Getting Started with the Open Dylan IDE

Release 1.0

Dylan Hackers

December 15, 2018

Contents:

CHAPTER

ONE

COPYRIGHT

CHAPTER

TWO

PREFACE

This guide explains how to use the Open Dylan IDE (Windows only) to develop and deliver Dylan applications.

For help getting started with the command-line tools, see the Getting Started with the Open Dylan Command Line Tools guide.

The Dylan Reference Manual

The Dylan Reference Manual (Shalit, 1996) is published by Addison Wesley Developers' Press, ISBN 0-201-44211-6. In this guide, we refer to the *Dylan Reference Manual* as "the DRM".

Chapters in this guide

The early chapters on the IDE give you a quick tour of the application development cycle under Open Dylan, using the Reversi example for illustration. In these early chapters we come across most of the Open Dylan development tools. Later chapters examine those tools more directly, and provide a broader view of the development process in Open Dylan.

Quick Start, shows how to build a standalone executable application in Open Dylan. It discusses the Open Dylan main window and the project window.

Fixing Bugs, shows how to make changes to your application sources and how to debug compile-time and run-time errors. It discusses the project window, the debugger, the editor, and the browser.

Programming in Open Dylan, gives an overview of the programming model in Open Dylan.

Creating and Using Projects, discusses the New Project wizard, and looks at the project window in more detail.

Learning More About an Application, shows how you can examine the sources of a project, and run-time values in an application, using the browser.

Debugging and Interactive Development, returns to the debugger and studies it in more detail.

Remote Debugging, describes how to debug a Dylan application running on another machine.

Dispatch Optimization Coloring in the Editor, describes the editor's facility for coloring source code to show the degree to which it has been optimized.

Delivering Dylan Applications, shows how you can package an application with everything necessary to deliver it to customers as a stand-alone product.

CHAPTER

THREE

QUICK START

This first chapter is a quick introduction to building applications with Open Dylan.

Starting work with Open Dylan

In this section, you will start Open Dylan's development environment, and choose one of several supplied examples to work with.

1. Start the Open Dylan environment.

   You can do this by choosing the Start menu shortcut **Start > Programs > Open Dylan > Open Dylan**. Open Dylan will also start if you open any file associated with it, such as a Dylan source file.

   When Open Dylan starts, the *main window* appears:

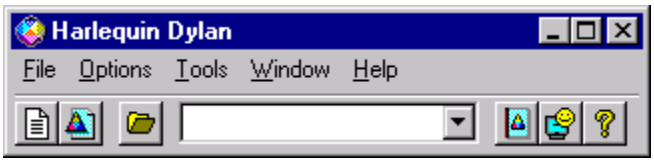

Fig. 3.1: The Open Dylan main window

   The *initial dialog* also appears:

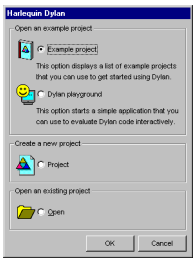

Fig. 3.2: The initial dialog

   The main window will be present throughout your Open Dylan session. It provides a way to set environment-wide options and to control the display of all Dylan windows. To exit Open Dylan, choose **File > Exit** in the main window.

The initial dialog is there to help you get working quickly, whether by looking at an example project, creating a new project, opening an existing project or text file, or starting an interactive Dylan session in the *Dylan playground*. (See *Interaction basics using the Dylan playground* for more information.)

We want to browse the example projects.

2. Select the "Example project" option and click **OK**.

   The Open Example Project dialog appears.

   The Open Example Project dialog shows several categories of example Dylan project. If you expand a category you can see the examples offered.

   The "Getting Started" category contains two very simple projects to help you start programming in Dylan and using the libraries included with Open Dylan. The "Documentation" category contains examples from the Open Dylan documentation set.

   The source files for the example projects are stored under the top-level Open Dylan installation folder, in the subfolder *Examples*. Each example has its own project folder. The files that Open Dylan creates when building a project are also stored under this folder.

   We are going to look at the example called Reversi, which is in the "Documentation" category.

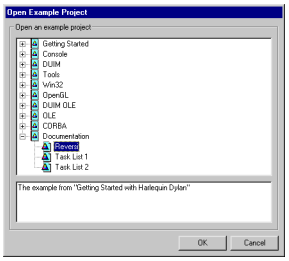

Fig. 3.3: The Open Dylan Examples dialog

3. Expand the "Documentation" category by clicking the + next to it.
4. Select "Reversi", then click **OK**.

   A *project window* appears.

   The project window is one of four programming tools in Open Dylan. The other tools are the browser, the editor, and the debugger.

The project window

In the project window you can see the project that you are working on. A *project* represents a Dylan library under development. We look at projects in more detail in Creating and Using Projects.

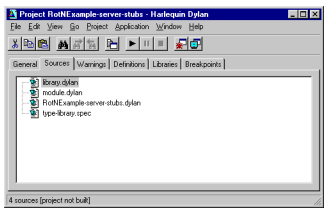

Fig. 3.4: The Open Dylan project window.

Projects are a key concept in Open Dylan. Virtually all programming tasks you can carry out in Open Dylan take place in the context of a project. (An exception is editing text files: you can edit a text file without its being part of a project.)

Every time you open a project, or create a new one, Open Dylan displays it in a project window. If you have more than one project open at a time, each is displayed in a separate window.

You will use the project window frequently during development. Some of the things you can do in or begin from the project window are:

- Building, running, and interacting with applications.
- Editing source files.
- Browsing warnings generated when a project was last built.
- Browsing the classes, functions, variables, and so on that a project defines.
- Browsing the libraries used by a project.
- Browsing and setting breakpoints on source code.

In short, the project window is a focal point for work in Open Dylan.

The Reversi project

The project window that is now on your screen has several pages of information about the project Reversi. This project represents a Dylan library called Reversi that can be built into an executable application. Reversi implements a popular board game.

The default view in the project window is the Sources page, which displays the source files that the project contains. We can see that the Reversi project contains six source files. These files include definition files for the Reversi library and the modules it exports—*library.dylan* and *module.dylan* respectively. Later, we will add some more files to the project to implement new game features.

Building an executable application

We can use the project window to build an executable application from the Reversi project.

This will be a normal Windows executable file with an EXE extension. Open Dylan puts the files it builds in a subfolder of your top-level Open Dylan installation folder, devoted to the Reversi project. We will see more about where projects and their build products reside on disk in *Projects on disk*.

Building Reversi

To build an executable application for a project, choose **Project > Build** in its project window.

Choose **Project > Build** in the Reversi project window.

Open Dylan starts building the application. A progress indicator window appears.

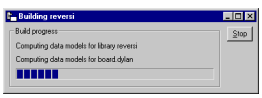

Fig. 3.5: The build progress indicator.

Because we have never built the Reversi application before, Open Dylan examines and compiles every source file. When compilation is finished, it links the compiled files together with the system libraries that the application uses, and creates an executable file. Before it does that, however, a dialog appears.

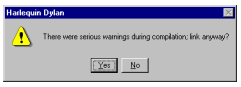

Fig. 3.6: The link warning dialog.

The dialog appears to let us know that the compiler issued *serious warnings* while compiling the project sources, and to let us choose whether to go ahead with the "link" phase of building. The "link" phase creates the executable application for the project.

Why are there serious warnings? Recall from *The Reversi project* that we will be adding some more source files to the project later on. These files implement new game features that require debugging and will help to demonstrate the Open Dylan development environment further. Because we built the project without adding the extra files, the compiler detected that some code in the project refers to name bindings that were otherwise undefined, and issued the serious warnings.

If there are serious warnings when compiling a project, we will usually want to fix the code first before trying to run the application. But sometimes it is useful to be able to execute an application that is only partly finished. As long as we know that the code containing the references to the undefined bindings is not going to be executed, we can safely test the rest of the application.

Reversi has been carefully coded to avoid calling these undefined names until the files containing their definitions are included in the project and the project is rebuilt. So there is no harm in building an executable for Reversi.

If we click **Yes** in the link warning dialog, an executable is created; if we click **No**, building stops. In either case, the serious warnings are recorded in the project window's Warnings page.

Click **Yes** in the link warning dialog.

Open Dylan links an executable for Reversi.

Running Reversi

Now that the application is built, we can run it. The project window menu command **Application > Start** runs the most recently built executable for that window's project.

Choose **Application > Start** and the Reversi application window appears.

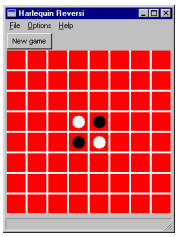

Fig. 3.7: The Reversi application.

The Reversi application is now up and running.

When you choose **Application > Start**, the executable starts, runs, and terminates normally, but at the same time it has an invisible connection to the Open Dylan debugger. This means you can pause execution at any time to debug the application and even interact with it. (Use **Application > Pause** to do this.) In addition, if there is an unhandled error

inside the application, the debugger will catch it automatically so that you can examine it. We will learn more about the debugger later, in *Problems at run time* and also in Debugging and Interactive Development.

Three ways of running Dylan applications

There are three ways we can run the Reversi application we have just built. The first is to choose **Application > Start** from the Reversi project window, as we have just seen. This menu command is also available in the Open Dylan debugger and editor.

The second way to run the application is to click the "Start/Resume" button (▶) on the project window's toolbar. Again, we can do this in the Open Dylan debugger and editor too. And again like **Application > Start**, running an application this way connects it to the debugger, so that any unhandled errors are caught and we can pause and interact with the application.

Finally, we can run the application from outside Open Dylan as we would any other executable application file—such as by typing its file name into an MS-DOS console window, or double-clicking on it in an Explorer window. If we run an application this way, Open Dylan cannot connect a debugger to it. Any unhandled errors could therefore crash the application, which we would not be able to pause and interact with in the paused context.

Looking at definitions and expressions

Each file listed on the Sources page of the Reversi project window is now preceded by a + symbol. This means we can now expand a file name to reveal a list of all the Dylan definitions and top-level expressions in that file.

1. In the Reversi project window, select the Sources page.
2. Expand the *algorithms.dylan* item by clicking the **+** next to it.

The definitions and expressions are sorted alphabetically under the file name in which they appear. Definitions are the Dylan name bindings created by compiling top-level definitions in the source code that the file contains. Thus `define class` causes a class name to appear in the list of definitions, `define method` a method name, and so on, but the names of accessors on a class's slots do not appear.

The expressions in the list are, roughly speaking, anything that appears at top level in the source file but is not a definition. Typically such expressions are assignments to global variables or function calls. For example, in the listing for *algorithms.dylan*, there are several top-level calls to the method `install-algorithm`, which stores values in a globally visible table. Expressions that are part of a larger expression, or part of a definition, are not shown.

In addition to the normal Dylan syntactic conventions—a leading dollar sign for a constant, enclosure in angle brackets for a class, and so on—icons appear next to definition names to indicate the kind of Dylan object to which the names are bound. Constants, for example, are indicated by an octagonal icon containing a stylized dollar sign (). Expressions are indicated by a green octagon ().

You can also see definitions (but not expressions) listed by library and module on the Definitions page. This page includes a facility for filtering definitions out of the visible list according to their type or whether their name contains a given string.

These lists are just one use to which the Open Dylan environment puts its *compiler database*, a rich collection of information that the compiler derives from every project it builds.

The fact that the compiler database is derived during compilation explains why the file names in the Sources page were not expandable when we first opened the Reversi project, and also why we would have seen that the Definitions page was empty at that time. The compiler database for Reversi did not exist until after we built the Reversi application. However, when we open the Reversi project in future sessions, Open Dylan will read in the database from disk.

Compiler databases are mostly used by the *browser* tool, which we will look at later. See *Compiler databases* for more details of the compiler database and Learning More About an Application for details of the browser.

Building DLLs

By default, Open Dylan projects are built into executable applications (.EXE files), but with a simple setting we make them be built into dynamic-link libraries (.DLL files).

This option is just one that we can change in the project window's **Project > Settings** dialog. From that dialog we can also set compiler optimizations, project version information, and command-line arguments for console applications. For more details, see *Project settings*.

Making changes to an application

We will now make a change to the Reversi application. We are going to add a new feature that allows someone playing Reversi to change the shape of the pieces.

If you look at the Reversi application again now, you will see that some of the commands on the *Options* menu—*Circles*, *Squares* and *Triangles* —are unavailable. Our changes will enable these items.

Among the Reversi example files, there is a prepared Dylan source file with the changes we need for this new piece-shapes code. It is not yet a part of the project, so to incorporate it into our Reversi application, we must add it to the project.

1. Exit Reversi by selecting **File > Exit** from the Reversi application window.

   You can also tell the environment to terminate a running application using **Application > Stop** or the project window's stop button (▣). When you ask to terminate an application in this way, the environment asks you for confirmation, to prevent application state being lost by accident.

2. In the Reversi project window, select the Sources page.

   The positions of files in the sources list are important. The last file in the list should always be the file that contains the code that starts the application running. Unlike C or Java, Dylan does not require us to write a function of a predetermined name in order to start an application. We simply make the last piece of code in the last source file an expression that does something with all the Dylan definitions that the source files contain.

   In the Reversi project, *start-reversi.dylan* contains the code that starts the application and so must be at the bottom of the source file list. We want the file we are going to add to appear between *board.dylan* and *start-reversi.dylan*.

3. Select *board.dylan*.

4. Choose **Project > Insert File**.

   The Insert File into Project dialog appears.

5. In the Insert File into Project dialog, select *piece-shapes.dylan* and click **Open**.

   Open Dylan adds *piece-shapes.dylan* below *board.dylan*.

   Now that *piece-shapes.dylan* is part of the sources that will be used to build the Reversi application, we can rebuild the executable.

6. Choose **Project > Build** in the Reversi project window.

   Open Dylan builds the application again.

This time, Open Dylan compiles only one file: *piece-shapes.dylan*. No changes had been made to the existing source files, so it did not need to recompile them. It simply linked the existing compiled files with the new one to make the new executable.

This incremental compilation feature can save a lot of time during development, when you want to rebuild your application after a small change in order to test its effects. Open Dylan automatically works out which files it needs to recompile and which it does not. The compiler also updates a project's database during incremental compilation.

When compilation of *piece-shapes.dylan* is complete, there are still some serious warnings. The link warning dialog appears to ask you to confirm that you want to link an executable for Reversi.

7. Click **Yes** in the link warning dialog.

   We can now run the new version of Reversi.

8. Choose **Application > Start** in the Reversi project window.

   A new Reversi application window appears.

9. In the Reversi application, select the *Options* menu.

   Thanks to our compiling the changes to the project, the *Circles*, *Squares*, and *Triangles* items are now available:

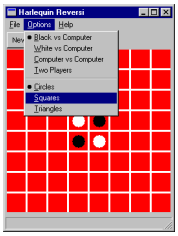

Fig. 3.8: The Reversi application's *Options* menu after the code changes.

10. Choose **Squares**.

    The Reversi application updates the board, laying the pieces out again as squares rather than circles.

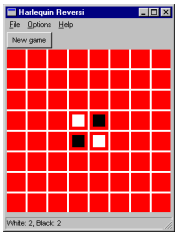

Fig. 3.9: The Reversi application with square pieces.

CHAPTER

FOUR

FIXING BUGS

Though we can play Reversi, we cannot save games in progress and resume play at a later date. In this section, we add a save feature to the game, available through **Save**, **Save As** and **Open** commands on the **File** menu. (If you look at the version of Reversi we have at the moment, you will see that those items are disabled.)

The code we will add to implement the save feature has a small bug that we must correct before the feature will work properly. In doing so, we will see the editor, debugger, and browser tools.

Rebuilding the application

There is another prepared Dylan source file with the changes we need to implement the game-saving facility.

In the Reversi project window, go to the Sources page and select *piece-shapes.dylan*.

Choose **Project > Insert File**.

The Insert File into Project dialog appears.

In the dialog, select *saving.dylan* and click **Open**.

Open Dylan puts *saving.dylan* below *piece-shapes.dylan* in the list of project sources.

Now we can rebuild the executable Reversi application.

Choose **Project > Build** in the Reversi project window.

Open Dylan builds the application again.

If you are still running Reversi when you choose **Project > Build**, you will be asked to confirm that you want to stop running it and go ahead with the build.

Towards the end of the build, the link warning dialog appears again, asking whether we want to link the application executable despite there being serious warnings. Again, we do want to link the application.

Click **Yes** in the link warning dialog.

Notice the message in the status bar at the bottom of the Reversi project window after rebuilding Reversi:

```
Build completed. 5 serious warnings, 1 warning.
```

In the next section, we look at what serious warnings and warnings are, and how the environment helps us deal with them.

Problems at compile time

In this section we look at how the compiler handles problems that it comes across in a project's source code. When we rebuilt Reversi in *Rebuilding the application*, the compiler discovered nine problems in the project's source code. It reported those problems and divided them into two categories: serious warnings and warnings.

The compiler issues a *serious warning* for code that would lead to an exception at run time if it was executed. Among the causes of serious warnings are Dylan syntax errors, references to undefined bindings, and calls to functions with the wrong number or type of arguments.

For code with only cosmetic problems, such as a method definition that ends with `end class` instead of `end method`, the compiler issues a *warning*.

You can see a table of any warnings or serious warnings that were generated the last time a project was compiled by choosing the Warnings page in the project window. After a build where there were warnings or serious warnings, like this one, the Warnings page is selected automatically.

