent context.

The approach I'm going to suggest here is a refinement of option 2: You can code your Agent so it operates as a standalone program so that you can use a real debugger; you can also *fake* the context part by writing just a bit of extra code. It is possible to write an Agent that, in a sense, has multiple personalities. It has a main() method so it can run as a standalone program, and it extends AgentBase and has a NotesMain() method, so it can be a real Agent ("it's a floor wax *and* a dessert topping"). As a standalone program it can also have a hard-wired Agent context, which is the real key. Our goal is to have a version of the Agent that can be debugged as a standalone program, but which doesn't have to be recoded to work as a "real" Agent. Read on.

Sample Debuggable Agent

Original Agent

Let's start by reprinting a sample Agent from Chapter 8 in Listing 9.1, then we'll rewrite it a bit to be debuggable. The Agent is real simple, but this technique can be applied to a program of any complexity. This Agent was originally Ex81Agent.java.

Listing 9.1 Original Sample Agent (Ex81Agent.java)

```
import java.lang.*;
import java.io.*;
import lotus.notes.*;

public class Ex81Agent extends AgentBase
{
```

```
    public void NotesMain()
        {
        try {
            Session s = this.getSession();
            AgentContext ctx = s.getAgentContext();
            Database db = ctx.getCurrentDatabase();
            DocumentCollection dc =
ctx.getUnprocessedDocuments();
            System.out.println("Found " + dc.getCount() + "
selected docs");
            Newsletter nl = s.createNewsletter(dc);
            nl.setSubjectItemName("Subject");
            Document doc = nl.formatMsgWithDoclinks(db);
            doc.send(ctx.getEffectiveUserName());
            System.out.println("Newsletter sent to " +

ctx.getEffectiveUserName());
            }
        catch (Exception e) { e.printStackTrace(); }
        }
}  // end class
```

Okay, now how do we give this a main() so that it can be invoked from the command line, or from a Java development tool? In order to do that, we need to write a main() and have it initialize Notes properly. But if that's all we do, the program will never run. Why? Because the first thing the NotesMain() method (normally the place where the Agent is first invoked by the AgentBase base class) does is call AgentBase.getSession() to retrieve a Session instance. If we run from the command line, AgentBase hasn't been set up as it normally would be by Domino, so there will be no Session instance, and our program will bomb out with a NullPointerException right away.

Never fear though, we can code around that by creating a new constructor for our Agent class that takes a Session instance as an argument, and by then overriding

AgentBase's getSession() call to return the right thing (see Listing 9.2). I've added a main(), changed the name of the class, and added a new constructor. This is an intermediate form of the Agent we're working toward, so it isn't included in the CD (the final version is included).

Listing 9.2 First Revision of the Agent

```
import java.lang.*;
import java.io.*;
import lotus.notes.*;

public class Ex90Agent extends AgentBase
{
    private Session my_session;

    public static void main(String argv[])
        {
        try {
            NotesThread.sinitThread();
            Session temp = Session.newInstance();
            Ex90Agent agent = new Ex90Agent(temp);
            agent.NotesMain();
            }
        catch (Exception e) { e.printStackTrace(); }
        finally { NotesThread.stermThread(); }
        }  // end main

    public Ex90Agent() { super(); }
    public Ex90Agent(Session s) { this.my_session = s; }
    public Session getSession()
        {
        if (this.my_session == null)
            return super.getSession();
        else return this.my_session;
        }

    public void NotesMain()
```

```
        {
        try {
            Session s = this.getSession();
            AgentContext ctx = s.getAgentContext();
            System.out.println("User is " +
ctx.getEffectiveUserName());
            Database db = ctx.getCurrentDatabase();
            DocumentCollection dc =
ctx.getUnprocessedDocuments();
            System.out.println("Found " + dc.getCount() + "
selected docs");
            Newsletter nl = s.createNewsletter(dc);
            nl.setSubjectItemName("Subject");
            Document doc = nl.formatMsgWithDoclinks(db);
            doc.send(ctx.getEffectiveUserName());
            System.out.println("Newsletter sent to " +

ctx.getEffectiveUserName());
            }
        catch (Exception e) { e.printStackTrace(); }
        }
}  // end class
```

Our new main() is pretty standard: initialize Notes, create a Session, create an instance of our class (passing the session into the constructor). Then, instead of calling agent.start() as we normally would, we call agent.NotesMain() directly. Why can't we still call agent.start(), and just have NotesMain() run on another thread? Because in "debug mode" we're entering our Agent program through main(), not through AgentBase, and when we do that our AgentBase instance is not properly initialized (it is normally initialized from Notes when an Agent is started). Were we to call agent.start(), we'd get a NullPointerException somewhere in AgentBase at the point where the code accesses a member variable that it expects to be there, but in this case is *null*.

