Eclipse Corner Article

How to write an Eclipse debugger

Summary
One of the major tasks of adding a new language to an Eclipse-based IDE is debugging support. A debugger needs to start and stop the program being debugged, suspend and resume, single-step, manage breakpoints and watch points, and so on. This article explains the Eclipse Platform debug framework and steps through a simple, yet illustrative, example of adding debug support for a new language.

By Darin Wright, IBM Rational Software Group
Bjorn Freeman-Benson, Predictable Software
August 27, 2004

Language support

No matter how simple we want to make the topic, there is no getting around the fact that adding a new language to an Eclipse IDE is not a trivial task. This paper, together with its two companion papers, focuses on the launching-running-debugging side of the process. Other aspects of adding new language support include: editors, code assist, refactorings, new views, builders, and other tools. These features are outside the scope of these papers.

The first paper in this series, *We Have Lift-off: The Launching Framework in Eclipse*, describes the launching framework through an applet launcher example. In this paper, we describe the basic debugging framework using a small assembly language example. And in our third paper (not yet published), *Enhancing the Eclipse Debugger*, we describe how to enhance the UI for our assembly language debugger by adding all the little bells and whistles that one expects in modern IDEs.

The Eclipse SDK provides a framework for building and integrating debuggers, collectively known as the debug platform. The debug platform defines a set of Java™ interfaces modeling a set of artifacts and actions common to many debuggers, known as the debug model. For example, some common debug artifacts are threads, stack frames, variables, and breakpoints; and some common actions are suspending, stepping, resuming, and terminating. The platform does not provide an implementation of a debugger – that is the job of language tool developers. However, the platform does provide a basic debugger user interface (that is, the debug perspective) that can be enhanced with features specific to a particular debugger. The base user interface operates against the debug model interfaces, providing views for a program's call stack, variables, breakpoints, watch items, and console I/O, and allows a user to step through source code. The debug platform also provides a framework for launching applications from within the Eclipse IDE, and a framework to perform source lookup.

In order to keep this paper to the stated goal of "how to write a debugger," and away from the larger problem of "how to add a support for a new language," we make the following assumptions:

- The new language has an existing execution engine or interface or VM. This paper does not describe how to write an interpreter, virtual machine, or compiler run-time. For our example, the interpreter is implemented as a Perl program (pdavm/pda.pl in the example code). The language and the debugger interface are described in the next section.
- An Eclipse launcher and launch configuration already exists for the new language. At this point, the launcher needs to create a "run" configuration – this paper will describe how to add a "debug" configuration. (For completeness, here are the parts that comprise the example launcher support:
  - org.eclipse.debug.examples.core contributes an org.eclipse.debug.core.launchConfigurationTypes named org.eclipse.debug.examples.core.launchConfigurationType.pda, as well as the two class implementation of the launch configuration (IPDAConstants and PDALaunchDelegate)
  - org.eclipse.debug.examples.core contributes an org.eclipse.core.variables.valueVariables that points to the Perl executable
  - org.eclipse.debug.examples.ui contributes an org.eclipse.debug.ui.launchConfigurationTabGroups for that launch configuration type, as well as the two-class implementation of the tab groups (PDATabGroup and PDAMainTab).
- The source for the new language is text files that are displayed in a text editor. The text editor might be a specialized editor with code coloring, formatting, extra menu items, etc., or it might be just the standard Eclipse text editor.

Our language and its interpreter

To demonstrate how to write a debugger for Eclipse, we need a language and a run time to debug. For this example, we chose an

enhanced push down automata (PDA) assembly language and a simple interpreter implemented in Perl. Each line contains a single operation and any number of arguments. Our language differs from a standard PDA in two major ways:

- Our language has a control stack and thus has call-return subroutines.
- Our language allows data to be stored either on the data stack or in named variables on the control stack.

In order to actually run this example, you will need a Perl interpreter. Linux®™ comes with Perl. For Microsoft® Windows®, we use either ActivePerl (http://www.activeperl.com/) or Indigo Perl (http://www.indigostar.com/). You also have to set the string substitution variable named "perlExecutable" to the complete path to your Perl interpreter. (For example, ours was C:\perl\bin\perl.exe) To set a string substitution variable, use the Windows > Preferences > Run/Debug > String Substitution preferences page.

Here is an annotated example of the Fibonacci computation (note that the annotations themselves are not valid syntax in this language – in this language, all comments must start at column 1 and be the entire line):

```
push 6
call Fibonacci      function call with one argument on the data stack
output              print result to stdout
halt
#
# f(n) = f(n-1) + f(n-2)
# f(0) = 1
# f(1) = 1
#
:fibonacci
var n               define variable n on control stack
pop $n              get n from data stack
push $n
branch_not_zero gt0
push 1              f(0) = 1
return              return with one value on data stack
:gt0
push $n
dec
branch_not_zero gt1
push 1              f(1) = 1
return              return with one value on data stack
:gt1
push $n             stack: n
dec                 stack: n-1
call fibonacci      stack: f(n-1)
push $n             stack: f(n-1) n
dec                 stack: f(n-1) n-1
dec                 stack: f(n-1) n-2
call Fibonacci      stack: f(n-1) f(n-2)
add                 stack: f(n-1)+f(n-2)
return              return with one value on data stack
```

Interpreter debug interface

Our PDA assembly language interpreter can be started in either run mode or debug mode. When started in debug mode, the interpreter listens for debug commands on a specified local TCP socket and sends debug events to a separate local TCP socket. The commands include:

- `clear N` – clear the breakpoint on line N
- `data` – return the contents of the data stack; the data is returned from oldest to newest as a single string "value|value|value|…|value|"
- `exit` – end the interpreter
- `resume` – resume full speed execution of the program
- `set N` – set a breakpoint on line N
- `stack` – return the contents of the control stack (program counters, function and variable names); the stack is returned from oldest to newest as a single string "frame#frame#frame#…#frame". Each frame is a string "filename|pc|function name|variable name|variable name| …|variable name"
- `step` – single step forward
- `suspend` – end full speed execution and listen for debug commands
- `var N M` – return the contents of a variable M from the control stack frame N (stack frames are indexed from 0).

The debug events that are reported asynchronously to the second socket include:

- `started` – the interpreter has started (guaranteed to be the first event sent)
- `terminated` – the interpreter has terminated (guaranteed to be the last event sent)

- `suspended X` – the interpreter has suspended and entered debug mode; X is the cause of the suspension, either step or client or breakpoint N
- `resumed X` – the interpreter has resumed execution in run mode; X is the cause of the resume, either step or client
- `unimplemented instruction X` – an unimplemented instruction was encountered
- `no such label X` – a branch or call to an unknown label was encountered

Adding debugger support to the launch delegate

> *First Notation Note:* The class and object diagrams in this article use a mostly UML syntax. It is only mostly UML because, unfortunately, UML does not provide a way to model extension points and extensions. Therefore, we decided to use a dashed box to indicate an extension in the plugin.xml file. We also use a few non-standard, but labeled, lines with arrows to indicate certain semantics.
>
> *Second Notation Note:* The Eclipse design guidelines include separating the code for the model from the code for the user interface. We abide by that design principle in our example code resulting in two plug-ins. The extensions and code discussed in this article come from both plug-ins.
>
> *Third Notation Note:* Each code and plugin.xml fragment in this article includes a header describing the plug-in, package, class, and other details of its location in the complete example code. In order to fit all this on one line, the header abbreviates the org.eclipse.debug.examples.pda.core plug-in as "core," the org.eclipse.debug.examples.core.pda.model package as "pda.model," etc.