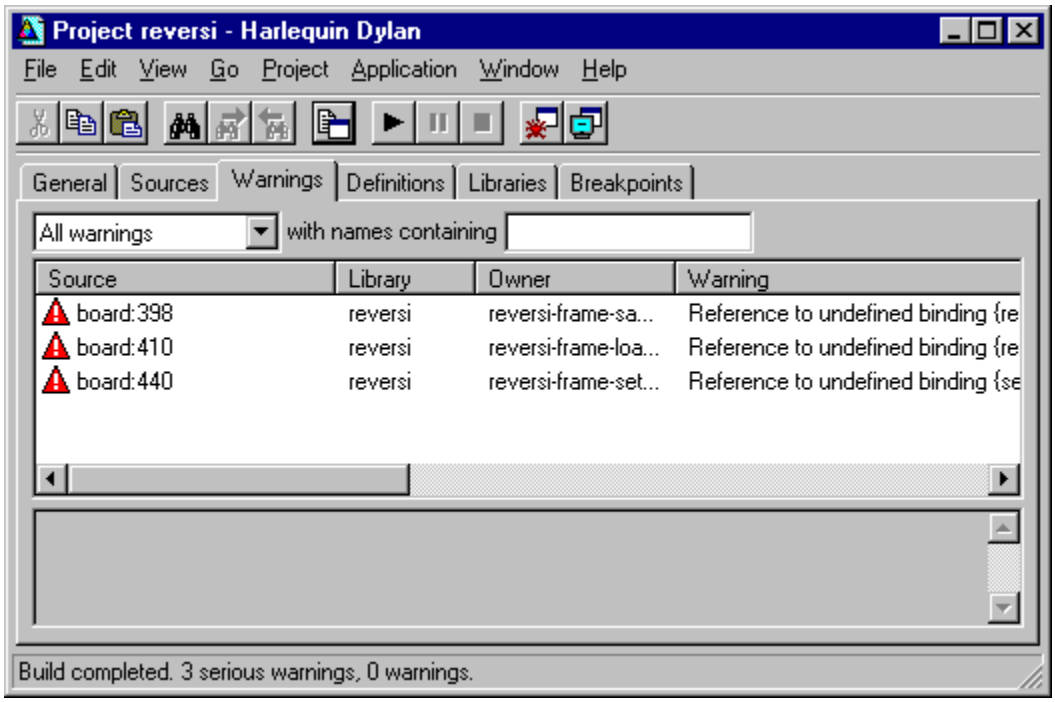

Fig. 4.1: The project window's Warnings page.

The table has four columns:

- Source Shows the source code location of the problem that caused the warning or serious warning. The column has the form {*file name* }:{*line number* }.
- Library Shows the library containing the source code that caused the warning.
- Owner Shows the problem area that caused the warning.
- Warning Shows the warning or serious warning message.

The items in the table are sorted lexically by the Source column value. Warnings and serious warnings without associated source locations have empty Source fields, and appear at the top of the list. (To sort the table by the values in another column, click on that column's header.)

Each warning or serious warning is linked to the source definition that caused it. We can select individual warnings and serious warnings to learn more about the problems they are describing.

Select the first item in the Source column, a serious warning in *saving.dylan* at line 20.

We can now see the full text of the serious warning:

```
Serious warning at saving:20:
Unexpected token "define".
```

If we double-click on an item, Open Dylan opens the appropriate source file in an editor window, and positions the insertion point at the line containing the problem. If the problem spans a range of lines, the entire range is highlighted in the editor. Likewise, if the problem is an undefined binding, the binding in question is highlighted.

Double-click on the same item.

An editor window appears.

Notice how the editor separates each definition in a source file with a gray line. Printed in the middle of each line is the name of the definition below it. These code separators also appear above top-level expressions wrapped with *begin* ... *end*.

The code separators are just a visual aid, and are not part of the file itself. If you opened the source file in a different editor you would not see the separators. They are ignored by the compiler.

When you add a new definition, or a new *begin* ... *end* top-level form, the code separators will only be updated if you manually refresh the editor window (**View > Refresh**), move the cursor past an existing separator, or perform some other operation that forces the editor to redisplay.

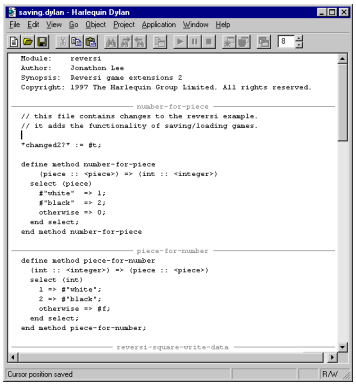

Fig. 4.2: The Open Dylan editor.

Now we can see the cause of the first serious warning. A semi-colon is missing from the end of the definition of *number-for-piece* in *saving.dylan*. The missing semi-colon makes the definitions of *number-for-piece* and *piece-for-number* run into one another instead of being separate.

As it turns out, all the other serious warnings that were reported were caused by this single missing semi-colon. The compiler could not parse the definitions of *number-for-piece* and *piece-for-number*. The compiler skips over such source code and does not generate object code for it.

This means that any subsequent references to *number-for-piece* and *piece-for-number* in the source code would be references to name bindings that are never defined in the compiled application. Lines 32, 50, 59, and 73 referred to one or the other of these names, which triggered the serious warnings.

Add the missing semi-colon so that the last line of the definition *number-for-piece* appears as follows:

```
end method number-for-piece;
```

While we are editing the file, we can fix the non-serious warning. It is caused by a mismatched `end` clause in *reversi-game-write-data*. It is a method, but the `end` clause says `end class` instead of `end method`.

Locate the definition of the *reversi-game-write-data* method in *saving.dylan*.

Change the last line of the definition so that it appears as follows:

```
end method reversi-game-write-data;
```

Choose **File > Save** in the editor.

Open Dylan saves the file, first making a backup of the previous version in *saving.dylan~* —that is, in a file of the same name, but with an extra character in the file extension, a tilde (~), to show that it is a backup file.

Having attended to the cause of the serious warnings and warnings, we can rebuild the application and try out the new version.

Choose **Project > Build**.

You can choose this in either the editor or the project window.

Notice the status bar in the Reversi project window after the build is complete:

```
Build completed with no warnings.
```

As well as removing the serious warning our semi-colon correction addressed, all the other serious warnings that were follow-on effects of the missing semi-colon have gone away. In addition, the single ordinary warning was removed by the *end* -clause fix.

Controlling the compiler's treatment of warnings

We have seen that serious warnings are caused by code that, if executed, would lead to a run-time exception. Some programming language compilers would refuse to link an executable file or DLL for such code, on the grounds that the code ought to be fixed before it is executed.

In Open Dylan, we can choose to go ahead and link in this situation. The choice is controlled from the main window, under the Build page of the **Options > Environment Options** dialog.

The option "Always link, even if there are serious warnings" forces the compiler to link an executable file or DLL for a project regardless of any serious warnings. We can also choose "Ask whether to link if there are serious warnings", and "Don't link if there are serious warnings". "Ask whether to link if there are serious warnings" is the default setting.

Problems at run time

Now we have taken a brief look at how Open Dylan treats compile-time problems, we will look at how it lets us debug problems that only emerge as exceptions at run time.

Note: The numbered example steps in this section lead us through a possible debugging scenario. In places the example is a little unrealistic. This is because usually you are familiar with at least some of the code you are debugging, and also because the main purpose of the example is to introduce features of Open Dylan.

With the rebuilt version of Reversi that compiled with no warnings, start a new game, with **Application > Start**.

After a couple of moves, save the new game by choosing **File > Save** in the Reversi window.

The Save dialog appears.

Choose a file to save into, and click **Save**.

An application error dialog appears.

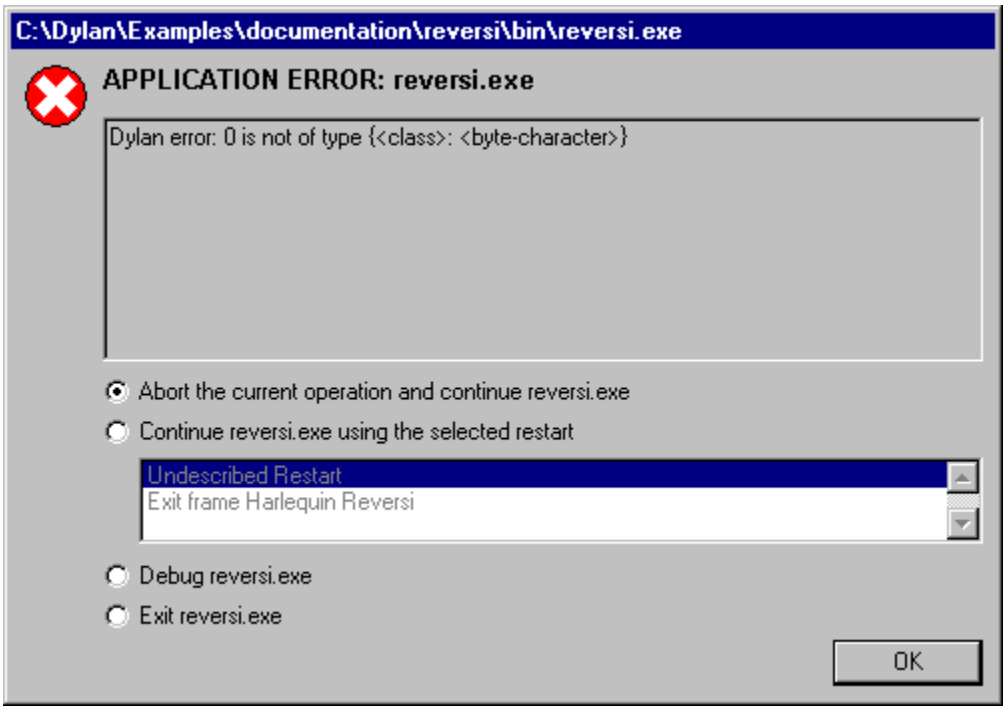

Fig. 4.3: A Dylan run-time application error.

The dialog appeared because the Open Dylan debugger caught an unhandled Dylan exception in the Reversi application. Something is wrong with the game-saving code. We must start up a debugger window to see what went wrong.

Choose Debug reversi.exe and click **OK** to enter the debugger.

The Open Dylan debugger appears. We discuss the debugger in detail in Debugging and Interactive Development.

In its uppermost pane, the debugger shows the error that it caught. It will be:

```
Dylan error: *n* is not of type {<class>: <BYTE-CHARACTER>}
```

where *n* is either 0, 1, or 2. (The value depends on the state of the game when we saved it. The reason for this will become clear shortly.)

In the left-hand pane beneath the message, there is a tree item for the main thread of the Reversi application. This tells us that the exception was raised in that thread. (In Reversi's case, there happens to be only one thread, but other applications might have multiple threads, and knowing the thread that raised the exception is useful. See Debugging and Interactive Development for more information about debugger options.)

When expanded, the tree item shows the current state of the call stack for Reversi's main thread. When the debugger is invoked on a thread, it pauses execution in that thread. So when we expand the tree we see the stack almost exactly as it was at the moment that the debugger was invoked.

The reason why what we see is *almost* exactly what the stack was like at the moment the debugger was invoked is that the stack pane normally filters out a lot of call frames that the Open Dylan run-time system creates. Because these are not frames that the running application creates directly, most of the time they are of no interest, and so it is convenient

to hide them. You can change the filtering with the drop-down list available directly above the thread pane in the debugger. The default filter setting is "Filtered visible frames".

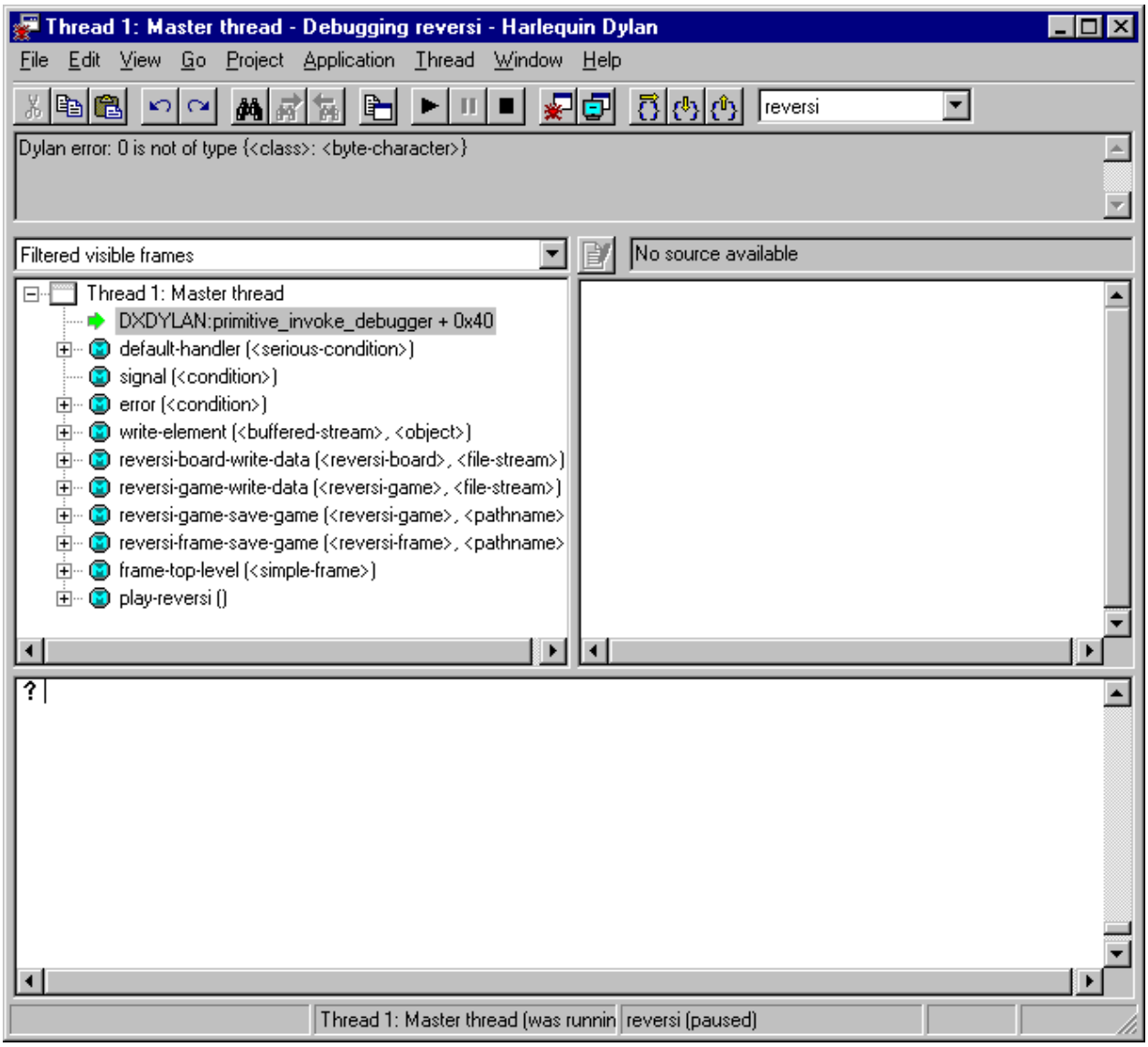

Fig. 4.4: The Reversi application stack after a game-saving error.

Each item in the list is a call frame on the stack for the thread being debugged. We call this list of call frames a stack backtrace or simply a *backtrace*.

The backtrace shows frames in the order they were created, with the most recent at the top of the list. The frames are represented by the names of the functions whose call created them, and are accompanied by an icon denoting the sort of call it was. See *Call frames* for details of the icons and their meanings, but note for now that the green arrow icon represents the current location of the stack pointer—that is, the call at which the thread was paused.

Searching the stack backtrace for the cause of the error

In this section we examine the backtrace and see what events led up to the unhandled exception.

Looking at the top of the backtrace, we can see that the most recent call activity in the Reversi main thread concerned catching the unhandled exception and invoking the debugger. The calls to `primitive_invoke_debugger`, `default-handler`, and `error` were all part of this. But if we move down the backtrace to the point below the call to `error`, we can examine the sequence of calls that led to the unhandled exception and find out how to fix the error.

The first interesting call for us is the one to `write-element`. This is the last of the calls appearing in the stack frame that Reversi made before the unhandled exception.

1. Select the call frame for `write-element`.

   The source code definition of `write-element` appears in the pane opposite. This source code pane is read only; if we wanted to edit a definition shown in it we would click on the Edit Source () button above the source code pane, which would open the file containing the definition in an editor window.

   Looking at the source code for `write-element`, the green arrow icon points to an assignment to `sb.buffer-next`. Here, the green arrow is showing the point at which execution would resume in that call frame if the application's execution was continued. What we do not know is whether the preceding call, to `coerce-from-element`, returned. It may be that the call failed (because the arguments were not what `coerce-from-element` was expecting) or that it succeeded but does not appear in the stack pane because of the default filtering.

   To work out what has happened, we can examine the stack pane filtering with the filtering drop-down list.

2. Choose "Filtered frames" from the stack pane filtering drop-down list (which by default is set to "Filtered visible frames"). The stack pane updates itself.