Note also that we now provide a nondefault constructor for the Ex90Agent class, so that our main() can pass a Session instance in to it. And, having written a nondefault constructor, we have to write a default constructor as well.

The next piece of code we rewrote was to implement a getSession() override. Normally our NotesMain() method just calls the getSession() that's implemented in AgentBase, and we get the Session instance that has been all set up with an AgentContext for us. This version of getSession() is designed so that it will work equally well in both our standalone/debug case and in the "real Agent" case. All it does is return the cached Session instance if there is one (debug case), or else call the method we would normally call anyway in AgentBase to get the Session (that's the line "return super.getSession();").

So what happens when we actually run this from the command line? Let's step through it so far.

1. We type "java Ex90Agent" in a command window.
2. Java invokes our main() procedure.
3. main() initializes Notes, then creates a Session instance as you normally would for a standalone Application.
4. main() creates an instance of Ex90Agent, passing in the Session instance. The Ex90Agent constructor saves the Session instance in a private member variable.
5. main() invokes NotesMain() on Ex90Agent.
6. NotesMain() calls this.getSession(), which returns the Session cached in step 4.
7. NotesMain() calls Session.getAgentContext() and then AgentContext.getEffectiveuserName().
8. At this point, Java raises a NullPointerException, because the kind of Session that we created [using the static call Session.newInstance()] doesn't contain any Agent context information. Only Sessions created by AgentBase can contain a real Agent context.

Still, not to worry, we're making real progress here. The next thing we need to do is to fake up some Agent context. To do that we have to create our own versions of the Session and AgentContext classes. And what better way to do that, given that we're using a great object-oriented language here, than to simply "specialize" the existing classes by inheriting from them?

Debug Class Extensions: DbgSession

We'll make our two new classes *public*, meaning that they each have to be coded in their own file. We could have made them private to the Ex90Agent class, but that would really limit their reusability. If we make them first-class objects, then we can use them for all our Agent debugging needs without recoding them each time. See Listings 9.2 and 9.3, and the corresponding files on the CD. The versions here (so far) are skeletons, to which we'll be adding more methods in just a bit.

Listing 9.2 Extended Session Class (DbgSession.java)

```
import lotus.notes.*;
public class DbgSession extends lotus.notes.Session
{
    private Session s;
    DbgAgentContext context;

    public DbgSession() throws NotesException
        {
        super(1);  // any number will do

        // get a "real" session to redirect calls to
        this.s = Session.newInstance();
        }

    public AgentContext getAgentContext() throws
NotesException
        {
        // return one of our debug versions
```

```
            if (this.context != null)
                return this.context;
            return new DbgAgentContext(this.s);
            }
    } // end class
```

The only method we're overriding (so far) in this class is getAgentContext(). We want to return our debuggable AgentContext instance, not the normal NOI one. Our debuggable Session class acts as a wrapper for the real Session class, and we can redirect calls to our instance to the real one as needed. That way we don't have to re-implement all the logic of the Session class ourselves. Because DbgSession is also a Session (it inherits from Session, therefore it is one) we don't have to change any of the declarations in the main() function (or anywhere else for the most part) that use Session; assigning an instance of DbgSession to a variable declared as Session is perfectly kosher.

You'll note, however, that DbgSession both inherits from Session and creates a real Session instance that it can redirect calls to. What gives with that, why do both? The answer lies in the interface from the Java NOI classes to the C++ implementation in the Notes code.

Normally each NOI Java object instance, of whatever class, is really acting as a wrapper for a C++ class implemented inside Notes core. The Java instances keep track of pointers to C++ objects and redirect all the method invocations into the Notes code. When we create our DbgSession instance, we use the constructor you see above. It has to call its base class constructor (in the Session class). Were we to call the default Session class constructor we'd hit a safety check in all the NOI classes. Since the Session class was originally designed to be instantiated only by the static newInstance() method, the default constructor for Session makes a call to System.exit() to terminate the current process. Looking back, I wish we'd done it differently, because it would have made writing this chapter on debugging Agents a bit easier. However, as it stands in Domino

4.6, you can't call the default constructor of any of the NOI classes, as each was meant to be strictly contained by a parent object (with the exception of Session, which has a static creation method).

So, to make a long story short, we have to use a nondefault Session base class constructor from DbgSession, one which expects a handle to a C++ object as an argument. Of course, we have no such C++ handle to give it, so we make up a number (1 in this case; any number will do). That works okay, but we now have to be absolutely sure that we never invoke a method on our DbgSession instance that will go to our base class instance and try to invoke a C++ call on a handle that we've phonied up. If we do, it'll definitely crash.