Using the previous assumptions, we already have a launch configuration type (org.eclipse.debug.examples.core.launchConfigurationType.pda) and a launch configuration delegate (PDALaunchDelegate). To add debugger support, we modify the delegate by adding code for DEBUG_MODE that performs the following actions:

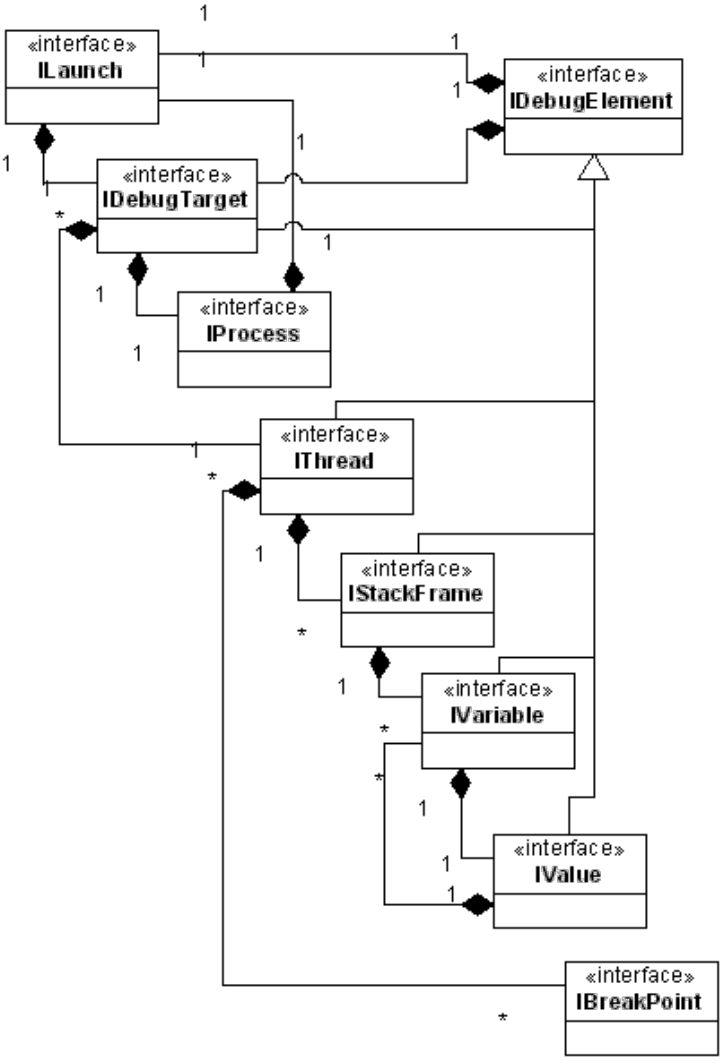
- (1) Modifies the process command line to include the debug parameters. For details on what the debug parameters for our interpreter are, see the previous section.
- (2) Creates and registers the PDADebugTarget object for this launch. For details on where the debug target fits into the model, see the next section.

Plug-in: core, Package: pda.launching, Class: PDALaunchDelegate, Method: launch

```
    commandList.add(file.getLocation().toOSString());
1   int requestPort = -1;
.   int eventPort = -1;
.   if (mode.equals(ILaunchManager.DEBUG_MODE)) {
.      requestPort = findFreePort();
.      eventPort = findFreePort();
.      if (requestPort == -1 || eventPort == -1) {
.         abort("Unable to find free port", null);
.      }
.      commandList.add("-debug");
.      commandList.add("" + requestPort );
.      commandList.add("" + eventPort );
.   }
    String[] commandLine = (String[])
    commandList.toArray(new String[commandList.size()]);
    Process process = DebugPlugin.exec(commandLine, null);
    IProcess p = DebugPlugin.newProcess(launch, process, path);
2   if (mode.equals(ILaunchManager.DEBUG_MODE)) {
.      IDebugTarget target = new PDADebugTarget(launch,p,requestPort,eventPort );
.      launch.addDebugTarget(target);
.   }
```

The debug model

The Eclipse debug model is documented in the *Platform Plug-in Developer Guide* under **Programmer's Guide > Program debug and launch support> Debugging a Program> Platform debug model**. Being more visually oriented, we prefer to look at pictures, so here is a mostly UML diagram of the Eclipse platform debug model:



To implement our debugger (the PDA debugger), we have to provide an implementation of each of these debug model interfaces. Most of the implementations are very straightforward except, perhaps, these small items:

- The PDADebugElement class includes fireEvent, fireCreationEvent, fireSuspendEvent, and other methods as a convenience to avoid code duplication in the subclasses.
- The PDADebugTarget is the object that communicates with our interpreter and is complex enough that we describe it in a following section of its own.
- There is only one thread in a PDA program, so the PDAThread class delegates all its operations (suspend, resume, isSuspended, getStackFrames, and so on) to the debug target. This allows us to encapsulate all communication with the interpreter in our single PDADebugTarget object.
- The only thing the PDAStackFrame does that is mildly interesting is to parse and cache the stack frame reply message from the interpreter (the string described in the previous section). Other than that, the implementation of getLineNumber returns the previously cached line number; the implementation of stepOver delegates to the thread; and so on.
- PDAVariable is just a wrapper for the variable name; it delegates the fetching of the value to the debug target.
- PDAValue is even simpler because the PDA language has only one type: a value. Strings and numbers are stored identically in the interpreter and thus the only thing the PDAValue object does is display the value.
- The PDALineBreakpoint class is the second most complex class in our model (behind the debug target class), but the Eclipse debug framework provides an excellent abstract implementation of line-oriented breakpoints named LineBreakpoint. Thus PDALineBreakpoint subclasses LineBreakpoint and adds its model identifier and its own constructor. The constructor creates and associates a marker with the breakpoint so that they show up in the annotation ruler. (See section below on breakpoints for more details about this code and the marker ID.)

```
public PDALineBreakpoint(IResource resource, int lineNumber) throws CoreException {
    IMarker marker = resource.createMarker(
        "org.eclipse.debug.examples.core.pda.lineBreakpoint.marker");
    setMarker(marker);
    setEnabled(true);
    ensureMarker().setAttribute(IMarker.LINE_NUMBER, lineNumber);
    ensureMarker().setAttribute(IBreakpoint.ID, IPDAConstants.ID_PDA_DEBUG_MODEL);
}
```

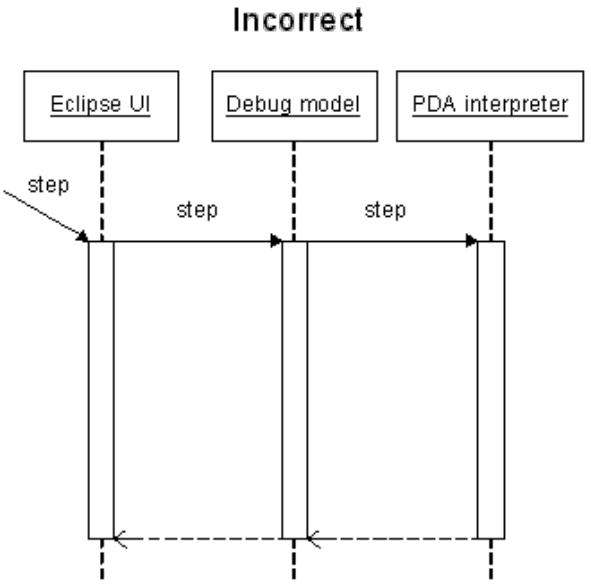
There are two main issues to keep in mind when implementing the debug model classes:

- The Eclipse platform is inherently multi-threaded so it is important that all the classes are thread safe, including those that must

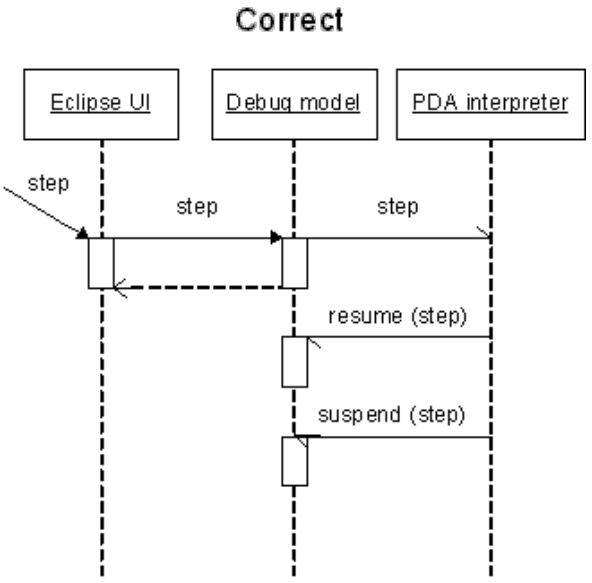
be single threaded. For example, the debug socket channel to our PDA interpreter is single threaded: the interpreter can only receive one command or request at a time. Thus the methods in our model that communicate with the interpreter must be synchronized to prevent overlapping requests.

- The user interface actions, for example the step command, are issued from Eclipse's user interface thread. Thus these actions must be non-blocking. We use step as the example here, because it might seem that step should block, that it should send the step debug command and wait for the interpreter to do the step, but that behavior would be wrong. As we will see below, the interpreter communicates with the debug model by firing events rather than returning result codes from commands.

The incorrect blocking command would block the entire Eclipse user interface until the interpreter had finished stepping. In the case of infinite loops and other client program defects, the worst case is that the interpreter might never return thus leaving the Eclipse user interface completely frozen.

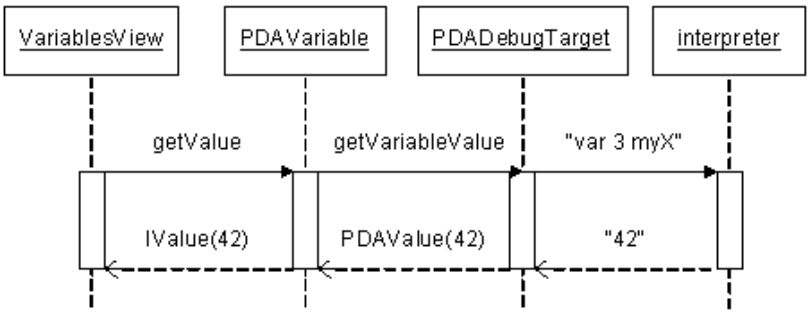


By sending the step command asynchronously, the Eclipse user interface remains responsive while the interpreter performs the command. (For those readers who are not 100% familiar with UML, notice the difference in the arrows: a synchronous call has a full arrow head and an asynchronous message has a half arrow head.)



Code highlights of our debug target

In our debug model, the PDADebugTarget is where most of the action happens because the debug target is where we centralize the communication with the PDA interpreter. For example, when the variable view asks a variable for its value, the variable asks the debug target to ask the interpreter for the value.



Constructor and initialization

As described above, the debug interface to our interpreter consists of two sockets: one for debug commands and one for debug events. Thus when we create our debug target, 1 first we initialize some instance variables (our launch, our process, our one thread, etc), then 2 we open two debug sockets with readers and writers, then 3 we start a thread to listen on the event socket (we are using the Eclipse background process mechanism), and lastly 4 we register to listen for breakpoint changes in order to send the "set" and "clear" commands to the interpreter.

Plug-in: core, Package: pda.model, Class: PDADebugTarget

```
  public PDADebugTarget(ILaunch launch, IProcess process,
     int requestPort, int eventPort) throws CoreException {
     super(null);
1    fLaunch = launch;
.    fTarget = this;
.    fProcess = process;
.    fThread = new PDAThread(this);
.    fThreads = new IThread[] {fThread};
2    try {
.       fRequestSocket = new Socket("localhost", requestPort);
.       fRequestWriter = new PrintWriter(fRequestSocket.getOutputStream());
.       fRequestReader = new BufferedReader(new InputStreamReader(
.                              fRequestSocket.getInputStream()));
.       fEventSocket = new Socket("localhost", eventPort);
.       fEventReader = new BufferedReader(new InputStreamReader(
.                              fEventSocket.getInputStream()));
.    } catch (UnknownHostException e) {
.       abort("Unable to connect to PDA VM", e);
.    } catch (IOException e) {
.       abort("Unable to connect to PDA VM", e);
.    }
3    fEventDispatch = new EventDispatchJob();
.    fEventDispatch.schedule();
4    DebugPlugin.getDefault().getBreakpointManager().addBreakpointListener(this);
  }
```

Communicate with interpreter

Now that communications with the debug sockets on the interpreter are initialized, the debug target methods can send the appropriate commands. For example, step and getVariableValue:

Plug-in: core, Package: pda.model, Class: PDADebugTarget

```
  public void step() throws DebugException {
     sendRequest("step");
  }
  private void sendRequest(String request) throws DebugException {
    synchronized (fRequestSocket) {
      fRequestWriter.println(request);
      fRequestWriter.flush();
      try {
        // wait for "ok"
        String response = fRequestReader.readLine();
      } catch (IOException e) {
        abort("Request failed: " + request, e);
      }
    }
  }

  protected IValue getVariableValue(PDAVariable variable) throws DebugException {
     synchronized (fRequestSocket) {
```

```
            fRequestWriter.println("var "
               + variable.getStackFrame().getIdentifier()
               + " " + variable.getName());
            fRequestWriter.flush();
            try {
               String value = fRequestReader.readLine();
               return new PDAValue(this, value);
            } catch (IOException e) {
               abort(MessageFormat.format("Unable to retrieve value for variable {0}",
                       new String[]{variable.getName()}), e);
            }
         }
         return null;
      }
```

There are many more "communicate with the interpreter" methods. Notice that all communication with the interpreter is serialized using synchronized blocks – as mentioned previously, this is necessary because the Eclipse platform is inherently multi-threaded but our PDA interpreter is not.

Note that our getVariableValue method is synchronous which goes against our earlier caveat to avoid using blocking communication with the target due to the risk of freezing the Eclipse user interface. We chose the synchronous "send the request and wait for reply" rather than an asychronous "send command and later receive the response in an event" for its simplicity in explaining the essential concepts of communication.