The six settings available from the stack pane filtering drop-down list provide a quick way of changing what you view in the stack pane:

All frames Shows all frames in the thread being debugged.

All visible frames Shows all the frames in the thread that are part of the module's context, in this case the reversi module's context, which includes calls to any functions imported from other modules.

All local frames Shows all frames defined in the current (reversi) module.

Filtered frames Shows a filtered list of function calls in the thread being debugged.

Filtered visible frames Shows a filtered list of function calls in the current module plus calls to functions imported from any other modules used.

Filtered local frames Shows a filtered list of function calls from the current module only.

The "Filtered…" settings do not, by default, show foreign function calls, cleanup frames, and frames of unknown type, whereas the "All…" settings show everything. You can set the filtering rules using **View > Debugger Options…**, see *Stack options* for details.

So the question is whether the call to `coerce-from-element` failed, or whether it succeeded, but comes from a module that Reversi does not explicitly use. The stack pane now shows a frame for the call to `coerce-from-element`. The name has the suffix `streams-internals:streams`. This means that `coerce-from-element` is a name from the `streams-internals` module of the `streams` library.

This *name*:*module*:*library* form of printing Dylan names is used in a number of different places in Open Dylan. It shows that *name* is not part of the module, or module and library, that a tool is currently focused on. (The debugger and browser both have a toolbar pop-up where you can change the current module.)

Returning to our example, we now know that `write-element`'s call to `coerce-from-element` succeeded, because it created a call frame. We can see that `coerce-from-element` is now the last frame on the stack before the call to `error`.

3. Select the call frame for `coerce-from-element`.

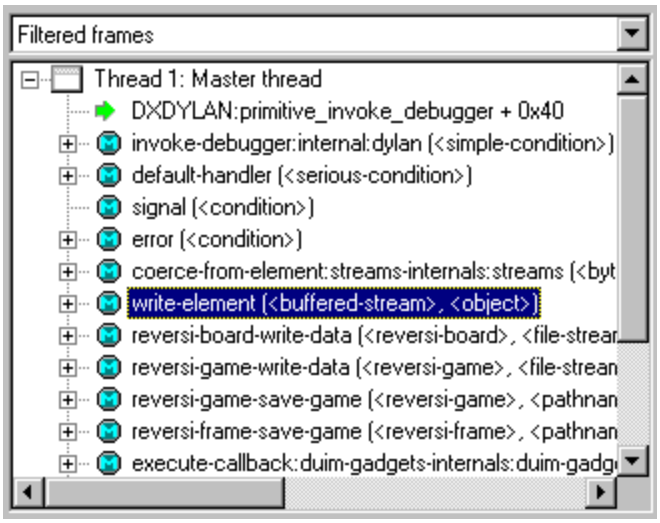

Fig. 4.5: Stack pane showing call frames from all modules.

The green arrow in the source code definition for `coerce-from-element` points to an assignment containing a call to `byte-char-to-byte`. Notice that this call does not appear in the backtrace. Because the backtrace is now showing call frames from all modules, we know that the exception must have been raised while attempting to call this function, before a call frame was created for it.

Since the error dialog told us that the exception was caused by something being of the wrong type, there is a good chance that the value of *elt*, the argument to `byte-char-to-byte`, is of the wrong type. Notice too that *elt* 's type is not specified in the signature of `coerce-from-element`.

We need to know the value passed to *elt*. We can find out by expanding the `coerce-from-element` call frame: a call frame preceded by a + can be expanded to show the values of its arguments and local variables.

4. Expand the call frame for `coerce-from-element`.

   We can now see the value that was passed for *elt*. It is an integer value, either 0, 1, or 2. It is this value that caused the error that occurred. This is the message again:

```
Dylan error: *n* is not of type {<class>: <BYTE-CHARACTER>}
```

   where *n* is either 0, 1, or 2.

   Our next task is to find out why `coerce-from-element` was sent an integer instead of a byte character. To do this, we can simply move down the backtrace and examine earlier calls.

5. Select the call frame for `write-element`.

   We can see here that the value passed to *elt* in `coerce-from-element` is the value of one of `write-element` 's parameters, also called *elt*.

   We need to move further down the stack to the `reversi-board-write-data` call.

6. Select the call frame for `reversi-board-write-data`.

The `reversi-board-write-data` method takes an instance of `<reversi-board>` and an instance of `<file-stream>` as arguments. A `<reversi-board>` instance is what the application uses to represent the state of the board during a game. A `<file-stream>` is what Reversi is using to write the state of the board out into a file that can be re-loaded later.

We can see that this method calls `reversi-board-squares` on the `<reversi-board>` instance and then iterates over the value returned, apparently writing each element to the stream with `reversi-square-write-data`.

(Notice that `reversi-square-write-data` does not appear on the stack—this is because it contains only a tail call to `write-element`, and so is optimized away.)

We are closing in on the bug. It is looking like the value representing the Reversi board squares (*squares*), and the file stream the squares are being written to (*stream*), have incompatible element types, with the squares being represented by integers, and the file stream being composed of byte characters.

Browsing local variables

In this section we use the Open Dylan browser to help confirm the cause of the unhandled Dylan exception.

Expand the call frame for `reversi-board-write-data`.

We can now see the values of the local variables in this frame. The arguments are listed first: *board* and *stream*, followed by the *squares* sequence and iteration variable *square*.

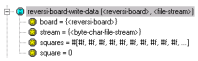

Fig. 4.6: Local variables in the `reversi-board-write-data` call frame.

The notation

```
board = {<reversi-board>}
```

means that *board* is an instance of `<reversi-board>` —an actual instance in the paused application. The curly braces mean that this is an instance of the class rather than the class definition itself.

We can look at this `<reversi-board>` instance in the *browser*, which allows us to examine the contents and properties of all kinds of things we come across in Open Dylan.

Double-click on the *board* item.

The browser appears.

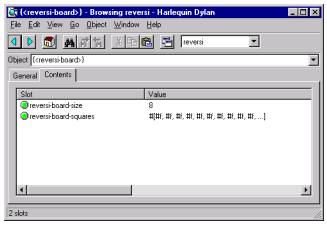

Fig. 4.7: Browsing an instance of `<reversi-board>`.

The browser shows us in its Object field that we are browsing an instance of `<reversi-board>`. Like the debugger, the browser uses the curly braces notation to depict an *instance* of a class as opposed to its definition.

The browser presents information in property pages. In the page selected by default, we see the names of the slots in the instance and the values they had when the exception occurred. The property pages that the browser shows depend on what it is browsing; the set of pages for a class definition is quite different from that for a method definition, for example.

However, the browser always provides a General page. The General page gives an overview of the currently browsed object.

Choose the General page.

The fields on the General page for our `<reversi-board>` value tell us that it is an instance of type `<reversi-board>` and that it has two slots. The third field, Source, is labeled "n/a" for "not applicable". The Source field shows a source file name for anything the compiler saw during compilation, such as a definition. We are browsing an instance, not a compiler record, so it is not relevant to associate the instance with a source location. For more on the browser's distinction between run-time and compile-time objects, see *Browsing a project in source and run-time contexts*.

Choose the Contents page.

If we double-click on items on the Contents page, the browser moves on to browsing them.

Double-click on the *reversi-board-squares* item.

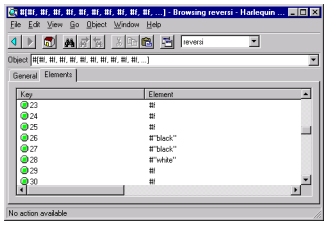

Fig. 4.8: Browsing the elements of a collection.

Now we can see the elements of the *reversi-board-squares* collection.

Click on the Back (◙) button to return to browsing *board*, the `<reversi-board>` instance.

Going back to the bug we are tracking down, two more useful pieces of information have emerged from seeing the `<reversi-board>` instance in the browser.

First, we can tell from the Contents page, which shows the slot values in the instance, that the call to *reversi-board-squares* in `reversi-board-write-data`, below, is clearly just a call to the default accessor on the `<reversi-board>` slot of the same name.

```
define method reversi-board-write-data
    (board :: <reversi-board>, stream :: <file-stream>)
=> ()
  let squares = reversi-board-squares(board);
  for (square from 0 below size(squares))
    reversi-square-write-data(squares[square], stream);
  end for;
end method reversi-board-write-data;
```

Second, we can see that the *reversi-board-squares* slot holds a sequence, and that the sequence does not have an `<integer>` element type.

So we still do not know where the integer that caused the exception came from. However, we have yet to check what goes on in `reversi-square-write-data`; perhaps that method is converting the elements in the *reversi-board-squares* sequence into integers?

Browsing definitions

In this section, we browse the definition of `reversi-square-write-data` to see whether it converts the board squares into integers.

To browse the definition, we have the option of locating it on the project window Definitions page or (more efficiently) moving it directly in the browser.

Delete the text in the browser's Object field and type `reversi-square-write-data` in its place.

Press Return.

The browser switches to the definition of the `reversi-square-write-data` method. When we browse a definition as opposed to an instance, the browser usually shows a larger set of property pages that supply a lot of information about the definition and the relationships between it and other definitions in a project. The default property page here is the Source page, which shows the source code for the method.

Here is the code:

```
define method reversi-square-write-data
  (square :: <piece>, stream :: <file-stream>)
=> ()
  write-element(stream, number-for-piece(square));
end method reversi-square-write-data
```

So *number-for-piece* is most likely returning the integer value that was passed to `write-element` (and that we can see on the stack as the *elt* local variable). The square value has type `<piece>` —this, then, is the element type of the sequence used to represent the state of the board.

Browse the definition of `number-for-piece`.

You can do this either by typing the name into the Object field, or by clicking on the name on the Source page and selecting *Browse* from the right-click popup menu.

The definition of `number-for-piece` completes the story. It is here that the board square representations are converted into integers. This is where the integer that caused the exception came from.

Fixing the error

In this section, we fix the Reversi project source code to eliminate the cause of the exception we have been tracking down.

This is what we have learned about the error so far:

- It occurred when trying to save a Reversi game.
- It was caused in a call to `coerce-from-element`, which attempted to pass an integer to *byte-char-to-byte*, a method which expects an instance of `<byte-character>`.
- The `coerce-from-element` method received the integer from `write-element`, which received the integer from `reversi-square-write-data`.
- The `reversi-square-write-data` method uses the `number-for-piece` method to translate board square representations (type `<piece>`) into instances of `<integer>`. The `<piece>` values are either `#f` (no piece on this square), `#"white"` (a white piece on this square), or `#"black"` (a black piece on this square); those values are translated into 0, 1, and 2 respectively. That is why *n* could have been either 0, 1, or 2 in the error message:

```
    Dylan error: *n* is not of type {<class>: <BYTE-CHARACTER>}
```

So the value of *n* depends on the state of the first square to be written.

In addition:

- The `write-element` generic function is from the Open Dylan Streams library. It is part of that library's protocol for writing to streams.
- The stack shows that the `write-element` method tried to coerce an integer to a byte character, and that the attempt failed.

So we know that Reversi is trying to write integer values to a file stream with a <byte-character> element type, and the exception occurs during the attempt to coerce an integer into a byte character.

We could simply change the file stream's element type to <integer>.

In fact, we have not yet looked at the call that created the file stream. That call is reversi-game-save-game.

Return to the debugger and select the call frame for reversi-game-save-game.

As expected, the source pane shows that the file stream is created with an element type of <byte-character>. The relevant code fragment is:

```
let file-stream = make(<file-stream>, locator: file,
                       direction: #"output",
                       element-type: <byte-character>);
```

Click the Edit Source () button above the source code pane.

An editor window opens on *saving.dylan*.

We now have *saving.dylan* in the editor, and the insertion point is positioned at the start of the definition for reversi-game-save-game. We can make the change to <integer>, but should first check reversi-game-load-game, the method that loads games saved by reversi-game-save-game, to see what sort of file-stream elements it expects to read back.

That definition is located directly below that of reversi-game-save-game. It shows that the file-stream element type expected is <byte>.

```
let file-stream = make(<file-stream>, locator: file,
                       direction: #"input",
                       element-type: <byte>);
```

The class <byte> is actually a constant value, defined:

```
define constant <byte> = limited(<integer>, min: 0, max: 255);
```

So there is no harm in changing the element-type: argument in reversi-game-save-game's call to make from <byte-character> to <integer> (because 0, 1, and 2 are all within the defined range for <byte>), but for symmetry we may as well change it to <byte>.

Fix the definition of reversi-game-save-game.

The element-type: keyword in the call to make on <file-stream> should take <byte>, not <byte-character>.

Choose **File > Save** in the editor.

Before we can rebuild the application we need to stop the current version running.

Choose **Application > Stop** in the editor.

A dialog appears asking you to confirm that you want to stop the application.

Click **OK**.

Rebuild the application with **Project > Build**.

Start the application again, and try to save a game.

The save operation now works without raising an unhandled exception.

Loading the saved game back in

The next step is to test the code for loading a saved game. To test this we need to change the state of the board from what it was like when we saved the game.

Clear the Reversi board by clicking **New Game** in the Reversi application.

Choose **File > Open** in the Reversi application, select the file you saved the game into, and click **Open**.

Reversi now shows the state of the game you saved earlier.

CHAPTER

FIVE

PROGRAMMING IN OPEN DYLAN

Now we have taken a tour of Open Dylan using the pre-written Reversi application, we take a step back to look at the programming model in Open Dylan, and to review the features of the development environment and the Dylan compiler.

Projects

In Open Dylan, all development work is done in terms of *projects*. Projects are the development environment and compiler's way of representing Dylan libraries.

A project consists mainly of a list of the source files that define the library, but also contains information about how to compile the library.

While it is possible to edit text files that are not associated with any project, nearly all other programming tasks in Open Dylan take place within the context of a project.

Projects and libraries

A project represents a single Dylan library. Think of a project as something that gathers together all the information Open Dylan needs to be able to compile a particular library.

For example, earlier in this manual, we worked with the Reversi project. The Reversi project represents a single Dylan library, called Reversi.

Projects and deliverables

You can create deliverable applications, libraries, and components from projects. Projects can be built into executable (.EXE) or dynamic-link library (.DLL) files.

Use the *Project* menu for project building. See Delivering Dylan Applications, for details of delivery to customers.

When we worked with the Reversi project, we built an executable from it, but we could just as easily have built a DLL. See *Project settings* for details.

Creating new projects

Open Dylan includes a New Project wizard for creating new projects.

The New Project wizard asks questions about what you are going to develop, and then creates a new project configured to help you get working quickly. See *Creating a new project*.

You can also create a project if you have a Dylan Library Interchange Description (LID) file for it. LID files have a .LID extension. When you open a LID file in the development environment, it is converted into a project file and opened in a project window. (This process does not modify the original LID file on disk.) See *Project files and LID files*.

Project files

Open Dylan stores projects on disk in *project files*. Project files have the same name as the project, with the extension . HDP. Thus a project called Hello is stored in the file *Hello.hdp*. (Projects get their names in the New Project wizard or, if they were created by conversion from a LID file, get their name from the name of the library that the LID file defines.)

Project files are the files you select when you want to open a project or add it to another project as a subproject.

The development environment tools automatically save changes to project files. For example, if you add a new source file to a project, the change is saved to disk immediately.

Project components

We now describe the components of a project in more detail. Every project consists of:

A Dylan library Every project defines a single Dylan library. We call this library the "library of the project" or, for clarity, the *main* library of the project, to distinguish it from other libraries that the project uses.

A project name Every project has a name. When you create a new project with the New Project wizard, the wizard uses this name to generate default names for initial files, libraries and modules in the new project. The compiler uses the project name to generate default names for the executables and other files it produces during compilation. In both cases, you can override the defaults.

Source code files, and other files Every project includes source code files. Projects created with the New Project wizard will have a *library.dylan* file (which defines the project's library); a *module.dylan* file (which defines the modules of the library); and at least one other Dylan source code file containing definitions and expressions.

Projects can also include Windows resource files, static libraries (.LIB files), and text files. The compiler ignores any file it does not recognize. This flexibility allows such things as including README files in a project.

Subprojects A project can have *subprojects*. Subprojects are other projects that are included in a project; they define their own main library, contain their own source files and may have subprojects themselves. For clarity, we can call a project a *superproject* when describing it with reference to its subprojects.

See *The build cycle* for more on the relationship between projects and their subprojects.

Version numbers Every project has a major and minor version number. The version numbers affect the build process for projects. See *Link page* and *The build cycle*.

Project settings Every project has settings. Among these settings are:

- The list of source code files and their locations on disk.
- Compilation mode options. See *Compilation modes*.
- Debugging options. See *Debugger options*.
- The list of subprojects the project uses.
- The locations on disk of the subprojects.

Projects on disk

A project consists of several files and folders on disk.

First, all the information necessary to build the project is stored in a *project file* (.HDP file). Then there are the Dylan source files, and possibly Windows resource and static library (.LIB) files, that make up the code for the project.

The files that make up a project are stored in a folder called the *project folder*, which normally has the same name as the project. The files are stored in the project folder and in several subfolders of the project folder. The files themselves can refer to other folders where subprojects and used libraries are stored.

The project folder contains the following files and subfolders:

- The project file. (.HDP file.)
- The source code files. (.DYLAN files)
- The *bin* folder.

  This folder holds the executable (.EXE) or DLL (.DLL) file produced from the project.