That's why, even though our DbgSession instance is also a Session (which gives us the convenience of not having to redeclare stuff) it isn't a real Session that we can invoke real Session methods on. We can, however, override certain Session methods to suit our own purposes, and, in cases where we want to use the real Session methods, we've also generated a real Session instance on the side to which we can redirect calls. You'll see how we need to override additional Session methods below, as we refine the example.

Debug Class Extensions: DbgAgentContext

to the debug version of AgentContext (see Listing 9.3).

Listing 9.3 Extended AgentContext Class (DbgAgentContext.java)

```
import lotus.notes.*;
public class DbgAgentContext extends
lotus.notes.AgentContext
{

    public DbgAgentContext(Session s) throws NotesException
        {
        super(s, 1);  // use any number here
```

```
        }

    public String getEffectiveUserName()
        {
        return new String("Bob Balaban");
        }
} // end class
```

Nothing too controversial here, really. As with DbgSession, we have to call the base class constructor with a C++ handle (we lie and say "1"). We also have to provide a Session instance, and we're going to use the one passed in by DbgSession in its getAgentContext() call, which is going to be the real Session instance to which we redirect calls.

One interesting thing is that we've overridden the getEffectiveUserName() method to return a hardwired string. This is the way we'll get around not having a real Agent context to work with: We'll fake one for the caller. We'll add some more overrides to AgentContext later.

Further Revisions to Ex90Agent

Now let's recode our Agent's main() function slightly to create a DbgSession instead of a regular old Session and try it again. The only change we need to make is to call "new DbgSession();" instead of "Session.newInstance()". Of course we have to have compiled our new Dbg classes too.

What happens? We get a NullPointerException in the Session's constructor. Something it needs isn't initialized. Again, that's because the Session class wasn't designed to handle this sort of situation; the first ever instantiation of a Session instance in our program is being forced to take an unexpected code path (because we're calling a Session constructor that we're really not supposed to be using). Had we world enough and time before Domino 4.6 shipped, certainly we would have cleaned this up for you.

But never fear, we're nearing the end of most of the bad hackery, and once you get set up to debug Agents this way, you won't have to worry about it again. So onward.

We need to, in our Agent's main() method, initialize a dummy Session object just to get the whole system in gear, then we can get rid of it. So we'll just add a call to Session.newInstance() before the "new DbgSession()" call (the version of Ex90Agent.java on the CD has all these changes in it, don't worry).

Let's recompile and run it again. This time our Agent's NotesMain() method does get a valid (well, sort of valid) AgentContext instance, and we see a message with a user name: so far so good. Then the program crashes because we invoked AgentContext.getCurrentDatabase(). The DbgAgentContext class has no implementation of this method, so Java directed the call to the AgentContext base class, which does have one. That call then got tossed over to Notes with a totally invalid C++ handle ("1"), and Notes croaked. Not to worry, though. All we need to do is continue to hardwire some additional AgentContext methods to return something valid, so the program can keep going. To avoid torturing you by continuing to go step by step with this, let's look ahead and see what other classes we're going to have to create wrappers for.

First we need DbgAgentContext to return some kind of Database instance for the getCurrentDatabase() call. Can we use a real Database instance (easy to get, just pick a database), or will we have to extend the class? In this case we can just use a real Database instance that getCurrentDatabase() can pick. I just used the Database that goes with the original Ex81Agent sample.

The next thing in our NotesMain() function that we need to worry about is the getUnprocessedDocuments() call on AgentContext. Again that's no real problem, we just have to implement a getUnprocessedDocuments() call in DbgAgentContext that uses the current Database to create some kind of collection for us. I decided to use FTSearch() to find some Documents in the Database.

Looking again at NotesMain(), we see that there's a call to Session.createNewsletter(). That's one that we'll have to override in DbgSession, so that the call gets redirected to the real Session object.

From there on out we don't have to do anything special. Listing 9.4 shows all the code for Ex90Agent as modified, plus the full listings of DbgSession and DbgAgentContext (all are also on the CD).