Breakpoints are slightly more interesting

The add-a-breakpoint method is a little more interesting because it has to check if 1 the breakpoint is valid and 2 active, but otherwise it has the same “communicate with the interpreter” style. The key to determining if the breakpoint is valid is realizing that breakpoint listeners (like this one) get notified of all breakpoint changes, thus we filter for breakpoints 3 supported by our debug model and for 4 our PDA program:

Plug-in: core, Package: pda.model, Class: PDADebugTarget

```
   public void breakpointAdded(IBreakpoint breakpoint) {
1     if (supportsBreakpoint(breakpoint)) {
         try {
2           if (breakpoint.isEnabled()) {
               synchronized (fRequestSocket) {
                  try {
                     sendRequest("set "
                       + (((ILineBreakpoint)breakpoint).getLineNumber() - 1));
                  } catch (CoreException e) {
                  }
               }
            }
         } catch (CoreException e) {
         }
      }
   }
   public boolean supportsBreakpoint(IBreakpoint breakpoint) {
3     if (breakpoint.getModelIdentifier().equals(IPDAConstants.ID_PDA_DEBUG_MODEL)) {
         try {
            String program = getLaunch().getLaunchConfiguration()
                  .getAttribute(IPDAConstants.ATTR_PDA_PROGRAM,
                        (String)null);
            if (program != null) {
               IMarker marker = breakpoint.getMarker();
               if (marker != null) {
                  IPath p = new Path(program);
4                 return marker.getResource().getFullPath().equals(p);
               }
            }
         } catch (CoreException e) {
         }
      }
      return false;
   }
```

Synchronizing with the interpreter at startup

The other interesting piece of our debug target is the startup code. When the interpreter starts up, it has no breakpoints. Users, having set breakpoints in the code using the Eclipse user interface, expect those breakpoints to work. Thus, after the interpreter starts up, but before it processes any instructions, the debug target has to reach in and set the initial breakpoints. These initial breakpoints are known

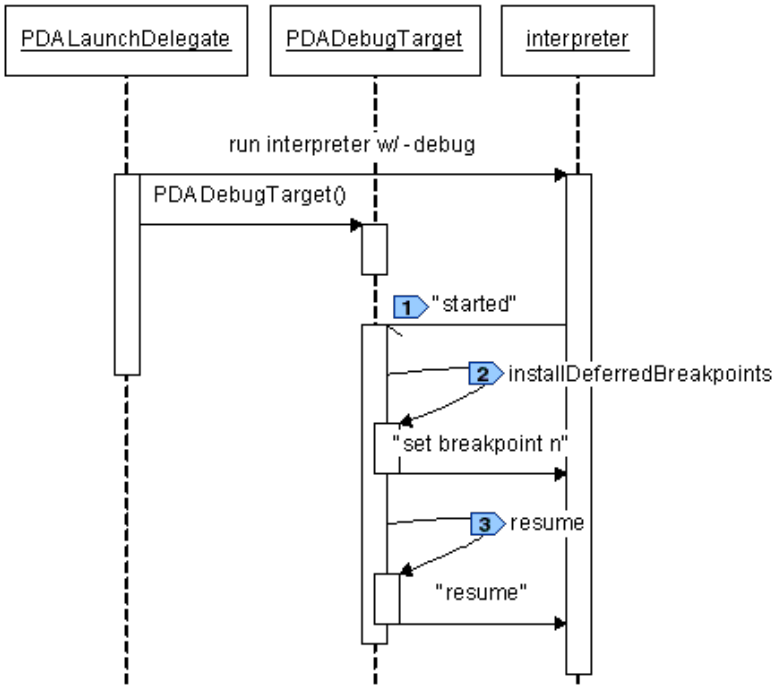
as *deferred breakpoints*.

The standard way this initial setup is accomplished is by having the interpreter suspend on startup, 2 wait for the debug target to set all the breakpoints, and then 3 have the debug target resume the interpreter. (Our apologies to the reader here, but we've gotten a little out of order in describing things. The 1 started method is called by our debug event handler right after the interpreter starts. But at this point in the paper, we have not described events, so you'll just have to trust us on this one.)

Plug-in: core, Package: pda.launching, Class: PDADebugTarget

```
1  private void started() {
       fireCreationEvent();
2      installDeferredBreakpoints();
       try {
3         resume();
       } catch( DebugException x ) {
       }
   }
   private void installDeferredBreakpoints() {
       IBreakpoint[] breakpoints = DebugPlugin.getDefault().getBreakpointManager()
             .getBreakpoints(IPDAConstants.ID_PDA_DEBUG_MODEL);
       for (int i = 0; i < breakpoints.length; i++) {
          breakpointAdded(breakpoints[i]);
       }
   }
```

Here's a sequence diagram showing how the debug target, the interpreter, and the events interact:



Note that because the interpreter events are delivered over a TCP socket to the PDADebugTarget, even if the interpreter starts up and sends the "started" event before the PDATarget is instantiated, the "started" event will be queued in the socket. The potential race condition between the interpreter and debug target is eliminated by having the interpreter suspend on startup.

The debug events

The Eclipse debug model uses debug events (DebugEvent) to describe events that occur as a program is being debugged. Each element in the debug model has a specific set of events that is supports – all of this is documented in the Javadoc of DebugEvent and in the *Platform Plug-in Developer Guide*.

Code highlights of our debug events

As described previously, our example PDA interpreter uses two sockets to communicate with its debugger: a command channel and an event channel. Our debugger uses synchronous RPC over the command channel (send command, block, receive reply), but the events can arrive asynchronously so the debugger must use a separate thread to listen on the event channel.

We create the separate event listener thread when we create the debug target (see previous). The main routine of the thread is a dispatch loop that reads from the event channel and dispatches to the appropriate event behavior:

Plug-in: core, Package: pda.model, Class: PDADebugTarget.EventDispatchJob

```
protected IStatus run(IProgressMonitor monitor) {
   String event = "";
   while (!isTerminated() && event != null) {
      try {
         event = fEventReader.readLine();
         if (event != null) {
            fThread.setBreakpoints(null);
            fThread.setStepping(false);
            …dispatch to the event behavior method…
         }
      } catch (IOException e) {
         terminated();
      }
   }
   return Status.OK_STATUS;
}
```

Our interpreter sends six kinds of events (see previous), but we only handle four kinds with five subtypes. We simply (and safely) ignore the events that we do not yet handle.

```
if (event.equals("started")) {
   started();
} else if (event.equals("terminated")) {
   terminated();
} else if (event.startsWith("resumed")) {
   if (event.endsWith("step")) {
      fThread.setStepping(true);
      resumed(DebugEvent.STEP_OVER);
   } else if (event.endsWith("client")) {
      resumed(DebugEvent.CLIENT_REQUEST);
   }
} else if (event.startsWith("suspended")) {
   if (event.endsWith("client")) {
      suspended(DebugEvent.CLIENT_REQUEST);
   } else if (event.endsWith("step")) {
      suspended(DebugEvent.STEP_END);
   } else if (event.indexOf("breakpoint") >= 0) {
      breakpointHit(event);
   }
}
```

The event handlers are also simple: they turn around and fire the appropriate DebugEvent to all the listeners. As implementers of the debug model, we don't have to know who those listeners are, but curiosity gets the better of us and we want to know. The answer is "the debug views". The debug views listen for events and use those events to update the user interface to show the current state of the debugged program.

In the debug target and the debug element:

Plug-in: core, Package: pda.model, Class: PDADebugTarget

```
private void resumed(int detail) {
   fSuspended = false;
   fThread.fireResumeEvent(detail);
}
```

Plug-in: core, Package: pda.model, Class: PDADebugElement

```
public void fireResumeEvent(int detail) {
   fireEvent(new DebugEvent(this, DebugEvent.RESUME, detail));
}
```

Handling the suspended-due-to-breakpoint event in the debug target is a little more involved because the IThread object keeps track of which breakpoint was hit. The IThread object keeps track of the breakpoint that caused the suspension in case the user interface wants to use that information in labels or icons or some other view. To gather this information, we 1 extract the breakpoint number from the PDA interpreter event message, 2 find the corresponding breakpoint object, and then 3 annotate the current thread object with that breakpoint.