  In addition, the DLLs of the project's subprojects are automatically copied into this folder, so that they can be found when you execute your project's application.
- The *project -build* folder.

  This folder, whose name begins with the name of the project, holds a number of intermediate files produced during builds. You will never have to do anything with these intermediate files.

  The folder also contains the *compiler database* file for the project. This file has the same name as the project and the extension .DDB. See *Compiler databases* for more details.

  You can remove the compiler database and intermediate files with **Project > Remove Build Products**. This forces a complete recompilation of a project next time you build it.
- The *lib* folder.

  This folder holds the *linker* file for the project. This file has the same name as the project and the extension .LIB or .DEFS. This file is needed for other projects to be able to link against the project, a process that is part of using a project as a subproject.

  The extension is .LIB if you are using the Microsoft linker, or .DEFS if you are using the GNU linker.
- The *release* folder.

  This folder holds a stand-alone version of the project's application, suitable for redistribution to customers or other third parties without a copy of Open Dylan on their system. It is created when you choose the **Project > Make Release** command.

Projects in the development environment

The Open Dylan development environment offers a variety of ways to examine and manipulate projects. You can view a single project in multiple windows at the same time. You can also have more than one project open in the environment at a time.

Apart from the main window and dialog boxes, windows in Open Dylan are generally instances of programming tools. The tools provide views onto different pieces of a project, or sometimes different views of the same pieces.

For example, you might want to have editor windows open on multiple files in the project, as well as browser windows to show you structural views and debugger windows to show you stack backtraces or other information from a running program.

As we saw when touring the environment with the Reversi example, Open Dylan offers:

- A project window.
- A debugger for examining and interacting with paused application threads associated with open Dylan projects.
- A browser for examining the contents and properties of projects and of the objects in paused application threads associated with open Dylan projects.
- An editor for source files. Editors are most often invoked from other windows on a project, but can be invoked on files outside the context of a project.

Development models

The process of development in Open Dylan can be much the same as in interactive development environments for other languages. Applications written in Dylan can be developed in the same way as applications written in static languages like C and C++, for instance.

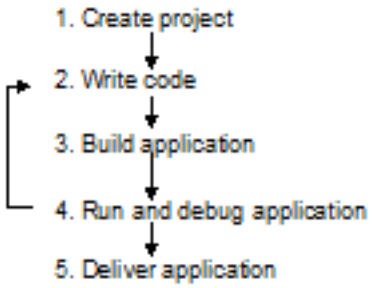

Fig. 5.1: "Static" development model.

You can also develop applications in a more dynamic fashion, using features in the debugger and browser tools that allow you to interact with a running application. With these dynamic, interactive features, you can test bug fixes on the fly and keep your application running before committing to a rebuild.

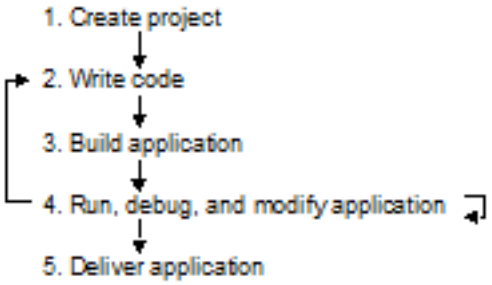

Fig. 5.2: "Dynamic" development model.

Interactive and incremental development

Open Dylan offers both interactive and incremental development features. It is important to distinguish them clearly:

Incremental development is the ability to recompile portions of a project and save the resulting object code. By contrast, some compilation systems require that the entire project be recompiled in response to any change, however small. Open Dylan always performs incremental compilations when it can, to keep build times as short as possible.

Interactive development is the ability to execute code fragments, including definitions and redefinitions, in a running program. Open Dylan offers interactive development via the debugger's interaction pane. The object code produced during interactive development is not saved, but just patched into the running program and added to the in-memory *compiler database* (see *Compilation modes*). The object code is lost when the program terminates.

Compilation

This section discusses compilation modes, compiler databases, optimization (including loose and tight binding), the build cycle algorithm, and linkers.

Compiler databases

When compiling a project, Open Dylan produces a compiler database which models the project. The database provides a rich source of information to Open Dylan tools about the contents, properties, and relationships between source code definitions, libraries, and modules.

A project's compiler database is used when browsing and debugging the project, and is also used when compiling other projects that use the project.

The compiler database for a project does not exist until the project has been built for the first time. Before then, if you try to do anything that requires the database, the development environment will ask you if you want to create it.

Once the compiler database has been built, the development environment will ensure it is kept up to date with each recompilation of the project.

Open Dylan stores project files on disk for persistence between sessions. When you close a project, the development environment checks whether the database has changed since it was last saved, and if it has it asks you if you want to save the database. (You can use **File > Save Compiler Database** from the project window to save the compiler database at other times, if necessary.) When you re-open the project later, the database is read into memory from the disk file, if it exists.

Compiler database files have a .DDB suffix.

Compilation modes

The Dylan language encourages programmers to write programs that can be compiled as efficiently as programs written in static languages. By adding type declarations and sealing to your project code, the Open Dylan compiler can optimize it very successfully.

However, the best optimizations come at the costs of longer build times, and less symbolic information in the debugger. During the larger proportion of your project's development, you want projects to build quickly and to be easier to debug. When it is time to deliver your product, you will want to turn all the code optimizations on even at the expense of debugging information and compilation speed.

Like other compilation systems, Open Dylan allows you to switch between both styles of compilation. For any project, you can specify the style of compilation to perform by choosing **Project > Settings** in any window with a **Project** menu, and then choosing the Compile property page.

That page offers two mode choices:

- Interactive Development mode

- Production mode

You should do the majority of your work on a project in Interactive Development mode. When compiling a project in this mode, the compiler does not perform as many optimizations as it can, and is not as strict about error checking as it can be. The idea here is to keep compilation times as short as possible.

This mode keeps symbolic information in the compiled code that will make debugging work easier. Also, if your project was compiled in this mode you will be able to do more interactive work in the debugger's interaction pane, including redefinition. However, compiled code will not be as fast as it can be.

When your project work is nearing completion, and you want to see the compiled version running as fast as possible, switch to compiling the project in Production mode. Production mode turns on all compiler optimizations. However, build times will be slower than in Interactive Development mode, and debugging and interaction will be more limited.

When you have switched to Production mode, you can use Open Dylan's *optimization coloring* feature to highlight inefficiencies in your code. This feature colors source code so that you can see where optimizations did and did not occur. Adding type declarations and sealing will secure new optimizations, which you can verify by refreshing the coloring after rebuilding the project. See Dispatch Optimization Coloring in the Editor.

Versioning

A project can have major and minor version numbers that will be recorded in the DLL or EXE that the project builds. You can enter these numbers on the **Project > Settings...** dialog's Link page.

Open Dylan uses version numbers at compile time and run time to determine if compatible versions of Dylan libraries are in use.

The rules differ for compilation in Interactive Development mode and Production mode. For applications compiled in Interactive Development mode, the procedure at run time for initializing a library involves checking the major and minor versions of the Dylan libraries used by the library being initialized. If the major version number of a used library does not match that of the library using it, or the minor version number of a used library is lower than that of the library using it, the Open Dylan run-time system signals an error.

In Production mode, the run-time check ignores the user-supplied version numbers and checks whether the used library is the very same one that was used at compile time. If the library is different, a run-time error is signalled even if the version is the same.

Binding

Interactive Development mode and Production mode are in fact combinations of some lower-level compiler modes. Open Dylan presents these two compilation modes to make development simpler, but some understanding of these lower-level modes is useful. They are *loose binding* and *tight binding*.

- Loose binding This is a way of compiling code that makes no use of the type information available in the source. When the compiler is run using loose binding, it considers only names and macro definitions. References to objects and types are always made indirectly through the objects' names, so that the objects can be changed without forcing recompilation of code that uses them.
- Tight binding This is a way of compiling code that uses all type information available in order to drive optimizations. This type information includes declared types and some inferred types. Tight binding bypasses names, referencing objects and types directly. Amongst other optimizations, tight bindings inlines some methods, performs tail-call elimination, and removes unused code. These optimizations can affect the information seen in the debugger.

Code can be loosely or tightly bound within a library, and it can be loosely or tightly bound with respect to other libraries. If code within a library is loosely bound, other libraries will be loosely bound to it. Similarly, if code is tightly bound within a library, other libraries will bind tightly to it.

The code within all libraries that Open Dylan supplies—the system libraries—is tightly bound. This means that all libraries you develop will bind tightly to whichever of the system libraries you use.

When libraries are compiled in Interactive Development mode, they are loosely bound internally, and therefore libraries that use them will be loosely bound to them. When libraries are compiled in Production mode, they are tightly bound internally, and therefore libraries that use them are tightly bound to them.

The build cycle

Building an application or DLL from a project consists of up to three phases:

1. Building the subprojects.
2. Compiling some or all of the project source code.
3. Linking the project.

For efficiency, when the compiler is asked to build a project it minimizes the number of these phases that it performs, using the following decision rules:

- If phase 2 or 3 is performed, the project is considered changed.
- A *clean build* always performs all phases for the project and its subprojects.

You can ask for a clean build by choosing **Project > Clean Build** in any window that has a **Project** menu.

- A build command is always recursively performed on subprojects (phase 1).
- If the major version number of any subproject has been changed, then all of the source code in the project is recompiled.
- If the project is tightly bound to any subproject which has changed, then all the source code in the project is recompiled.
- If the project is tightly bound to itself, and if any source code in the project has changed, then all the source code in the project is recompiled.
- If the project is loosely bound to itself, then any source code files that have changed are recompiled. Additionally, files that depend on those changes (such as through macro usage) are recompiled.
- If the project or any of its subprojects has changed, then the project is relinked.

Note: To ensure change propagation according to these rules, you should always increment the major version number of a project after altering any macro definitions in it.

Linkers

Open Dylan offers you a choice of linkers to use to link your Dylan programs. The default linker is a GNU linker. If you own Microsoft Developer Studio, you can use Microsoft's linker instead. See the Linker page of the main window's **Options > Environment Options...** dialog.

Executing programs

This section discusses running applications within Open Dylan (and the benefits of doing so), and the process of library initialization in an application.

Starting applications up from within Open Dylan

An application written in Dylan cannot be started and later connected to Open Dylan and its project. If you want to be able to debug an application and browse its compiler database within Open Dylan, you must start it up by opening its project and starting it with **Application > Start**. This starts the application up under the debugger, providing the development environment with a connection to the application and the capabilities necessary to control its execution and to interact with it.

Application and library initialization

When a Dylan application starts up, it begins by loading the libraries that it uses. Each library performs its own initialization when it is loaded. In general, libraries are loaded in a demand-driven, depth-first order. However, you should not depend on used libraries being loaded in the same order that they are mentioned in a library definition.

Library initialization is performed by executing the code which comprises the library, in the order in which it is defined by the library's project. This means that the order of the Dylan source files in a project is significant, and that the order of definitions and expressions in a Dylan file is significant.

Definitions in a Dylan library are not, in general, said to execute. Rather, they define the static structure of a program. This is true of variables and constants initialized to literal values or other values computable at compile time, and it is also true of classes and functions. Forward references to such objects are allowed, and all such objects are created at the start of library initialization, before expressions are executed. Some definitions rely on the computation of run-time values; in these cases, forward references may not be allowed.

Expressions in a Dylan library are executed in the order in which they appear in the project, and the last expression in a project should be a call to a project's start function.

Source, database, and run-time views

We have seen that Open Dylan provides several tools to allow us to view projects in different ways. Some tools can look at the source representation of a project, while others can look at the run-time representation—the threads of a running application built from a project.

It is useful to think of there being three "worlds" in which we can simultaneously view projects: source, database, and run-time.

Every project has a representation in source code. We view this source representation with the editor mainly, but the debugger's source pane can show us the source code for a function on the stack, and the browser can show the source for some kinds of object in its Source page.

When we build a project, the compiler database that is created provides a second representation. Then, when we run the application or DLL we have created, the running program is itself a third representation of the project.

So, at any given time, an object may exist in each of these worlds simultaneously. The source code of the object may exist in a Dylan source file, a model of the object may exist in the compiler database, and the object may be instantiated in a running program.

Editor windows show projects in their source representation only. Browser windows show information from the compiler database, and, if a program is running, this database information is combined with information from the program, so you can see the "live" version of the object.

The debugger and its interaction pane allows you to view the threads of running programs, and allows you to execute expressions and definitions in these threads. When you do this, the running program is modified. When you enter definitions in this way, the definitions are saved in a temporary layer of the compiler database so that browsing will continue to be accurate. However, these temporary changes are not saved to disk in the compiler database file, nor are they reflected in the project source code files.

There are ways in which the three worlds can get out of sync. Remember that if you edit a source code definition, the model of it in the database will not be updated until you rebuild the project. So, for instance, if you change the inheritance characteristics of a class, the change will not be reflected in the browser Superclasses page for that definition until you rebuild. And if you add new definitions to the project sources, they will also not be visible until you build the project again.

CHAPTER

SIX

CREATING AND USING PROJECTS

In this chapter we look more closely at projects. We take a break from Reversi and develop some new projects of our own. To do this, we use Open Dylan's New Project wizard. We also look at some of the options and settings associated with projects, and some of the features of the project window.

Creating a new project

We now create a new project for a simple "Hello World" application that will run in an MS-DOS console window.

We create the project by clicking the New Project () button in the main window, or by choosing **File > New** in any window.

Click the New Project button in the main window.

The New Project wizard appears.

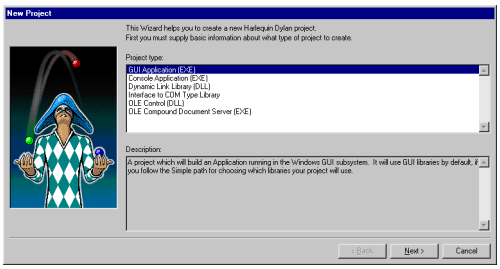

Fig. 6.1: The New Project wizard.

The New Project wizard guides us through the process of creating a new project. Across a series of pages, it gathers the following information:

- The type of target file (.EXE or .DLL) that should be created when the project is built.
- The name of the project and a folder on disk for storing its files.
- The libraries and modules that the project will use.
- Documentation keywords to be used in the project's various files.

When we have supplied all the information that it requires, the wizard creates a new project, consisting of the following files:

- A project file. The file has the same name as you give the project, but with a .HDP extension.
- A Dylan source file, called *library.dylan*, that defines a library with the same name as the project

- A Dylan source file, called *module.dylan*, that defines a module with the same name as the project
- An initial Dylan source file into which we can write application code. The file has the same name as the project, but with a .DYLAN file extension.

Some application types contain:

- A Dylan source file containing some useful constant and function definitions. The file has the same name as the project, with *-info* appended, and with a .DYLAN file extension. So for a project called Hello, the file would be *Hello-info.dylan*.

The *project -info.dylan* file appears only if you ask for template code to be included in your project.

And for GUI applications that use the DUIM library, there can also be:

- A Dylan source file, called *frame.dylan*, that defines the application's DUIM frame and a set of default menus.

The *frame.dylan* file appears only if you ask for template code to be included in your project. We discuss this in *Projects for GUI applications*.

For more information on DUIM, see *Building Applications Using DUIM* and the *DUIM Reference*.

Specifying the type of the project

The first page in the New Project wizard asks us to specify the kind of target we want to build from the project.

In the Project type box, the default option is "GUI Application (EXE)". Our "Hello World" application is going to be an MS-DOS console application, so we need to change this option setting from the default.

Select "Console Application (EXE)" in the Project type box.

Click **Next**.

We now move to the second page.

Specifying the project name and location

The second page in the New Project wizard asks us to supply a name for our project, and to specify a location for the automatically generated project and source files.

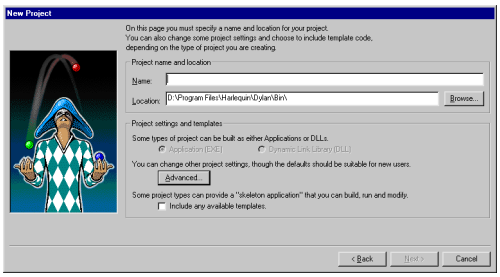

Fig. 6.2: The New Project wizard's second page.

When naming projects, remember that Open Dylan uses the name given to a project as a basis for naming some of the files that make up the project, so the name should only contain characters that are valid in Windows file names.

In addition, the project name is used as the default name of the library and module that the project defines, so unless you change those names (with the *Advanced...* dialog; see *Advanced project settings*), the project name must also be a valid Dylan name.

Here we can also specify, using the "Include any available templates" check box, that the wizard should generate skeleton code for our project if a template for the project type is available.

Template code provides a skeleton version of the sources for a project of the type we are creating. We can then modify the skeleton code to fit our needs. The content of the skeleton code not only reflects the type of project selected on the first page, but also our answers to subsequent questions that the wizard asks.

Since we want to develop a simple "Hello World" application, we are going to call the project Hello.

Type **Hello** in the Name box.

As you type Hello, the Location box fills in a folder for the automatically generated project and source files.

The New Project wizard will create the folder in the Location box automatically if it does not already exist.

Click **Next**.

We now move to the third page.

Choosing the libraries that the project uses

The next stage in creating our project is to decide which libraries and modules it is going to use. The third page of the wizard offers three different ways to do this. Each of the three options is described below.