Listing 9.4 Debuggable Agent Plus Debug NOI Classes

```
/*** Class Ex90Agent ***/
import java.lang.*;
import java.io.*;
import lotus.notes.*;

public class Ex90Agent extends AgentBase
{
    private Session my_session;

    public static void main(String argv[])
        {
        try {
            NotesThread.sinitThread();
            Session temp = Session.newInstance();
            temp = new DbgSession();
            Ex90Agent agent = new Ex90Agent(temp);
            agent.NotesMain();
            }
        catch (Exception e) { e.printStackTrace(); }
        finally { NotesThread.stermThread(); }
        }  // end main

    public Ex90Agent() { super(); }
    public Ex90Agent(Session s) { this.my_session = s; }
    public Session getSession()
        {
        if (this.my_session == null)
```

```
            return super.getSession();
        else return this.my_session;
        }

    public void NotesMain()
        {
        try {
            Session s = this.getSession();
            AgentContext ctx = s.getAgentContext();
          System.out.println("User is " +
ctx.getEffectiveUserName());
            Database db = ctx.getCurrentDatabase();
            DocumentCollection dc =
ctx.getUnprocessedDocuments();
            System.out.println("Found " + dc.getCount() + "
selected docs");
            Newsletter nl = s.createNewsletter(dc);
            nl.setSubjectItemName("Subject");
            Document doc = nl.formatMsgWithDoclinks(db);
            doc.send(ctx.getEffectiveUserName());
            System.out.println("Newsletter sent to " +

ctx.getEffectiveUserName());
            }
        catch (Exception e) { e.printStackTrace(); }
        }
}  // end class

/*** Class DbgSession, override of lotus.notes.Session ***/

import lotus.notes.*;
public class DbgSession extends lotus.notes.Session
{
    private Session s;

    public DbgSession() throws NotesException
        {
        super(1);  // any number will do
```

```
        // get a "real" session to redirect calls to
        this.s = Session.newInstance();
        }

    public AgentContext getAgentContext() throws
NotesException
        {
        // return one of our debug versions
        return new DbgAgentContext(this.s);
        }

    public Newsletter createNewsletter(DocumentCollection
dc)
             throws NotesException
         { return this.s.createNewsletter(dc); }

} // end class

/*** Class DbgAgentContext: override of
lotus.notes.AgentContext ***/

import lotus.notes.*;
public class DbgAgentContext extends
lotus.notes.AgentContext
{
    private Session session;
      private Database currentdb;

    public DbgAgentContext(Session s) throws NotesException
        {
        super(s, 1);  // use any number here
       this.session = s;
        }

      public String getEffectiveUserName() throws
NotesException
        { return new String("Bob Balaban"); }

      public Database getCurrentDatabase() throws
NotesException
```

```
            {
            Database db = this.session.getDatabase("",
    "book\\Ex81.nsf");
            this.currentdb = db;
            return db;
            }

         public DocumentCollection getUnprocessedDocuments()
                        throws NotesException
           { return this.currentdb.FTSearch("Balaban", 0); }
    }  // end class
```

So, are we done? In fact, so far as this particular Agent is concerned, we are. You can run this Agent now from the command line, like any application. Any Java development tool that handles Java 1.1.x code will be perfectly happy to let you debug this program. You can also import it into a Notes Database and let it run as a real Agent, though if you leave it unmodified you'd have to import the DbgSession and DbgAgentContext class files also, because they are explicitly referenced.

How does the code know the difference between the two situations? Simple, when you run it from the command line, code starts executing in the Ex90Agent's main() method. Alternatively, when you let it run as a real Agent (either foreground or background), code execution starts in NotesMain(), invoked from AgentBase. In that code path the getSession() call returns the real AgentContext instance set up by AgentBase, because there's no cached DbgAgentSession instance.

If you wanted to save a bit of space in the database, you could—when you're ready to go to production mode with your Agent—comment out the main() method. Then you wouldn't have to import the debug class files at all. Alternatively, you could code the main() function slightly differently. Instead of

```
temp = new DbgSession();
```

you could do it this way:

```
temp = Class.forName("DbgSession").newInstance();
```

This is the only line of code in the Agent class that explicitly references any of the debug classes. If we change it to remove the load time "link" to our debug classes, we can import just the one Agent class into our database. Using Class.forName() to load the DbgSession class and Class.newInstance() to create an instance of it accomplishes exactly that. This way the only time that the debug classes are required is when our main() function is called, which it never would be from Notes. This technique works because the compile time reference to the debug class no longer appears in our Agent's compiled code, and the Java class loader no longer knows at load time that the class is referenced.

What other modifications might you have to make if you re-use these debug classes yourself? Essentially you just have to add more methods to DbgSession and DbgAgentContext as needed: Any Session or AgentContext method that your Agent uses must be overridden in the debug extensions, otherwise your program will crash Domino.

So, how bad a hack is this, really? My own opinion (although I admit to being prejudiced) is that it's in the neighborhood of slightly grungy but not too bad. The business of having to wrapper real objects with debug objects and the strange constructor limitations are slightly repellent, and that's something that could and should definitely be cleaned up in NOI. The idea of extending NOI classes to do debug type activities is perfectly normal in an object-oriented world, and one of the things that Java actually tries to be good at, so no problem there. The NOI classes were explicitly and purposely not declared final to allow just this sort of extension by anybody at all.

Summary

Debugging Java Agents in Domino 4.6 is not as simple an undertaking as we'd like. Still, with a bit of effort, as you've seen in this chapter, it is possible to recast an Agent

program slightly so that it can be debugged using any Java 1.1 compatible development tool.

Chapter 10 goes into still more detail about how objects can be shared across multiple threads in a Java NOI program.