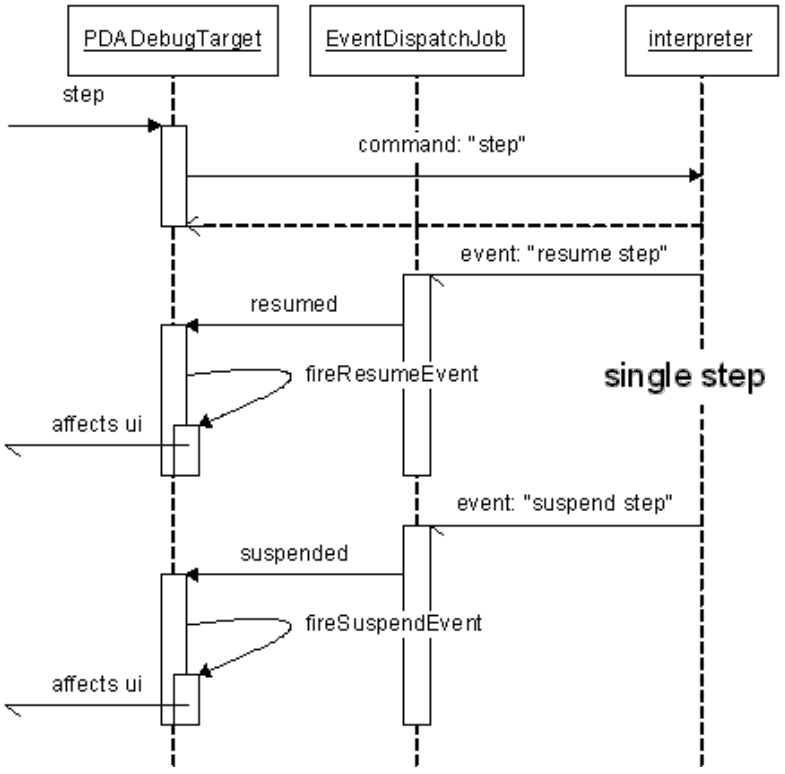
Plug-in: core, Package: pda.model, Class: PDADebugTarget

```
private void breakpointHit(String event) {
   int lastSpace = event.lastIndexOf(' ');
   if (lastSpace > 0) {
      String line = event.substring(lastSpace + 1);
1     int lineNumber = Integer.parseInt(line);
```

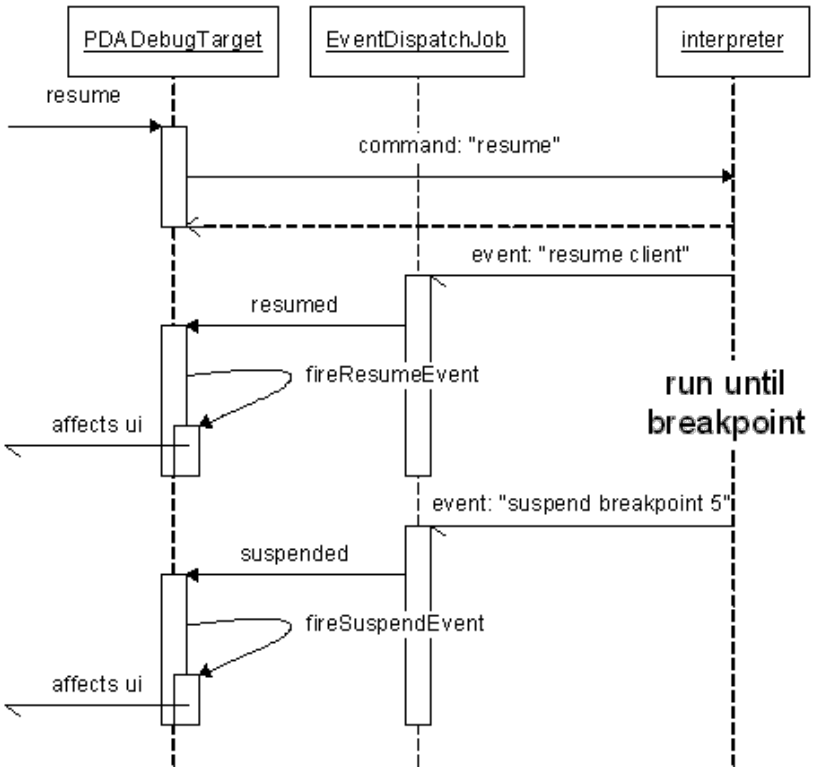
```
2        IBreakpoint[] breakpoints = DebugPlugin.getDefault().getBreakpointManager()
               .getBreakpoints(IPDAConstants.ID_PDA_DEBUG_MODEL);
         for (int i = 0; i < breakpoints.length; i++) {
            IBreakpoint breakpoint = breakpoints[i];
            if (supportsBreakpoint(breakpoint)) {
               if (breakpoint instanceof ILineBreakpoint) {
                  ILineBreakpoint lineBreakpoint = (ILineBreakpoint) breakpoint;
                  try {
                     if (lineBreakpoint.getLineNumber() == lineNumber) {
3                       fThread.setBreakpoints(new IBreakpoint[]{breakpoint});
                        break;
                     }
                  } catch (CoreException e) {
                  }
               }
            }
         }
      }
      suspended(DebugEvent.BREAKPOINT);
   }
```

Pretty pictures of how our debug events work

Here are two sequences showing how the events arrive asynchronously and are processed by the Eclipse debug framework. First, here is how a step action from the user interface gets sent to the interpreter as a “step” command on the command channel. The interpreter operates asynchronously, sending events (“resume” and then “suspend”) back to the debugger by way of the event channel.



Very similarly, here is the sequence for a resume action, followed by the interpreter running for a while and then hitting a breakpoint:



The astute reader will notice that the sequence diagram is exactly the same (except for the slightly different command and different subevents). This regularity to the structure and behavior is a deliberate design decision in the Eclipse debug framework.

Source lookup

Once our debug model and event handler are in place, the Eclipse debugger will work with our interpreter, but its user interface will be quite generic and uninteresting. For example, the generic debugger does not show or highlight source code.