- Minimal If we choose this the project uses the Functional-Dylan library only.
- Functional-Dylan is a convenience library that combines the standard Dylan library with a language extensions library called Functional-Extensions. Thus Functional-Dylan provides a "Harlequin dialect" of Dylan. (The standard Dylan library, without Harlequin's extensions, is also included in the set of Open Dylan libraries.)
- Simple If we choose this the wizard presents a series of choices we can make to determine which libraries and modules the project should use.
- Custom If we choose this the wizard presents a table of all the libraries and modules available, and allows us to select the ones we want our project to use.

All our Hello project needs to do is print a text message saying "Hello World" to the standard output in an MS-DOS console window. The Functional-Dylan library contains a function to do this, so for our project we can select the Minimal button and move on to the next page in the wizard.

Select **Minimal** in the Use Libraries box.

Click **Next**.

We now move to the final page.

The final page in the New Project wizard

The final page of the New Project wizard gives us the option of supplying text for the documentation keywords *Synopsis:*, *Author:*, *Copyright:*, and *Version:*.

If we supply values for these keywords, the wizard adds them to the top of each of the files that it creates for the project, including the project file itself. With the exception of *Synopsis:*, these keywords are defined as part of the Dylan interchange format, on page 23 of the DRM. *Synopsis:* is a not a standard Dylan interchange keyword, but an additional one that Open Dylan accepts.

Change the default keyword text as you wish, or turn the keywords off altogether.

Click **Finish**.

Now we have supplied all the information the wizard asks for, it creates the new Hello project and opens it.

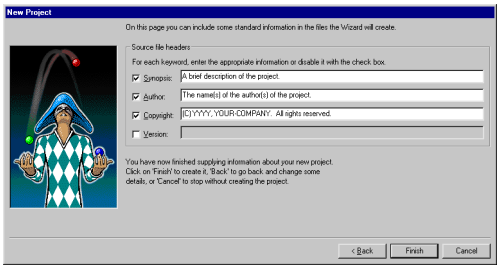

Fig. 6.3: The New Project wizard's third page.

Examining the files in the Hello project

The Hello project. shows our new Hello project.

The default view shows the Sources page, where we can see the files *library.dylan*, *module.dylan*, and *Hello.dylan*.

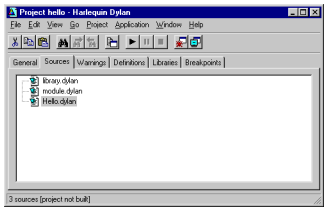

Fig. 6.4: The Hello project.

The *library.dylan* file defines a Dylan library called *Hello*, which uses the library *harlequin-dylan*. The *module.dylan* file defines a module of the *Hello* library which is also called *Hello*, and which uses various modules exported from the *harlequin-dylan* library.

The *Hello.dylan* file is an initial file into which we can write the code for our project. It contains a default start function called *main*, and the last lines of the file call this *main* function. For more on the purpose of this function, see *The project start function*.

We can add further files to the project as we see fit. But our "Hello World" application is trivial: we can write the code into *Hello.dylan* now, and our work will be done. The application will simply call the function *format-out* on the string *"Hello World\n"*. The *format-out* function (exported from the *simple-format* module) formats its argument on the standard output.

Open the *Hello.dylan* file in the editor.

Add the following code in the definition of *main*:

```
format-out("Hello World\\n");
```

Choose **File > Save** to save the change to *Hello.dylan*.

Now we can build our "Hello World" application.

Choose **Project > Build** in the project window.

Test the application by choosing **Project > Start**.

An MS-DOS console window appears, into which "Hello World" is written. Then a notifier dialog appears to confirm that the console application has terminated.

You can find the *hello.exe* file in the *bin* subfolder of the *Hello* project folder we specified on the second page of the New Project wizard. See *Projects on disk* for more details of where build products reside.

Projects for GUI applications

In this section, we define a more typical project. This project will be a for GUI application. To do this, we take a different path through the New Project wizard. We look at the project files that the wizard creates, then build and run our GUI application.

Creating a GUI project

First, we create the new project for our GUI application.

Click the New Project () button in the main window.

On the first page, we want to specify the project type.

Select "GUI Application (EXE)" in the Project type box.

Click **Next**.

We now move to the second page of the wizard.

Here, we want to name the project and specify a folder for its files.

Name the project *GUI-App* and choose a location for it.

The New Project wizard can set up some skeleton program code for our project, according to the project's characteristics as we specify them. Template code is not relevant for all kinds of projects—for instance, our Hello project would not have benefited from any more initial program structure than it had—but the wizard will include any that is relevant if we check the "Include any available templates" box.

Make sure the "Include any available templates" box is checked.

We ignore the **Advanced...** button again.

Click **Next**.

We now move to the third page of the wizard.

When we created the Hello project, we chose the Minimal option here, to use only the Functional-Dylan library. Our GUI application also needs to use other libraries for access to the native window system.

Select "Simple" in the Use Libraries box.

Click **Next**.

We now proceed through a series of pages allowing us to specify our project requirements in high-level terms, without knowing the names of specific Open Dylan API libraries.

The wizard will make our project's library definition use the right libraries and modules to do what we ask on these pages, and will include suitable template code in the project sources. Thus the Simple option is a useful way to create projects until you are more familiar with the libraries that Open Dylan offers.

On the first page we can specify the what I/O and system support we want in our project. For each option, the wizard shows which libraries the project will use.

Leave the default settings on this page as they are, and click **Next**.

The next page is for specifying GUI support details. Here, we can decide whether we want to do the window programming for the application by using DUIM, Open Dylan's high-level GUI toolkit, or by using the Win32 API libraries described in the *C FFI and Win32* library reference. We want to use DUIM in this project.

Select "Dylan User Interface Manager (DUIM)".

Click **Next**.

Now the wizard offers different pages, which we don't explain here. We will keep clicking *Next* until we get to the last page of the wizard. This is the page for specifying source file headers, as we saw in *The final page in the New Project wizard*.

Click **Next** until the last page of the wizard appears.

If you made any changes to this page last time, they will have been preserved. Whenever you click **Finish**, the wizard saves all these headers (except *Synopsis:*) and some other details, and reinstates them next time you create a project. See *Saving settings in the New Project wizard* for a list of the details that the wizard saves.

Make any changes you want to here, and then click **Finish**.

The wizard creates the new GUI-App project and opens it.

Examining and building the new GUI project

Now, we examine the template code that the wizard has set up for us in the GUI-App project sources.

The GUI-App project contains the same basic set of files as Hello. There is a *library.dylan* file, a *module.dylan* file, and a *GUI-App.dylan* file. In addition, there is a *GUI-App-info.dylan* file and a *frame.dylan* file.

The *GUI-App-info.dylan* file appears whenever you choose "GUI Application (EXE)" as the target type on the first page of the wizard. It contains some simple code that you might want to use for identifying your application and its version number.

The *frame.dylan* file defines a DUIM frame for the application and a set of default menus. Frames are DUIM's way of representing application windows. More knowledge of DUIM is necessary to understand the code in *frame.dylan* properly, but we can start by seeing what the code actually does when we build the project. All projects including template code can be built without requiring any further work.

Choose **Project > Build** in the GUI-App project window.

Choose **Application > Start**.

An application window appears.

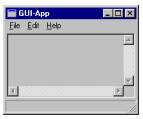

Fig. 6.5: The GUI-App skeleton application.

We can see from the window that the template code creates a skeleton application with File, Edit, and Help menus. There is even some functionality attached to the basic application. If we choose **File > New**, an editor pane is initialized, into which we can type. The other **File** and **Edit** menu commands have their standard effects. The **Help > About** command uses some of the constants from *GUI-App-info.dylan* to identify the application as "GUI-App Version 1.0".

Creating a project using the Custom library option

The New Project wizard's Use Libraries page has a Custom option which allows complete control over the libraries and modules a project will use. This section explains how to choose libraries and modules using this option.

After selecting Custom and clicking *Next*, the wizard shows a page with three list panes. We can make selections from each list pane.

At first, the only list enabled is the Choose Library Groups list. Because there are many libraries available in Open Dylan, the wizard puts libraries into groups according to their functionality. We can select a group to see the list of

libraries it contains, and then choose a library from the list. When we select a group, the wizard displays the library list in the second pane.

Libraries are grouped by functionality in a fairly broad fashion, so some libraries appear in more than one group because they fit more than one description. For instance, the C-FFI library appears in both the "Interoperability" group and the "Win32" group.

Notice the check next to "Core", indicating that "Core" is the only group from which a library or libraries will be used by default. Note that when using Custom library selection to create a project with any GUI or OLE features, you must explicitly specify the GUI and OLE libraries you wish to use.

If we select "Core", we can see which libraries from that group would be used in a default project.

Select "Core" in the Library Group list.

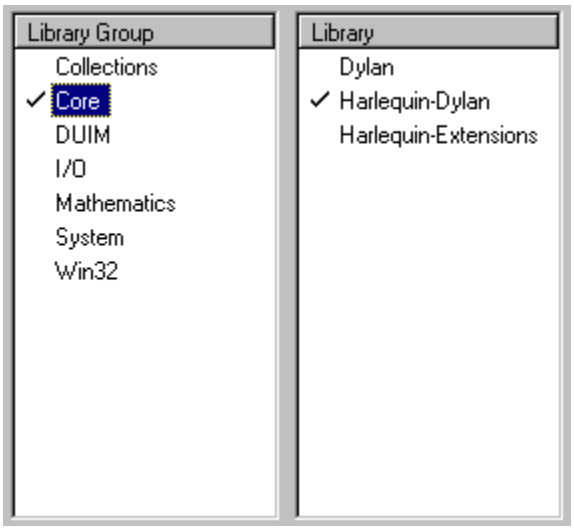

Fig. 6.6: Functional-Dylan is the default library for use in new projects.

So, by default, a project would use the library Functional-Dylan. (Note that Functional-Dylan is the default library for use in new projects. Your copy of Open Dylan may have more library groups.)

If we now select Functional-Dylan in the Library list, we can see which modules from the Functional-Dylan library a default project would include.

Select "Functional-Dylan" in the Library list.

Although the list shows that the Dylan and Functional-Extensions modules are not used, they are actually used indirectly, since the Functional-Dylan module is simply a repackaging of those two modules.

Remember that, in Dylan, the library is the unit of compilation, and modules are simply interfaces to functionality within a library. By deciding not to use a particular exported module, you will import fewer interfaces into your application, but the delivered application will not be any smaller on disk, or in memory when it is running.

Saving settings in the New Project wizard

Whenever you click **Finish** on the last page of the New Project wizard, the wizard stores some of the choices and text-field settings you made so that they are available next time you create a project. The details that are saved persistently are as follows.

- The parent of the folder in the Location box.

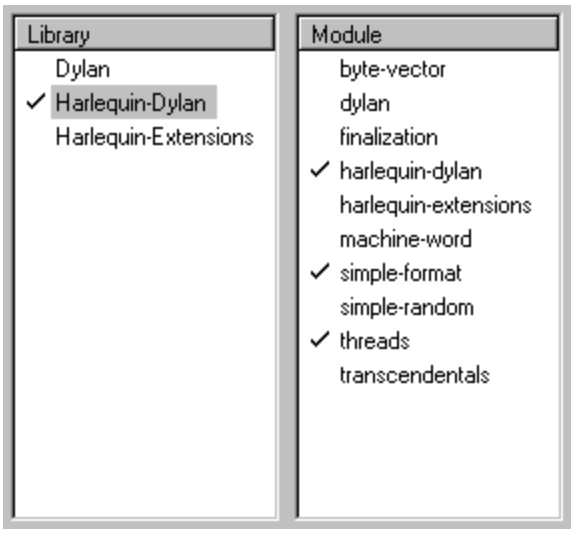

Fig. 6.7: Default modules from Functional-Dylan for use in new projects.

The parent folder is saved in the expectation that you will want to create several projects in sibling folders.

- In the *Advanced...* dialog (see *Advanced project settings*), the contents of the Start Function box and the setting of Compilation Mode.
- The setting of the "Include any available templates" check-box.

On the last wizard page:

- The contents of the Source File Headers boxes, except for *Synopsis:*.

Synopsis: is not saved because it is likely to change with each new project.

Nothing from the first page is saved.

Advanced project settings

The *Advanced...* button on the first page of the New Project wizard leads to the Advanced Project Settings dialog. The dialog has five sections.

The Library and Module Names section allows you to specify names for your project's main library and module. The default value in both cases is the name of the project.

The remaining sections—Start Function, Version Information, Compilation Mode, and Windows Subsystem—all control settings that you can both set here and change after creating a project by choosing **Project > Settings...**. See *Project settings* for details.

Adding, moving, and deleting project sources

In this section we discuss how to insert files into a project, how re-order them, and how to delete them from the project.

Inserting files into a project

To insert a new file or subproject into a project, choose **Project > Insert File...** in the project window. The project window prompts you with the *Insert File into Project* dialog, through which you can find a file to insert.

The file you choose will appear below the currently selected file in the list, unless you insert a subproject (a .HDP file), which will appear at the bottom of the list.

You can insert any file into a project; if the compiler does not know what to do with it, it ignores it. For instance, you can insert .TXT files into a project, and the compiler will skip over them.

When you have chosen your file, the project window places the file below the file currently selected in the list.

If you have added a subproject (a .HDP file), remember that you still need to edit the library and module definitions in your project to import from the new subproject.

Moving the position of a file within a project

To move a file to a new position in a project, select the file in the Sources page and use **Project > Move File Up** and **Project > Move File Down**.

Deleting files from a project

To delete a file from a project, select the file in the Sources page and choose **Project > Remove File**. You could also select **Edit > Cut**, **Edit > Delete**, or the scissors toolbar icon.

Open Dylan asks you if you are sure you want to delete the file from the project, because you cannot undo the operation. Note that the file is not deleted from disk, just removed from the Sources list in the project. You can always put it back with **Project > Insert File**.

Note: The project window's Definitions page shows the definitions that were part of the project when it was last compiled. The list is taken from the current compiler database for the project. If you delete a source file from the project, the definitions from that file stay on the Definitions page until you rebuild the project, which causes the compiler database to be updated.

The project start function

The New Project wizard always adds a *start function* to the end of the last file in the project.

The Dylan language does not require that a program define an explicit start function, such as *main* in C or Java. However, when you are debugging or interacting, Open Dylan finds it useful to know what you consider to be your program's start function. It allows the name of your start function to be recorded in its project information. By default, this name will be *main*, and corresponds to the *main* function that the New Project creates by default in the *project-name .dylan* file for all new projects. However, you are free to change the name if you like—there is nothing special about it.

The *project-name .dylan* file for all new projects will contain a definition of *main* and a call to it. Projects that include template code will contain this definition of *main* :

```
define method main () => ()
  start-template()
end method main;
```

Projects that do not include template code will contain this definition:

```
define method main () => ()
  // Your program starts here...
end method main;
```

For both kinds of project, the *project-name .dylan* file will end with this expression:

```
begin
  main();
end;
```

The name of the Start Function is one of the project settings you can change in the **Project > Settings...** dialog. It appears on the Debug page in the Start Function section. The default name is *main*, but you can change it to any valid Dylan name you like. If you do so, make sure to replace the call to *main* with a call to your new start function. The source file is not updated automatically.

Note that you can make the wizard use a different start function name in new project files by changing the default setting in the Advanced Project Settings dialog. Click **Advanced...** on the second wizard page to produce the dialog. In this case, the generated project code will call the correct new name without requiring you to make a change by hand.

The debugger uses the start function name to know where to pause a program that you start up in interaction mode with **Application > Interact** or the Interact () toolbar button, or in debugging mode with **Application > Debug**. When you start a program either way, the debugger allows the program to execute normally, but sets a breakpoint on the start function so that interaction or debugging begins at a point where the entire program has already been initialized.

If no start function is nominated for a project, the program pauses precisely before it exits but after everything in it has executed. This is usually what we want for a DLL, but not for an application.

Note: To be sure that you can access all the definitions in your application when you start it up in interaction mode, the call to *main* must come after all the definitions in the project. Typically, this means the call must be the last expression in the last file listed in the project. Otherwise, the application will be paused before all its definitions have been initialized, and interactions involving its definitions could behave in unexpected ways. See *Application and library initialization* for more information on this topic.

Project settings

The **Project > Settings...** dialog allows you to set options for compiling, linking, and debugging projects. There are separate pages for each category, each described below.

Compile page

The **Project > Settings...** dialog's Compile page controls the compilation mode setting for the current project. Any project can be compiled in one of two modes: Interactive Development mode, and Production mode. See *Compilation modes* for details of the modes.

Link page

The **Project > Settings...** dialog's Link page controls whether a project is linked as an executable or as a DLL, and what its name will be. It also allows you to specify version information for the target, a base address for it, and the Windows subsystem it runs in.

Note: The default linker used in Open Dylan is a GNU linker. If you own Microsoft Developer Studio, you can use the Microsoft linker instead. To change the default linker, go to the main window and choose **Options > Environment Options...**, then choose that dialog's Build page.

Target File section of the Link page

The **Project > Settings...** dialog's Link page has a Target File section that contains the name of the project target and the type of the target. The default target name is derived from the name of the project. Note that the name will always end in .EXE or .DLL according to the target type, regardless of any extension you give to the target's name.