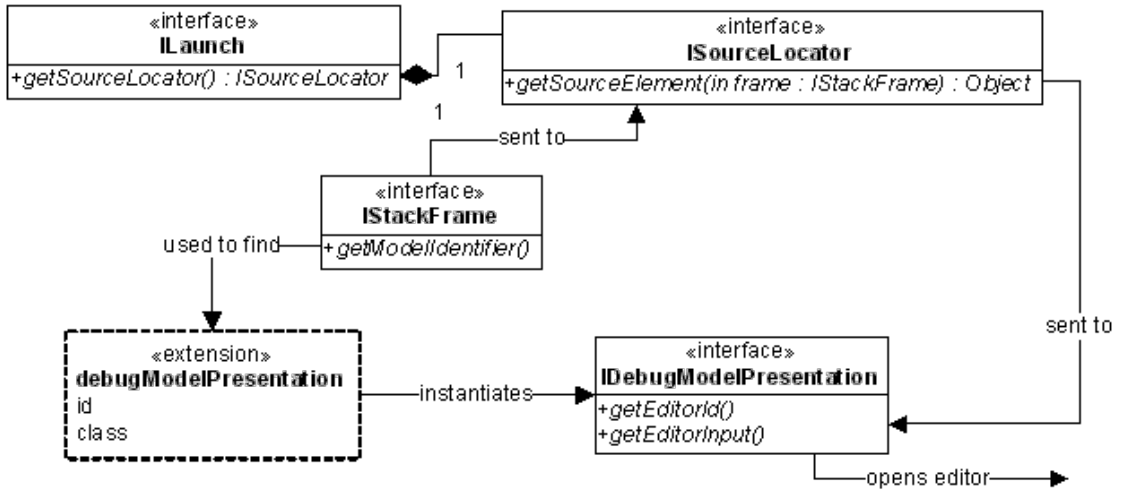
Highlighting the current source code line or statement is de rigueur in modern debuggers, so our next step is to enable what is known as “source code lookup” in our debugger. To do so, we add three new pieces:

1. Our launch object needs a source locator.
2. The source locator object translates stack frames into source elements. In our case, the source elements are IFile objects.
3. Finally, our debug model presentation object maps the source elements (IFiles) to editor ids and inputs. Our example uses the default Eclipse text editor.

To see how this works, let’s start by examining the Eclipse framework for displaying source code of stack frames.

The most general case

In the most general case, the launch object has a source locator object. When the source code for a stack frame needs to be displayed, the stack frame is passed to the source locator method to retrieve the source element. The stack frame’s model identifier is used to find the debug model presentation object (by way of the corresponding extension). The debug model presentation maps the source element to an editor id and editor input, which are then used by the workbench to open an editor to display the source code.



Extension defined source locator

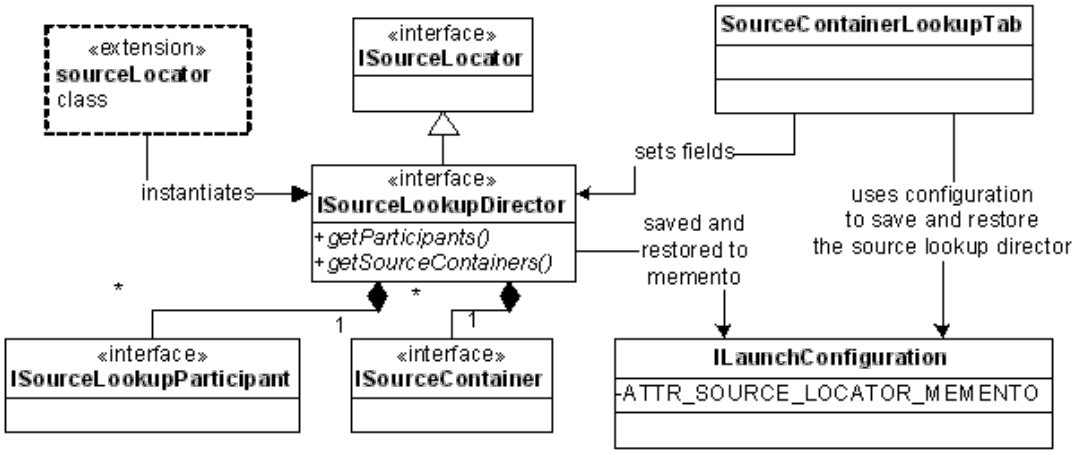
If the launch object does not have a source locator object (that is, a source locator is not assigned by the launch delegate), the debug framework uses the sourceLocatorId attribute of the launchConfigurationType extension to instantiate an ISourceLocator object and store it in the launch object. Thereafter, the rest of the code works as previously described.

Standard kind of source locator

If your source lookup mechanism is standard, that is, files in directories, you can use the Eclipse debug framework's standard source lookup. The ISourceLookupDirector mechanism looks up source files in directories (and zips and jars) along a path (known as the source lookup path). The framework has three pieces:

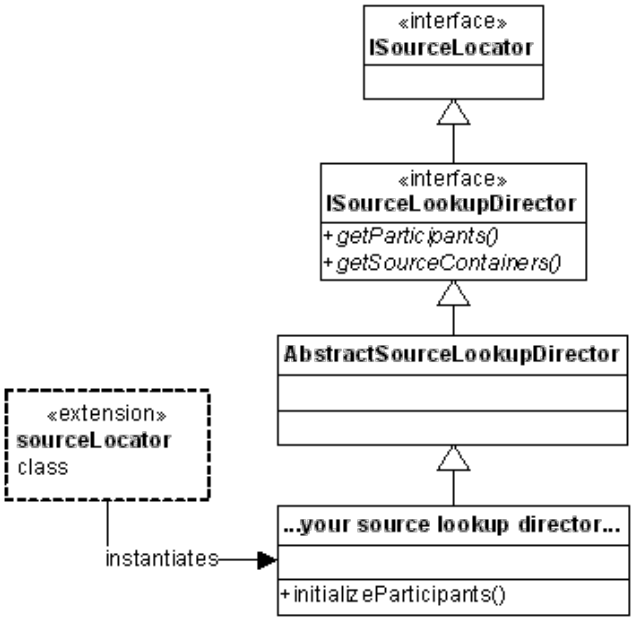
- participants (ISourceLookupParticipant) – objects that map an IStackFrame into a filename
- containers (ISourceContainer) – objects that find files by filename in directories, zips, jars, etc.
- source lookup tab (SourceContainerLookupTab) – an ILaunchConfigurationTab that provides a user interface for configuring and modifying a source lookup path.

The ISourceLookupDirector is an ISourceLocator, so once you have implemented an ISourceLookupDirector, you place the class name in the sourceLocator extension.



Standard implementation of standard kind of source locator

Although ISourceLookupDirector is available as an interface for reimplementation, most debuggers use the default implementation provided by the Eclipse debug framework. The AbstractSourceLookupDirector provides the algorithm for looking source files up along a source path. The only unimplemented method is initializeParticipants, which creates the collection of ISourceLookupParticipants for mapping the stack frames to filenames.

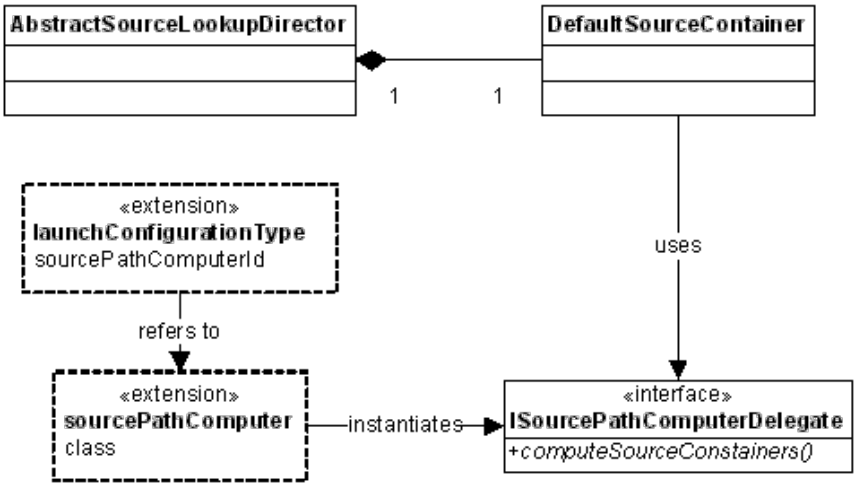


Default source path in the standard implementation

The AbstractSourceLookupDirector has one more trick up its sleeve: if the user has not specified the source path (that is, the source containers) either programmatically or by way of the SourceContainerLookupTab, the AbstractSourceLookupDirector will compute the default source path. Here's how it works:

When initialized, the AbstractSourceLookupDirector has a default source path of a single container: a DefaultSourceContainer. The DefaultSourceContainer uses an ISourcePathComputer (which in turn uses an ISourcePathComputerDelegate) to compute the source path. The source path computer is reused to recompute the default source path until such time as the source containers in the source lookup director are explicitly set to something other than the default.

The source path computer is defined for each launch configuration type using the sourcePathComputer extension. The source path computer delegate has one method: computeSourceContainers.



Code highlights of our simple solution to source code lookup

The simplest solution to source code lookup is to reuse that wonderful framework the Eclipse debug framework provides. Being good developers, we seek to reuse rather than reinvent, so our example uses an AbstractSourceLookupDirector to lookup source files, and a source path computer to compute the default source lookup path.

> *Notation note:* The six steps below are strongly interconnected and thus the code references 1 2 3 4 5 ... are uniform across the six steps. In other words, the 1 in step 1 is the same as the 1 in step 3.

Step 1. source locator

For additional simplicity, we chose to use the "extension defined source locator" described above rather than having our launch delegate create and assign the source locator to the launch object. Thus we start by adding 1 a source locator id to our launch configuration type. Then we added an extension for our source locator id pointing to 2 our source lookup director class.

Plug-in: core, Extension: org.eclipse.debug.core.launchConfigurationTypes

```
    <launchConfigurationType
       …other attributes…
1      sourceLocatorId="org.eclipse.debug.examples.core.sourceLookupDirector.pda"
    </launchConfigurationType>
```

Plug-in: core, Extension: 1 org.eclipse.debug.core.sourceLocators

```
    <sourceLocator
       name="PDA Source Lookup Director"
2      class="org.eclipse.debug.examples.core.pda.launching.PDASourceLookupDirector"
1      id="org.eclipse.debug.examples.core.sourceLookupDirector.pda">
    </sourceLocator>
```

Step 2. source lookup director

The abstract class does most of the work, but our PDASourceLocatorDirector subclass of AbstractSourceLookupDirector still has to initialize 3 the set of participants. In our case, the set of participants is the singleton PDASourceLookupParticipant. Our participant is trivial because in our debug model, 4 the stack frames keep track of their source filenames.

Plug-in: core, Package: pda.launching, Class: 2 PDASourceLocatorDirector

```
    public void initializeParticipants() {
3      addParticipants(new ISourceLookupParticipant[]{new PDASourceLookupParticipant()});
```

```
    }
```

Plug-in: core, Package: pda.launching, Class: 3 PDASourceLocatorParticipant

```
    public String getSourceName(Object object) throws CoreException {
       if (object instanceof PDAStackFrame) {
4         return ((PDAStackFrame)object).getSourceName();
       }
       return null;
    }
```

And that's all it takes to configure, instantiate, and use the framework's source path lookup director.

Step 3. source path computer extension

Next we have to deal with the default source lookup path issue. Again, being efficient, we use the framework provided source path computer extension as explained above. We again 5 add to our launch configuration followed by augmenting the corresponding extension to point to our source path computer class.

Plug-in: core, Extension: org.eclipse.debug.core.launchConfigurationTypes

```
    <launchConfigurationType
       …other attributes…
1      sourceLocatorId="org.eclipse.debug.examples.core.sourceLookupDirector.pda"
5      sourcePathComputerId="org.eclipse.debug.examples.core.sourcePathComputer.pda"
    </launchConfigurationType>
```

Plug-in: core, Extension: 5 org.eclipse.debug.core.sourcePathComputers

```
    <sourcePathComputer
6      class="org.eclipse.debug.examples.core.pda.launching.PDASourcePathComputerDelegate"
5      id="org.eclipse.debug.examples.core.sourcePathComputer.pda">
    </sourcePathComputer>
```

Step 4. source path computer implementation

Our source path computer returns a one-element source path where the one element is the ISourceContainer that contains the source file. This is the longest method we have to write to get source lookup to work.

Plug-in: core, Package: pda.launching, Class: 6 PDASourcePathComputerDelegate

```
    public ISourceContainer[] computeSourceContainers(
          ILaunchConfiguration configuration,
          IProgressMonitor monitor) throws CoreException {
       String path = configuration.getAttribute(
             IPDAConstants.ATTR_PDA_PROGRAM, (String)null);
       ISourceContainer sourceContainer = null;
       if (path != null) {
          IResource resource = ResourcesPlugin.getWorkspace().getRoot()
                .findMember(new Path(path));
          if (resource != null) {
             IContainer container = resource.getParent();
             if (container.getType() == IResource.PROJECT) {
                sourceContainer = new ProjectSourceContainer(
                      (IProject)container, false);
             } else if (container.getType() == IResource.FOLDER) {
                sourceContainer = new FolderSourceContainer(
                      container, false);
             }
          }
       }
       if (sourceContainer == null) {
          sourceContainer = new WorkspaceSourceContainer();
       }
       return new ISourceContainer[]{sourceContainer};
    }
```

Step 5. debug model presentation

As a penultimate step, we add the debugModelPresentation extension that will map our source element (an IFile) to an editor for the workbench to display.

Plug-in: ui, Extension: org.eclipse.debug.ui.debugModelPresentations

```
   <debugModelPresentation
7     class="org.eclipse.debug.examples.ui.pda.launching.PDAModelPresentation"
8     id="org.eclipse.debug.examples.pda">
   </debugModelPresentation>
```

The debug model is linked to the debug model presentation in the usual way that Eclipse user interfaces are linked to Eclipse models: through the extension id. For the debugger, the IDebugElement knows which debug model it is a part of.

Plug-in: core, Package: pda.model, Class: PDADebugElement

```
   public String getModelIdentifier() {
8     return IPDAConstants.ID_PDA_DEBUG_MODEL;
   }
```

The model presentation class has many methods, but for now the only two interesting methods are those that translate source elements into editor ids and inputs.

Plug-in: ui, Package: pda.model, Class: 7 PDAModelPresentation

```
   public IEditorInput getEditorInput(Object element) {
      if (element instanceof IFile)
         return new FileEditorInput((IFile)element);
      if (element instanceof ILineBreakpoint)
         return new FileEditorInput((IFile)((ILineBreakpoint)element).getMarker().getResource());
      return null;
   }
   public String getEditorId(IEditorInput input, Object element) {
      if (element instanceof IFile || element instanceof ILineBreakpoint)
         return "org.eclipse.ui.DefaultTextEditor";
      return null;
   }
```

Step 6. source path tab

The final step is to add the source path tab to our launch configuration dialog box. This is 9 a simple one-line addition to our existing tab group class (you will recall that we had created the tab group class as part of our original launching code back when we read the first article of this series)

Plug-in: ui, Package: pda.launching, Class: PDATabGroup

```
   public void createTabs(ILaunchConfigurationDialog dialog, String mode) {
      setTabs(new ILaunchConfigurationTab[] {
         new PDAMainTab(),
9        new SourceContainerLookupTab(),
         new CommonTab()
      });
   }
```

Breakpoints

Breakpoints are the final major component of a modern debugger that our debugger is still missing. The Eclipse debug framework support for breakpoints is documented in the *Platform Plug-in Developer Guide* under **Programmer's Guide > Program debug and launch support> Debugging a program> Breakpoints**. Our implementation follows that design.