Base Address section of the Link page

The **Project > Settings...** dialog's Link page has a Base Address section that allows you to specify a base address for your target file. This is the address at which the target will be loaded into memory.

Windows 95, Windows 98, and Windows NT all provide a default base address, one for EXEs and one for DLLs, and will also relocate the target automatically if there is no room for it at that address. You can provide a value in the Base Address if you would like the target to be loaded at a particular location. The value should be specified in hexadecimal, using Dylan's *#x* prefix: for example, *#x1000000*.

Version Information section of the Link page

The **Project > Settings...** dialog's Link page has a Version Information section that allows you to add major and minor version numbers to a DLL or EXE. The values in this section are recorded in the DLL or EXE that the project builds. Open Dylan uses them at compile time and run time to determine if compatible versions of Dylan libraries are in use. See *Versioning* for details.

Windows Subsystem section of the Link page

The **Project > Settings...** dialog's Link page has a Win32 Subsystem section that allows you to specify that the target should run in the "Windows GUI" (WINDOWS) subsystem or the "Windows Console" (CONSOLE) subsystem. You may wish to change this value if you change the code of a console-mode project to make it create its own windows, or vice versa.

The default for a project created in the New Project wizard as a "Console Application (EXE)" is to run in the Windows Console subsystem, while the default for a project created as a "GUI Application (EXE)" is to run in the "Windows GUI" (WINDOWS) subsystem.

Debug page

The **Project > Settings...** dialog's Debug page allows you to specify a command line with which to execute the project target, and the start function for the project.

The command line facility is especially useful for testing console applications from within the development environment. If there are values in the Command Line section of this dialog when you run a project target with *Project > Start* (and similar commands), Open Dylan uses them to execute the application. It creates a new process from the executable named in the Executable field and passes it the arguments from the Arguments field. Thus the values in these fields should form a valid MS-DOS command line when concatenated.

See *The project start function* for details of the start function.

Another use of the Command Line section is to arrange to test and debug DLLs and OLE components. See *Debugging techniques* for a description of these debugging techniques.

Project files and LID files

Open Dylan's project files can be exported in a portable library interface format called LID (library interchange description). Harlequin and other Dylan vendors have chosen LID as the standard interchange format for Dylan libraries. LID files describe libraries in a flat ASCII text format for ease of interchange between different Dylan systems. The *Core Features and Mathematics* reference volume describes the LID format. LID files must have the extension .LID.

Opening a LID file as a project

When you open a LID file in the development environment, it is converted into a project file and opened in a project window. (This process does not modify the original LID file on disk.)

In order to open a LID file as a text file in an editor, open the LID file using **File > Open** and select the file type filter "Dylan Library Interchange Descriptions (as text)" before clicking **Open**.

Exporting a project into a LID file

To export a project as a LID file for use in other Dylan implementations, use **File > Save As** and choose the file type "Dylan Library Interchange Descriptions".

Note that a LID file created by export will list source files by name only, and without paths. In addition it will not contain any of the project settings, any files that are not Dylan source files (.DYLAN files), and any information about whether the project was created as a console application.

CHAPTER

SEVEN

LEARNING MORE ABOUT AN APPLICATION

In this chapter, we examine the browser in detail.

The browser

The Open Dylan browser is a tool for examining the contents and properties of the different kinds of objects we deal with in Open Dylan.

In the browser, the term "object" has a broader sense than is usual in Dylan. Not only can we examine objects in the sense of Dylan class instances, but we can also browse libraries, modules, and even running applications, their threads, and run-time values in those threads.

Similarities between the browser and World Wide Web browsers

The way the browser works is similar to a World Wide Web browser. Just as a web browser shows one page of HTML at a time, the browser describes the properties of one object at a time. And just as an HTML page can contain links to other pages, object descriptions in the browser can refer to other browsable objects. To browse them, you simply double-click on their names.

The browser has a history mechanism just like a web browser, allowing you to move back and forth between objects you have browsed by clicking on Back () and Forward () toolbar buttons.

Compiler databases and the browser

The browser gets some of its information from the compiler database for the project. See *Compiler databases* and *Source, database, and run-time views* for details of how compiler databases are derived, and how they fit in to the overall view of a project that Open Dylan presents.

Browsing a project in source and run-time contexts

The browser allows you to look at projects in both source and run-time contexts. That is, the browser can show you information gathered from the source code representation of a project and also information gathered from a running instantiation of the application or DLL that you have built from that project.

You can look at the static relationships between source code definitions (for example, the superclass and subclass relationships between the classes a project defines) as well as the dynamic properties of run-time values (for example, the value of a local variable in a stack frame).

In the first case, you are looking at information taken from the project's compiler database, and in the second, you are looking at information taken from the application as it runs.

The browser matches run-time objects up with their source code definitions to make as much information as possible available when browsing. For instance, if you select a local variable on the stack in the debugger, and choose the shortcut (right-click) menu's *Browse Type* command, the browser is able to locate the source code definition of the variable's class.

Browsing Reversi

In this section, we use the Open Dylan browser to explore Reversi.

If it is not already open, open the Reversi project.

Go to the Reversi project's Definitions page.

Note that the project window's Definitions page will only contain information if that project has already been built. That is because, like the browser, the project window gets this list of definitions from the project's compiler database.

The Definitions page will be the starting point for our browsing work. We are going to browse definitions in Reversi, and examine the relationships between them.

Expand "library reversi", and then "module reversi".

Double-click on `<reversi-square>` in the list of definitions.

To find this definition easily, you can use the Definitions page's popup list to show only Classes.

The browser appears.

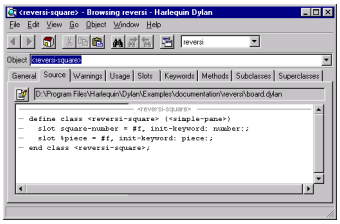

Fig. 7.1: The Open Dylan browser.

The browser arranges information about objects in a set of property pages, with the set of pages available varying according to the type of object being browsed. Though we do not describe all the possible pages here, nearly all have clear meanings. See *List of property pages* for a summary of the pages available in the browser. (Some of these pages also appear in the project window.)

We now take a look at some of the pages available for `<reversi-square>` .

The Source page

The browser provides a Source page for any object that has a definition in source code. For `<reversi-square>`, we can see the source code definition in *board.dylan* that created it.

The Source page shows the definition in a read-only source code pane. If we wanted to edit the definition, we could click the Edit Source (🖉) button above the source code pane. That would open an editor window on *board.dylan*, with the insertion point positioned at the start of the definition of `<reversi-square>`.

For objects that have a representation in the project sources, the Source page will always be the first page that you see when the browser displays the object for the first time.

We now move on to look at some of the other pages of information about `<reversi-square>`.

The General page

The browser provides a General page for every object it can browse. The General page gives an overview of the object being browsed: its name, the type of object it is, the source file containing its definition (if any), and so on.

Select the `<reversi-square>` definition's General page.

The General page for `<reversi-square>` shows that it is a class from the source file *board.dylan*, that it is defined in the *reversi* module of the *reversi* library, and that it has two slots.

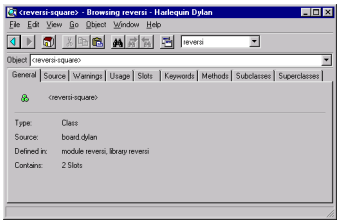

Fig. 7.2: General details about the `<reversi-square>` class definition.

Navigation in the browser

This section explains how to navigate through objects in the browser, and explains the browser history mechanism.

Moving from one object to another

The object information displayed in browser pages often has its own properties and contents that we might also want to browse. With a simple double-click on the information we are interested in, we can move on to browsing other objects.

Go to `<reversi-square>` 's Superclasses page.

The Superclasses page shows a class's superclasses in a tree view. In this case, we see a single expandable item, the class `<simple-pane>`, meaning that `<simple-pane>` is `<reversi-square>` 's only superclass.

If we want to browse the definition of `<simple-pane>`, all we need to do is double-click on it.

Double-click on `<simple-pane>`.

The browser switches to browsing the definition of the class `<simple-pane>`. The default view is again the Source page.

Select the Superclasses page again.

The Superclasses page now shows the four superclasses of `<simple-pane>`. Notice that the superclass names are not directly visible in the current module (*reversi*, as selected in the toolbar pop-up) and so are qualified. For example, `<standard-input-mixin>` appears as:

```
<standard-input-mixin>:duim-sheets-internals:duim-sheets
```

We see more about this in *Namespace issues in the browser*.

We could continue traversing the class hierarchy by double-clicking on a superclass name to browse that class definition in its own right, or, by clicking on the + signs, we could expand the names to reveal their superclasses.

Using the history feature

As soon as the browser has displayed more than one object, its history feature is enabled. You can access the browser history by choosing an object from the Object combo box, or from the *Go* menu. In addition, the Back ( ) and Forward ( ) buttons allow you to navigate the browser history.

Choose **Go > Back** or click on the Back button.

The browser returns to browsing the `<reversi-square>` definition.

Notice that the browser remembers which property page you were browsing.

Browsing a project's library

To browse the current project's library definition, choose **Go > Library** or the Browse Library ( ) button.

Click the Browse Library button.

The browser switches to the Reversi project's library definition. We see the usual General and Source page, as well as Warnings, Usage, Definitions, and Names.

The Warnings and Definitions pages are the same as those that we see in project windows. The Names page provides views of all the Dylan names in the library, with a variety of possible constraints.

The Usage page gives a tree view of the library usage relationships for the current library. The first level of expansion shows the names of the libraries that Reversi uses. Expanding those library names shows the libraries they use, and so on.

Namespace issues in the browser

Move through the browser history to find the `<simple-pane>` object again.

Go to its Superclasses page.

We saw this page in *Moving from one object to another*. There, we said that the special naming format used for the superclasses here meant that they were not part of the *current module* of the *current library*.

To the browser, the current module is whatever module name is selected in the drop-down list box above the Object field (currently *reversi*) and the current library is the library defined by the project.

Change the selected value in the drop-down list to *dylan:dylan*.

The name `<simple-pane>` in the Object list changes to `<simple-pane>:duim-layouts:duim-layouts`.

This new representation of the `<simple-pane>` name says that `<simple-pane>` is found in the *duim-layouts* module of the *duim-layouts* library. This browser uses this special *name* :*module* :*library* naming format whenever *name* is not exported by the current module of the current library.

By changing the list setting to *dylan:dylan*, we told the browser that any name not in the *dylan* module of the *dylan* library should be printed using the special suffix.

The browser's ability to display names from other modules than the current module in an unambiguous fashion is important, because while browsing you may come across names not defined in your library. The browser needs to be able to make it clear when a name is not from the current library and module.

Browsing run-time values of variables and constants

You can browse the values of variables and constants in running applications. The browser shows the run-time value of a variable or constant in its Values page. Simply browse the definition of the variable or constant by double-clicking on it in the project's Definitions page.

The values are shown in a table. Thread variables (variables local to a particular application thread) are shown with an entry for each thread containing a variable of that name. Constants and global variables only ever have one value across all threads, so this is shown as a single table entry entitled "All threads". You can update the value shown in the browser with **View > Refresh**.

We will browse Reversi's **reversi-piece-shape** variable to show how we can monitor the value of a variable while an application is running. That variable stores the current shape of the pieces being used on the Reversi board. By default, Reversi uses circular pieces.

Start the Reversi application.

In the Reversi project window's Definitions page, double-click on the variable **reversi-piece-shape**.

In the browser, choose the Values page.

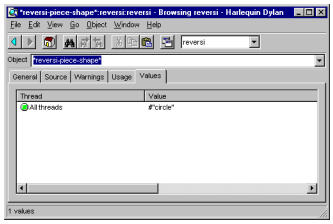

Fig. 7.3: Browsing values in a running application thread.

The value of **reversi-piece-shape** is shown as *#"circle"* for all threads. This is what we expected. Reversi has only a single thread, and we expected some value that would represent a circle.

Choose **Options > Squares** in the running Reversi application.

Choose **View > Refresh** in the browser.

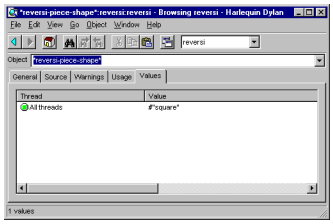

Fig. 7.4: Updated variable value after changing state of application.

The value is now *#"square"*. This reflects the internal change to the variable that our menu selection caused.

Browsing local variables and parameters on the stack

You can browse the contents of the local variables and function parameter values in call frames, as seen in the debugger. These are values on the stack in a paused application thread. Simply double-click on one in the debugger stack pane,

and the browser will display its contents. See *Browsing local variables* for an example of browsing local variable and parameter values.

Browsing paused application threads

Browsing functionality for paused application threads is done in the debugger, not the browser. Think of the debugger as a specialized browser for paused application threads.

However, you can browse a list of current application threads in the browser, along with a text message describing their state, by choosing **Go > Threads** in the project window, editor, or debugger.

If you double-click on a thread name in that list, Open Dylan opens a debugger window on the thread, or, if it already exists, raises the debugger window for the thread, thus pausing the application. Once in the debugger, you can browse the local variables and parameters in call frames in the usual way. See *Browsing run-time values of variables and constants*.

Keeping browser displays up to date

Because the browser shows values either gathered at a point in program execution or during compilation, there are opportunities for the information displayed on a browser page to get out of date:

- If you are browsing a definition and you have edited its source, you must recompile it to ensure that the compiler database is up to date.
- Even when you have recompiled a definition, you must make sure that the browser display is refreshed with **View > Refresh**.
- If you are browsing a run-time object, that object might have since changed. Refresh the browser display with **View > Refresh** to make sure you are seeing the most up-to-date value.

List of property pages

The following is a list of property pages supported in the browser. Some of these pages can also be seen in the project window.

- Breakpoints The breakpoints specified for a project.
- Contents The slot names and values of a run-time instance.
- Definitions The names of the Dylan language definitions created by a module, library or project.
- DLLs The DLLs currently loaded while debugging. You can sort them by version or by file location.
- Elements The keys and elements of a collection instance.
- General The properties of the object (name, type, location).
- Hierarchy The hierarchy of a DUIM sheet, layout or frame.
- Keywords The initialization keywords for a class.
- Libraries The list of libraries in the project.
- Memory The object's memory in the application, shown in bytes.
- Methods The methods of a generic function, or the methods specializing on a class. You can show either methods defined directly on the class or all methods applicable to the class.

- Names The names defined in a module or library. Includes details of from where a name was imported, and whether a name is exported. You can filter to show local names only (that is, names created by the module or library rather than imported from elsewhere), exported names only (which can be both local and imported), or all names (local and imported).
- Slots The slots of a class.
- Source The source code for a source form, with breakpoints shown.
- Sources The source files contained in a project, and their contents.
- Subclasses The subclass hierarchy of a class.
- Superclasses The superclass hierarchy of a class.
- Threads The threads in an application, with priorities, status, and other properties.
- Usage The used and client definitions for a source form.
- Values The run-time values of constants and variables.
- Warnings The compiler warnings associated with a source form.

CHAPTER

EIGHT

DEBUGGING AND INTERACTIVE DEVELOPMENT

In this chapter, we look more closely at Open Dylan's debugger tool.

The debugger

The debugger is a tool for browsing and interacting with a paused application thread. Any thread in an application can be viewed in the debugger (or you can use separate debugger windows for different threads). You can consider the debugger a specialized version of the browser that browses paused application threads.

The debugger provides standard debugging facilities like stepping and breakpoints. It also provides a graphical interface for browsing the state of the control stack in a paused application thread, allowing you to examine local variables and arguments in each stack frame.

You can also use the debugger's interaction pane to interact with a paused application thread. Simply enter Dylan code at a prompt, and the code is executed in the context of the paused thread.

Debugger panes

We now take a look at the basic debugger window panes.

The easiest way to bring up the debugger is to choose **Application > Debug** in the project tool. This starts the application, then pauses its main thread and opens the debugger window on that thread. We can demonstrate this now with Reversi. If you are already running that application, exit it now.

Choose **Application > Debug** in the Reversi project window.

The debugger has four panes: the context pane, the stack pane, the source pane, and the interaction pane.

Context pane

The pane at the top of the debugger is the *context* pane. The context pane gives an overview of the state of the application thread to which the debugger is attached. You can hide the context pane by using **View > Context Window**.

Here, the application stopped because there is a breakpoint on the call to *play-reversi* in *start-reversi.dylan*. That is the function call that starts the application running. The rules for what appears in the context pane are as follows.

If you paused the application yourself, by choosing **Application > Pause**, or by clicking the pause toolbar button (), the message "User requested pause" appears in the context pane of all open debugger windows.

If the application paused because of some other event,

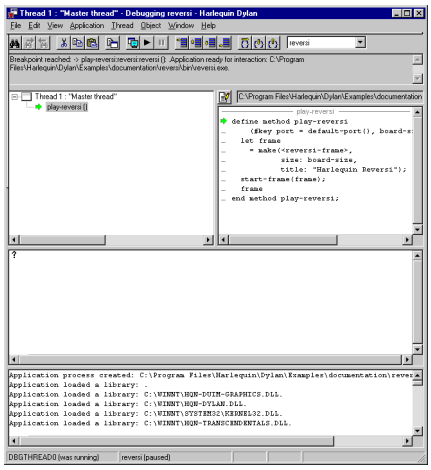

Fig. 8.1: The debugger.