Because our assembly language is line-oriented, our breakpoints are line-oriented, that is, breakpoints are associated with lines and there can be at most one breakpoint per line. Other more complex languages have statement or expression-oriented breakpoints and multiple statements or expressions on a single line. We have left discussion of those to the third paper of this series *Enhancing the Eclipse Debugger*.

> *Notation note:* We use the same "code references are uniform across the three steps" in this section that we used in the previous section.

Step 1. breakpoint objects

The first step to implementing our line-oriented breakpoints is to define our breakpoint data structure using the breakpoint and the resource marker. As described in the Platform Plug-in Developer Guide, the resource marker is the mechanism by which the breakpoint is persisted between Eclipse sessions. We conveniently reuse code by 1 creating the marker for our line-oriented breakpoints as a subtype of the Eclipse framework line-oriented breakpoint marker. Our marker inherits all the attributes of the Eclipse marker and we do not need, thus we do not define, any additional attributes.

Plug-in: core, Extension: org.eclipse.debug.core.breakpoints

```
   <breakpoint
1     markerType="org.eclipse.debug.examples.core.pda.lineBreakpoint.marker"
```

```
2     class="org.eclipse.debug.examples.core.pda.model.PDALineBreakpoint"
      id="org.eclipse.debug.examples.core.pda.lineBreakpoint">
   </breakpoint>
```

Plug-in: core

```
   <extension
1     id="pda.lineBreakpoint.marker"
      point="org.eclipse.core.resources.markers">
      <super type="org.eclipse.debug.core.lineBreakpointMarker"/>
      <persistent value="true"/>
   </extension>
```

Note that the org.eclipse.core.resources.markers extension id is "pda.lineBreakpoint.marker" rather than "org.eclipse.debug.examples.core.pda.lineBreakpoint.marker" because the id is automatically prefixed with plug-in id, i.e., "org.eclipse.debug.examples.core".

Step 2. toggle breakpoint menu item

The toggle breakpoint menu item in the Eclipse debug perspective uses a re-targetable action provided by the debug platform to toggle the breakpoints using the appropriate debug model. The retargetable action asks the active part (editor, view, and so on), for its toggle breakpoint adapter. When available, the adapter is used as a delegate to toggle breakpoints. We use plugin.xml to register the factory for the particular editor we are using, then we define 3 the factory and 4 the adapter. The adapter has to be an IToggleBreakpointsTarget, but other than that, this is standard Eclipse platform code.

Plug-in: ui, Extension: org.eclipse.core.runtime.adapters

```
   <factory
3     class="org.eclipse.debug.examples.ui.pda.model.PDABreakpointAdapterFactory"
      adaptableType="org.eclipse.ui.texteditor.ITextEditor">
      <adapter type="org.eclipse.debug.ui.actions.IToggleBreakpointsTarget"/>
   </factory>
```

Plug-in: ui, Package: pda.model, Class: 3 PDABreakpointAdapterFactory

```
   public Object getAdapter(Object adaptableObject, Class adapterType) {
      if (adaptableObject instanceof ITextEditor) {
         ITextEditor editorPart = (ITextEditor) adaptableObject;
         IResource resource = (IResource) editorPart.getEditorInput().getAdapter(IResource.class);
         if (resource != null) {
            String extension = resource.getFileExtension();
            if (extension != null && extension.equals("pda")) {
4              return new PDALineBreakpointAdapter();
            }
         }
      }
      return null;
   }
```

Note that we use the plugin.xml to register the adapter rather than programmatically registering the adapter. Using plugin.xml allows our adapter (and thus our toggle breakpoints) to be active before our lazily loaded plug-in is active.

Also note that this isn't a completely elegant solution because we are registering an adapter factory for the text editor. Which means that if other plug-ins register a toggle breakpoints adapter for the text editor, one of the plug-ins will work and the others will not, but there is no way to know which one will be "the winner". What we, as language IDE developers, should really do is supply our own source code editor and then adapt that editor rather than the generic text editor. However, source code editing is out of the scope of this article and thus we use the simple text editor for demonstrative purposes.

Step 3. toggle breakpoint action

Toggling a breakpoint is as expected: look for an existing breakpoint on the line; 5 if there is a breakpoint, remove it; 6 if there is not a breakpoint, create one (of the class we 2 registered in plugin.xml).

Plug-in: ui, Package: pda.model, Class: 4 PDALineBreakpointAdapter

```
   public void toggleLineBreakpoints(IWorkbenchPart part, ISelection selection)
         throws CoreException {
      ITextEditor textEditor = getEditor(part);
      if (textEditor != null) {
         IResource resource = (IResource) textEditor.getEditorInput()
                                          .getAdapter(IResource.class);
         ITextSelection textSelection = (ITextSelection) selection;
         int lineNumber = textSelection.getStartLine();
         IBreakpoint[] breakpoints = DebugPlugin.getDefault().getBreakpointManager()
```

```
                    .getBreakpoints(IPDAConstants.ID_PDA_DEBUG_MODEL);
            for (int i = 0; i < breakpoints.length; i++) {
               IBreakpoint breakpoint = breakpoints[i];
               if (resource.equals(breakpoint.getMarker().getResource())) {
                  if (((ILineBreakpoint)breakpoint).getLineNumber() == (lineNumber + 1)) {
5                    breakpoint.delete();
                     return;
                  }
               }
            }
2 6         PDALineBreakpoint lineBreakpoint = new PDALineBreakpoint(resource, lineNumber + 1);
            DebugPlugin.getDefault().getBreakpointManager().addBreakpoint(lineBreakpoint);
         }
      }
```

The only mildly interesting twist in our code is that our breakpoints store the user view of line numbers (the first line of a file is line 1) rather than the implementation view of line numbers (the first line of the file is line 0). But you, our dear reader, have vast experience dealing with off-by-one errors and have already figured this out.

The generic Eclipse text editor does not support double clicking in the ruler to create breakpoints the way the more specialized source code editors do, but again that is a feature of the way we chose to implement (or not implement) our source code editor rather than a feature of the Eclipse debug framework's support for breakpoints.

Summary and further information

The Eclipse platform provides a very powerful framework that enables developers to add support for a new language quickly and easily. Having read this far, you now realize that adding debug support for a new language is a relatively simple task. There are a small number of classes to implement (eighteen in our example), and an even smaller number of extensions to specify (ten in our example). For more details on any aspect of the framework, we refer you to the API documentation of the Debug Platform and, for a very complete example, of the Java Debugger.

Our journey, however, is not ended yet. The launch and debug facilities we have described and implemented in these first two papers are useful, but like a light read, will ultimately not be satisfying to most language implementers. We have described launching, debug target control (step/resume/suspend), basic debug element display, source lookup, and breakpoints. Most language implementers will also want watch points, more descriptive labels, informative icons, custom views and language specific actions. For details on these enhancements to the debug support, we refer you to the third paper of this trilogy, *Enhancing the Eclipse Debugger*, which is still under construction.

Acknowledgements

Thanks to Darin Swanson and Cameron Stevens for their careful reviews and critiques that have improved this paper.

Source code

try it! To examine the entire source code, or to run the example, download the zip file and unzip it into your Eclipse directory. Both plug-ins come with complete source code, although you will need your own installation of Perl to run the PDA interpreter. This article describes, and the included code requires, features that are available in Eclipse 3.0 or a later version.

Trademarks