Fig. 8.2: The debugger's context pane.

- the context pane for the debugger on the thread that caused the pause shows a message explaining why that thread paused the application.
- the context pane in all other open debuggers shows the message "Stopped in Thread *x*", where *x* is the name of the thread that has stopped.

Often the message describes an unhandled Dylan error or breakpoint. The message could also describe an out-of-language error (for example in foreign code) or one of a number of application events upon which you can ask the debugger to pause the application, such as when a library is loaded. You can see a list of possible exceptions and the actions that will be taken upon them in the Debugger Options dialog. See *Debugger options*.

Stack pane

The *stack* pane shows the thread's control stack at the moment it was paused, with certain call frames filtered out. It depicts the stack in a tree view. We call this a *stack backtrace* or *backtrace*.

When a debugger window opens, the stack pane opens in expanded form, showing the backtrace details. You can expand and collapse the backtrace in the normal way for a tree view, by clicking + or - symbols. In addition, the menu commands **View > Expand**, **View > Expand All**, **View > Collapse**, and **View > Collapse All** control expansion.

Note: The stack pane may not always contain exactly what you expect to see. First, a function call you expect to see might have been optimized away during compilation. Second, because the stack pane filters out certain call frames by default, a frame you are looking for may only become visible when you change the filtering settings. By default, only frames for Dylan functions defined in or imported into the thread's current module appear in the stack pane. This default setting is indicated by "Filtered visible frames", which appears in the stack pane filtering drop-down list (see *Searching the stack backtrace for the cause of the error* for information about the drop-down list settings).

Choose the **View > Debugger Options** dialog's Stack page in the debugger window to control the filtering rules. See *Stack options* for details.

You can select any item shown in a backtrace and right-click to produce a pop-up menu. The menu allows you to

carry out other operations on them, such as browsing their values or editing their source code representation. Double-clicking an item opens it in the browser.

Thread titles

The root node in the backtrace is the number and title of the thread. Threads are assigned titles using the following scheme:

- Thread *x* : "*name* ".
- A Dylan thread that has a name. The name is a string.
- Thread *x* : Anonymous thread
- A Dylan thread that has no name.
- Thread *x* : Foreign thread
- Any non-Dylan thread.

The initial thread in a Dylan application is always called Thread 1: "Main thread". Each subsequently created thread is assigned an integer number by incrementing the value used for the previous thread, and a title derived according to the scheme above.

In Reversi, there is only one thread, the main thread.

Call frames

Under the root node, at the first level of expansion, the stack pane lists the call frames in the backtrace. The most recently executed call frame is listed first.

Each call frame is represented by the name of the function or method whose call created the frame. If you select a name, the source code associated with it (if any) appears in the pane opposite. See *Source pane* for details of this pane.

Beside each frame name is an icon indicating the sort of call that created the frame:

This was a call to a method selected through run-time dispatch.

This was a direct call, either to a method whose dispatch details were all worked out at compile time, or to a method constant (a function).

This was a call to a foreign (non-Dylan) function. Some of the calls that the Dylan run-time system makes are foreign calls.

This arrow denotes the position of the stack pointer in the thread. It will always be at the top of the list of frames.

When debugging, be aware that some function call frames may be optimized away by, for example, inlining or tail-call optimization. This is particularly important to note if you are compiling in Production mode, where more optimization occurs than in Interactive Development mode.

The debugger also hides certain stack frames concerned with activities like method dispatch, since these are of no interest in debugging user applications.

You can filter other frames out of the backtrace if you wish, according either to their type or to their names. For instance, you can filter out all foreign call frames or all frames whose names contain a particular string. See *Stack options*.

Fig. 8.3: The debugger's stack pane.

Local variables and call parameters

Some call frames shown in the backtrace can be expanded one level further. At this final level of expansion, the pane shows the values of the local variable bindings, including the values that were passed as parameters in the call. Bindings are listed in the order in which they were created, so the parameter bindings appear first.

The bindings, preceded by a yellow star icon (), are shown in the form:

```
*name* = *value*
```

Where *value* is displayed in a summarizing notation, which defaults to the value's class enclosed in curly braces if there is no simple printed representation. For example, basic numeric types, strings, booleans, and sequences thereof can be printed literally, but an instance of `<reversi-frame>` cannot.

Source pane

The debugger's source pane shows a source code definition for the method or function that created the call frame that is currently selected in the stack pane.

A text field above the source pane shows the location on disk of the source file containing the definition. If you click the Edit Source () button, Open Dylan opens the source file in an editor window, with the insertion point placed at the start of the definition.

If the debugger cannot locate the source code for the method or function that created the selected call frame, or if you select a local variable or the backtrace's root node, the debugger leaves the source pane empty, writes "No source available" in the field above the source pane, and makes the Edit Source button unavailable.

The source pane shows the same green arrow () seen in the stack pane. This *current location* arrow shows the point to which execution within that call frame had proceeded before the application was paused. (Thus the current location arrow that you see in the source pane for the top-most call frame in the stack pane is the most accurate depiction of the point at which execution was paused.)

Often, the arrow is located at a point where a function has called another function that has not yet returned, or at the point at which execution will resume when a function returns.

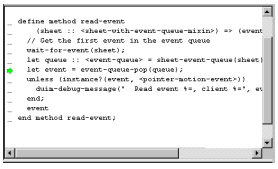

Fig. 8.4: The current location arrow in the debugger's source pane.

The line at which the arrow is located is not always precisely correct, because optimizations carried out by the compiler may have produced object code that does not correspond to the source code in a way that can be easily depicted in this pane. The chance of imprecision is increased when compiling a project in Production mode.

In our example, the arrow appears at the start of the definition of the *play-reversi* method. This shows that execution paused at the beginning of the execution of *play-reversi*.

Interaction pane

The interaction pane allows you to enter Dylan expressions for evaluation in the context of the paused thread. (In some language implementations, an interaction tool like this is called a *listener*.)

See *Interacting with an application* for a discussion of interaction.

The interaction pane also logs thread events that the debugger noticed (such as DLL loading) and debugging messages explicitly output by the thread (for example via the Dylan library's *debug-message* function in the `simple-debugging` module).

The pane does not collect ordinary output from the application being debugged. Such output continues to go to its ordinary destination, such as the standard output.

Keeping debugger windows up to date

It can be costly to keep all the panes of open debugger windows up to date. For this reason, Open Dylan only refreshes the information in debugger windows in certain circumstances.

Note: In the following descriptions, a window being "refreshed" can also mean the window being displayed for the first time, with up-to-date contents, if it has not yet been displayed.

- If an application thread pauses and a debugger window appears (or comes to the foreground), the debugger window for that thread is refreshed automatically.

Threads pause when: there is an unhandled Dylan error in the thread, there was an out-of-language error in the thread, execution in the thread reached a breakpoint or step point, or the thread reached the end of an interactive evaluation.

- If you issue the pause command (**Application > Pause** or
- If you issue the pause command from the project window or editor, the debugger for the application's main thread is refreshed.

In all of these circumstances, any other open debugger windows will not be refreshed unless you choose **View > Refresh** in them. If you choose **View > Refresh All Debuggers**, Open Dylan updates every open debugger window.

The Misc page of the debugger's **View > Debugger Options** dialog contains the option "Refresh all open debuggers when entering debugger". By default, this option is not set. When turned on, the rules above are ignored and every open debugger window is refreshed whenever the application pauses.

Controlling execution

The **Application** menu, shared by the debugger, editor, and project windows, contains a set of commands for controlling the execution of an application or DLL. Some toolbar buttons provide shortcuts to these commands.

In a project window, the commands on the **Application** menu act upon the executable application (.EXE file) or DLL that was last built for that project. The command selects the application or DLL depending on the target file type setting on the *Project > Settings* dialog's Link page.

In a debugger window, the commands act upon the application that created the thread being debugged. In an editor window, the commands act upon the application of the active project. (Note that the **Application** menu is not available in the editor if the source file being edited is not part of the active project—the project whose name is visible in the main window's drop-down list. See *The active project* for more details.)

Starting and stopping applications

Application > Start (or the toolbar start/resume button ()) executes the application with which the window is associated.

After you have started executing an application, the **Application > Start** command is never available again until you stop the application with **Application > Stop**, or until the application terminates normally.

Application > Stop (or the toolbar stop button ()) terminates the process of the application with which the window is associated. Before terminating the process, Open Dylan asks you to confirm that you want to do so. This helps reduce the chance of an accidental termination that loses valuable application state.

After you have stopped an application in this way, you can start it again with **Application > Start**.

Pausing and resuming execution of applications

Application > Pause (or the toolbar pause button ()) pauses the execution of the application with which the window is associated.

When an application is paused, you can browse and debug its threads or interact with it. Choose **Application > Resume** (or the toolbar start/resume button ()) to resume execution.

You should normally only use **Application > Resume** when the application stopped because you paused it or it reached a breakpoint (both of which are out-of-language events, that is, events not described completely in terms of the Dylan language). If the application stopped because of an unhandled condition or a call to *break* (both in-language events), you should instead use the items on the **Thread** menu to signal a Dylan restart. See *Restarts* for information on the **Thread** menu.

If you use **Application > Resume** to continue from an in-language event, your application may signal further errors because you did not use the (in-language) restart mechanism to deal with the existing error.

Any Dylan restarts which were available before you resumed the application should still be available, so you can continue by signalling a restart as before. See *Restarts* for more details.

Restarting applications

Application > Restart restarts the application with which the window is associated. There is no toolbar shortcut button for this command.

This command is only available if the application is already running. Since restarting an application logically consists of stopping it and starting it again, choosing this command is equivalent to choosing **Application > Stop**, then **Application > Start**.

Interacting with applications

Application > Interact (or the toolbar interact button ()) pauses the execution of the application with which the window is associated and opens a debugger window on it. The behavior is exactly the same as **Application > Debug** (see *Debugging techniques*) except the stack and source panes of the debugger window are hidden.

See *Interacting with an application* for a discussion of interaction.

Debugging techniques

Because of the different characteristics of executable (EXE) files, DLLs, and OLE components, in each case there is a slightly different technique for invoking the debugger. This section covers these techniques. Debugging a client/server application is discussed in *Debugging client/server applications*.

Debugging executables

Use **Application > Debug** (or the toolbar debug button ()) and **Application > Interact** (or the toolbar interact button ()) to debug an executable (EXE) application.

These commands start the executable associated with the window, then pause its main thread and open a debugger window on that thread. If the application is already running, these commands pause the application in its current state.

If you want to start an application up in the debugger, so that you can examine its initial state, you want the application's library and the libraries it uses to initialize completely before the debugger pauses it. To do this, you need to specify the application's *start function*. A start function is a function that the application calls upon startup to set things running, such as a call to start an event loop in a windowing application. See *The project start function* for details.

You can specify a start function on the Debug page of the **Project > Settings...** dialog. When you use **Application > Debug** or **Application > Interact**, the environment places a temporary breakpoint on the start function so that the application starts and then enters the debugger on entry to the start function. For this reason, the expression that calls the start function should appear after all definitions in the project, so that all definitions will be accessible in the debugger.

If the project does not specify a start function, the application will enter the debugger after all expressions are executed and the main thread is about to exit. In this case, the debugger is entered as the application has finished, which is not normally very useful. If you always specify a start function, you can pause the application at a more useful point.

Debugging DLLs

Debugging DLLs is similar to debugging executable (EXE) applications (see *Debugging executables* above), but there are a couple of differences.

One difference is that DLLs may not have a start function. Without a start function, there is nothing for the environment to place a breakpoint upon in order to pause the DLL's execution and enter the debugger.

Nonetheless, for debugging it is still useful to be able to pause the DLL once it has initialized completely but before it exits. To do this, simply remove the name in the Start Function section of the **Project > Settings...** dialog's Debug page. Then, when you choose **Application > Debug** or **Application > Interact**, the debugger lets the DLL execute all its top-level expressions, and pauses the DLL just as its main thread is about to exit. This gives you access to all the definitions and state that the DLL creates.

If the DLL does have a start function, simply specify it and the **Application > Debug** and **Application > Interact** commands will work as they do for EXE applications: the environment adds a breakpoint on the start function, and on entry to the function the DLL's main thread is paused and a debugger window is opened on it.

Another consideration is that it is not normally possible to execute a DLL directly; instead, you start an EXE that calls it. The environment normally handles this issue for you, by using a small EXE that takes the target DLL name as a command-line argument, loads it (causing all its top-level expressions to be executed), and exits. When you choose **Application > Start**, **Application > Debug**, or **Application > Interact** the environment runs the EXE. The point at which the EXE is paused again depends on whether you supply a start function.

Alternatively, you can supply your own EXE in the Executable field of the **Project > Settings...** dialog's Debug page. The **Application > Start**, **Application > Debug**, and **Application > Interact** commands then call your EXE and behave in the same way as if your EXE was the project target file. That is to say, execution of the EXE proceeds without intervention from the debugger until the DLL loads. Only then will the debugger be in a position to pause the DLL. (Again, the point at which the pause occurs depends on whether you specify a start function.)

Debugging OLE components

To debug in-process OLE servers and OLE controls, which must be built as DLLs, you can use the same debugging processes as described in *Debugging DLLs*.

If you want to test your server or control in a container application, simply enter the name of the application executable in the Executable field of the **Project > Settings...** dialog's Debug page. This executable could be any OLE container, such as WordPad. The **Application > Start**, **Application > Debug**, and **Application > Interact** commands then execute the container executable. Execution proceeds normally until the code of the OLE server or control is executed; only then will the debugger be able to intervene either by pausing when the server or control loads or when a start function is called.

In the case of an OLE compound document server, of course, the debugger will only be able to act if you choose to insert an instance of your OLE server object into your test container application.

Restarts

The debugger provides a way to signal any restart for which a handler is available at a given point in application execution. Restarts are part of the Dylan language's condition system, and are explained in chapter 7 of the DRM.

You can use the debugger to signal a restart if your application has entered the debugger due to a condition having been signalled but not handled, or due to it reaching a breakpoint. You cannot do so if the application has paused because you used **Application > Pause**. (Use **Application > Resume** to restart your application in that case.)

To select a restart to be signalled, choose **Thread > Continue...**, which displays a dialog listing all available restarts.

For convenience, there are two other menu items for signalling `<abort>` restarts, which are defined to mean "terminate the current computation".

Thread > Abort signals the innermost available `<abort>` restart—that is, it aborts as little as possible—whereas **Thread > Abort All** signals the outermost `<abort>` restart—that is, it aborts as much as possible.

Although the meaning of `<abort>` restarts is part of the Dylan language, your application must provide handlers to implement them. If you are using DUIM for your application's GUI, note that DUIM frames normally provide `<abort>` handlers in the event loop, so that aborting while processing an event will proceed to process the next event. See the DUIM documentation for *start-frame* and *start-dialog*.

Choosing an application thread to debug

As we noted earlier, each application thread can have its own debugger or you can use one debugger window to view various threads one at a time. The command we have seen so far, **Application > Debug**, debugs only the application's main thread. To debug another thread in the application, choose **Go > Threads** from the debugger, project window,

or editor. From the debugger window you can also use **Thread > Select Thread...**. To bring up multiple debugger windows, use **Window > New Window** from an existing debugger.

The **Go > Threads** command launches a browser on the application itself, treating it as an object consisting of one or more threads whose states are visible in a table. If you browse a particular thread, Open Dylan refreshes the existing debugger window to display the thread or opens a debugger window if none already exists.

You can then debug a thread from the table by double clicking, or by using the right-click popup menu. This action stops the thread if it is running, and opens a debugger window on it.

Changing the debugger layout

Open Dylan lays out the debugger window to suit the situation. If you choose **Application > Interact**, the debugger uses its interaction layout. In this layout, the interaction pane is maximized and the stack and source panes do not appear at all. This layout hosts interactive sessions and is sometimes casually referred to as "the interactor".

If the debugger was invoked because of an error or because you chose **Application > Debug** or **Application > Pause**, the interaction pane will be a small pane below the stack and source panes.

You can change the automatic layout using **View > Interacting Layout** and **View > Debugging Layout**. You can also hide or show the context window using **View > Context Window**.

Interacting with an application

Open Dylan allows you to interact with your applications. Interaction consists of executing Dylan expressions and definitions in the context of a paused application thread. Open Dylan also offers the *Dylan playground*, a facility for interactive Dylan programming experiments outside the context of application development.

To explain the things you can do interactively, this section includes two examples. One uses the Dylan playground and the other uses the Reversi application. First, however, we discuss the interaction pane, a debugger pane that hosts interactive sessions.

About the interaction pane

The debugger's interaction pane provides a prompt (where you can enter Dylan expressions and definitions for execution. The prompt is a question mark (*?*).

The interaction pane is similar to what some other languages call a *listener* tool, and it provides the "read-eval-print" model of interaction that is standard in those tools. However, in Open Dylan interactions, the "eval" phase is not really evaluation. It consists of compiling your code and then sending the compiled code to the paused application thread itself, where it is executed, modifying the state of the thread accordingly. This means that you can interactively add features to an application and even redefine parts of it, all while the application is still running.

The size of the interaction pane differs according to the situation. See *Changing the debugger layout* for details of the different layouts and how to change them.

Starting an interactive session with an application

The simplest way to start interacting with an application is to choose **Application > Interact** or click the interact toolbar button (). This starts the application if necessary, and then pauses it.

If the application was started afresh, the pause occurs at the same point as it would with **Application > Debug** (see *Debugging executables*). Otherwise the application is paused in its current state. A debugger window then opens on the paused thread.

Interaction basics using the Dylan playground

The Dylan playground allows you to carry out interactive Dylan programming experiments. The playground is a pre-built Dylan application that you can start from the main window using the Open Playground button (), or with the menu command **Tools > Open Playground** from any Open Dylan window. The playground has its own project, which also opens when you start it.

Start the playground with **Tools > Open Playground** in any open window.

Upon opening the playground, its project window appears. Then the playground application starts automatically and enters the debugger. The debugger window has a large interaction pane, and no visible stack or source panes. This is the debugger's *interaction layout*. (We can change the layout to the normal debugging layout with **View > Debugging Layout**.)

One of the simplest things we can do in the interaction pane is to use it as a desktop calculator.

Enter *56 - 24;* at the interaction pane prompt.

Make sure to include the terminating semi-colon, and to include spaces between the numbers and the - sign:

```
? 56 - 24;
=> $0 = 32
?
```

Here, text entered after the *?* represents interaction pane input, and text after the => represents interaction pane output.

Any compilation warnings resulting from typing errors are displayed in the interaction pane itself.

The interaction pane offers a history facility which allows us to refer to previous interaction results. Each value returned by an interactive expression is bound to a name, which we can then use in subsequent expressions. We call these bindings history variables. They are named using a dollar sign ($) suffixed with an integer. To keep the history variable names unique, the integer suffix increments each time a new history variable is created. So far, our one result was assigned to the history variable *$0*.

We can add the value bound to *$0* to itself.

Enter *$0 + $0;* at the interaction pane prompt:

```
? $0 + $0;
=> $1 = 64
?
```

This expression produces the expected result of 64 and creates a new history variable, *$1*, bound to that result.

Note: History variable values are local to the debugger in which they were created, so you cannot refer to a history variable from any other debugger's interaction pane.

We can define new classes and methods interactively simply by entering their definitions.

Enter the following definition of `<my-class>` at the interaction pane prompt.

```
define class <my-class> (<object>)
  slot my-slot :: <integer>
end class <my-class>;
```

Note: You can hit Return to format your input in multi-line form where you prefer. (An expression is only evaluated when you hit Return after a semicolon.)

The output in the interaction pane is:

```
? define class <my-class> (<object>)
slot my-slot :: <integer>
end class <my-class>;
=> No values
```

Enter *<my-class>;* at the interaction pane prompt:

```
? <my-class>;
=> $2 = {<class>: <my-class>}
?
```

Return values in the interactor are "live". You can use the shortcut (right-click) menu to perform a variety of useful operations on them.

The **Show Contents** command allows you to browse the contents of values within the interaction pane itself. What you see depends on the type of the value; with a class, each slot name and slot value is listed. Each slot value is bound to new history variable so you can refer to it in future interactive expressions.

Right-click over *$2 = {<class>: <my-class>}* and choose **Show Contents**:

```
=> $2 = {<class>: <my-class>}
? Contents of {<class>: <my-class>}
=> {<class>: <my-class>} is a <class>
$3 = instance?-iep : '\\<-49>'
$4 = debug-name : "<my-class>"
$5 = class-implementation-class : {<implementation-class>}
$6 = class-subtype-bit : 0
$7 = class-module : {<module>}
?
```

For the duration of the interactive session with a project, interactively created definitions, objects, and any resultant warnings are layered onto the project's compiler database. During an interactive session, these items will be available in the project window and browser. You can think of them as being like any definition or object, with the exception that they do not come from a source file.

Enter *define variable *obj* = make(<my-class>);* at the interaction pane prompt.

In the playground's project window, go to the Definitions page.

Expand *library dylan-playground* and then *module dylan-playground*.

Three definitions are listed under *module dylan-playground* : one for **obj**, one for `<my-class>`, one for the *my-slot* accessor, one for its getter, and one for the method *main*. Thus we see two interactively created definitions alongside one definition created at compile time.

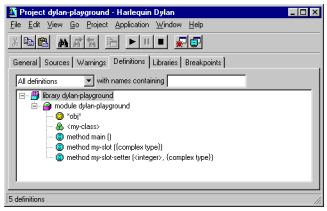

Fig. 8.5: Interactively created definitions alongside a compile-time definition.

We can also use the interactor to make a GUI window by using the Dylan User Interface Manager (DUIM) capabilities. For example:

At the interaction pane prompt, enter:

```
contain(make(<push-button>,
             label: "Hello World! This is my DUIM window."));
```

The code is compiled and run and a DUIM window opens.

Fig. 8.6: Window created interactively with the Dylan User Interface Manager.

For more information about creating GUI interfaces with Open Dylan, see the *Building Applications Using DUIM* and *DUIM Reference* manuals.

An example interaction with Reversi

In the following example we interact with the Reversi application after first making a few moves.

Open the Reversi project, and choose **Application > Start**.

If you were following the **Application > Debug** example earlier in this chapter, you could simply choose **Application > Resume** instead of starting Reversi again.

Make some moves on the board.

Choose **Application > Interact**.

A debugger window appears, in the interaction layout of a large interaction pane and no stack or source panes.

Now we are ready to write Dylan code interactively. Note that because **Application > Interact** pauses the application, we are not interacting with a running application. The only way the state of the application changes is through our interactions. So we must choose **Application > Resume** to see interactive changes in effect.

As an example, we can change the shape of the pieces on the board by setting the value of the variable **reversi-piece-shape** instead of by going to Reversi's **Options** menu.

Enter **reversi-piece-shape* := #"triangle";* at the interaction pane prompt.

Choose **Application > Resume**.

Make a few moves on the Reversi board.

The new moves are shown in triangular pieces, as well as any previous moves that have repainted after being obscured by other windows. To see only triangular pieces, minimize and restore the board or resize it.

Interactive access to local variables and arguments on the stack

Interactive expressions can refer to variables from the debugger's current stack backtrace, simply by using their name. Before referring to a variable, you must select the stack frame that contains the variables you want to use. This is so the debugger can resolve any ambiguities arising when local variables in different stack frames have the same name.

For example, with this backtrace:

```
[-] go()
    x = 4
    y = 5
[-] set()
    x = 30
    y = 2
[-] ready()
```

If you select the stack frame for the call to *go*, evaluating the expression *x* + *y* yields the result 9. But if you select the stack frame for the call to *set*, evaluating the expression *x* + *y* yields the result 32.

Effects of interactive changes to application threads

As stated in *About the interaction pane*, because the interaction pane compiles and executes the code you enter in the context of the paused thread to which the debugger is connected, it is possible to define new bindings, redefine existing bindings, and get and set values in an application. You can then resume execution to test your changes.

The level of optimization that occurred when the application was originally built does, however, affect the kinds of things you can do interactively. It is best to compile a project in Interactive Development mode if you want to define and redefine classes and methods interactively. Even in that mode you may encounter some restrictions, particularly when trying to make changes in system libraries.

As noted in *An example interaction with Reversi*, the results of compiling interactive changes to threads are added to a temporary layer of the compiler database for the application's project. This allows you to browse the effects of your changes while the application remains running, but these changes are not saved to disk in the compiler database file, nor are they saved in the project source code files. When you stop or close an application, Open Dylan removes the interactive layer automatically. (This is one of the reasons why you are asked to confirm when you choose **Application > Stop**.)

Interaction pane commands

The debugger's interaction pane accepts various commands. You can get a list of available commands, and documentation for each command, with the *:help* command.

:help *Interaction command*

```
:help
:help *command-name*
```

The first form prints a list of interaction commands in the interaction pane. The second form prints documentation on a command from the list.

:in *Interaction command*

```
:in *module-name* *[* :*library-name* *]*
```

Sets the context in the interaction pane. The current library is the default if not otherwise specified. Using this command is equivalent to using the context drop-down list on the debugger/interactor toolbar.

The active project

All tools in Open Dylan need to know with which project they are associated. A project window is, naturally, always associated with the project it describes. A browser window is associated with the project from which the object it is currently browsing came. A debugger window is associated with the project of the application to which it is connected.

These associations are fairly natural, but the situation for editor windows is slightly more complicated. An editor window can be editing a source file that is part of more than one open project. If we choose **Project > Build** in the editor window of a source file that appears in two open projects, how does Open Dylan know which project to rebuild?

The answer is that the editor has an *active project*. This is the project whose name is visible in the drop-down list in the main window. You can change the active project by changing the setting in the list. The active project is always one of the projects that have been explicitly opened—that is, one for which there is a project window.

The editor also uses the active project to determine two other things. First, the editor shows breakpoint information in its leftmost column for source files in the active project **only**. Second, the **Project**, **Build**, and **Application** menus are disabled in editor windows on source files that are not part of the active project.

A project can become the active project automatically as well as by being set explicitly in the main window's drop-down list. The main window's **Options > Environment Options...** dialog has two options on the General page controlling when projects become active automatically: "Project becomes active when opened" and "Project becomes active when application started".

Breakpoints

Open Dylan allows you to set breakpoints on application code from within any window. Breakpoints allow you to pause an application at a predefined point in execution, in order to examine it in a debugger window. You can set breakpoints on Dylan code lines in a source code file or on suitable generic functions, methods, and functions.

Non-pausing breakpoints are also available. These breakpoints do not pause the thread when execution reaches them, but simply log a message in the debugger interaction pane to say they have been reached. To distinguish breakpoints that pause the application from non-pausing breakpoints, we sometimes call them *pausing* breakpoints.

You can set, disable, and clear breakpoints from any Open Dylan tool that has an **Application** menu. In addition, you can use the shortcut (right-click) menu to do the same on any selected method in the project window or browser. Finally, you can set breakpoints on lines of code in the editor and on the browser Source page.

It is possible for the same code to be shown with different sets of breakpoints in different contexts. When looking at source in the browser or debugger, the set of breakpoints shown is that for the project being browsed. Within the editor, the set of breakpoints shown is that for the active project.

How breakpoints work

A breakpoint forces a pause in application execution, which we call a *break*. When a thread within an application attempts to execute an item of code that has a breakpoint on it, Open Dylan pauses the application and opens a debugger window on the thread that reached the breakpoint.

There are also non-pausing breakpoints that print a message in the debugger's interaction pane. Generally, when we refer to breakpoints, we mean the pausing kind.

Breakpoints can be either enabled or disabled. Enabled breakpoints are shown with a solid red octagon icon, while disabled breakpoints are shown with a hollow red octagon.

When you first set a breakpoint, it is enabled. You can disable a breakpoint if you do not want it to cause a break next time you run the application. If you decide that you never want the application to break at that point again, you can clear the breakpoint to remove it completely.

When Open Dylan encounters a breakpoint, it prints the breakpoint location in the debugger's context pane, and also (if the breakpoint option *Print Message* is checked) adds it to the debugger's interaction pane. See *Breakpoint options*.

Breakpoints are associated with the application's project, rather than the compiled application itself. This means breakpoints only have an effect if the application is executing under debugger control within the development environment, via **Application > Start** and similar commands.

When you set a breakpoint in a function that is in a used library, the breakpoint does not go into the project that owns the function, but instead into the project you are browsing—or the active project, if in the editor.

Setting breakpoints on functions

You can set breakpoints on generic functions and their methods. The application will break to the debugger on entry to the function.

To set a breakpoint on an individual method, select it on either the project window Definitions or Sources page, or in the browser's Methods page. Bring up the shortcut (right-click) menu and choose **Set Breakpoint**.

The simplest way to set a breakpoint on all the methods of a generic function is to choose **Application > New Breakpoint**, and enter the name of the generic function in the dialog that appears. The application will break to the debugger whenever any method on that generic function is called.

Setting breakpoints on lines of code

You can set breakpoints on lines of code in source code files. The application will break to the debugger when it executes that line of code, or, depending on the way the code has been compiled, as near as possible to that line.

You can set a breakpoint on a line of code in any tool that can show you it. The editor is the obvious tool to use, but you can also breakpoint lines of code that you can see in the debugger's source pane or the browser's Source page.

To set a breakpoint in any of these situations, click on the leftmost column of the line you wish to breakpoint. You can only do this for lines showing an underscore character in the leftmost column.

When you set the breakpoint, a solid red circle appears to show that it is an enabled pausing breakpoint. You can toggle the breakpoint between enabled and disabled by clicking on the circle. You can also use the shortcut (right-click) menu to manipulate breakpoints on lines of code. See *Breakpoint commands on the shortcut menu*.

Browsing a project's breakpoints

You can see all of a project's breakpoints in the project window's Breakpoints page. This shows the location of the breakpoint (the name of the function or the line of the source code file),

You can also manipulate any breakpoint here by selecting it and using the commands on the shortcut (right-click) menu. See *Breakpoint commands on the shortcut menu*.

Breakpoint commands on the shortcut menu

In some situations you can right-click to produce the shortcut menu, which contains several breakpoint manipulation commands. These are:

- When you are browsing the project breakpoints in the project window's Breakpoints page.
- When you have selected a method in the project window's Definitions or Sources page.
- When you are browsing a generic function and have selected a method in the browser's Methods page.
- When your mouse pointer is over a breakpoint icon in the leftmost column of either the editor window, the debugger source pane, or the browser Source page.
- When your mouse pointer is over an underscore in the leftmost column of either the editor window, the debugger source pane, or the browser Source page.

Underscores show lines where you could add a breakpoint.

The commands available on the shortcut menu depend on the context. The complete list of commands follows.

"Trace" Sets a trace point for the selected function. When you set a trace point for a function and then run and pause the application, the nesting levels of recursive calls and their subsequent output are printed to the debugger's interaction pane. This allows you to see the values of the function's arguments and the associated result values.

"Untrace" Removes the trace point for the selected function.

"Untrace All" Removes all trace points for the current project.

"Run to Cursor" Only available in the debugger. Sets a temporary pausing breakpoint at the line the mouse pointer is on, then starts the application or resumes the application if it was paused. The application runs until that line is reached, at which point the application enters the debugger and the breakpoint is cleared.

Temporary breakpoints are denoted by a solid green circle.

"Set Breakpoint" Sets an enabled pausing breakpoint at the line the mouse pointer is on.

"Clear Breakpoint" Removes any breakpoint at the line the mouse pointer is on.

Edit Breakpoint Options... Pops up a dialog for editing breakpoint options. See *Breakpoint options*.

The dialog appears even if a breakpoint did not exist on the function or line.

Breakpoint Enabled? A toggle for enabling and disabling the breakpoint at the line where the mouse pointer is.

Breakpoint commands on the Application and Go menus

The **Application** and **Go** menus available in the project window, editor, and debugger contains several breakpoint manipulation commands.

The **Go > Breakpoints** command, chosen from the project window, shows the Breakpoints page. Chosen from the debugger, it raises the project window for the application being debugged and shows its Breakpoints page. Chosen from the editor, it raises the project window for the active project (see *The project start function*) and shows its Breakpoints page.

The **Application > New Breakpoint** command sets a breakpoint on a generic function (and all its methods) or a non-generic function. It produces a dialog into which you enter the name you wish to breakpoint.

The **Application > Enable All Breakpoints** command sets enables all disabled breakpoints. **Application > Disable All Breakpoints** disables all enabled breakpoints.

Breakpoint options

The **Edit Breakpoint Options...** dialog, available from the shortcut (right-click) menu, contains the following sections.

- "Enabled" Check item for toggling whether a breakpoint is enabled or disabled. A disabled breakpoint does not affect the application's execution. New breakpoints are enabled by default.
- "Pause application"
- Check item for toggling whether the breakpoint pauses the application when it is encountered.
- New breakpoints pause the application by default.
- If you turn pausing off, the non-pausing breakpoint simply logs a message in the debugger interaction pane whenever it was reached. This kind of breakpoint is shown with a solid yellow triangle when enabled and a hollow yellow triangle when disabled.

- "Print message"
- Check box for toggling whether the breakpoint prints any message in the debugger's interaction pane when it is encountered. By default a new breakpoint does print a message.
- "Message text" Text field for entering some identifying message to be associated with the breakpoint (if any). The text is used in debugger messages referring to the breakpoint. This field is not available if *Print message* is not checked.
- "One shot" Check box for toggling whether the breakpoint is temporary or permanent. Temporary breakpoints are removed after they have been encountered. By default, new breakpoints are permanent.
- The shortcut (right-click) menu's *Run to Cursor* command creates temporary breakpoints.

Stepping

After pausing an application, the debugger allows you to continue its execution in small steps, after which control returns to the debugger. There are three stepping commands: Step Over, Step Out, and Step Into.

Each command makes all application threads begin executing again. The application executes until the thread belonging to the debugger that issued the stepping command reaches the destination of the "step". At that point, all threads pause and control returns to the debugger.

The steps relate to functions on the control stack for a particular thread. The steps are defined at the level of source code, not object code. This means that stepping operations in an application that was compiled in Production mode can sometimes work in unexpected ways, because of optimizations carried out by the compiler.

To step through in a particular application thread, issue the stepping command in the debugger on that thread. The commands are available on the debugger's **Thread** menu as well as on toolbar buttons.

The following sections give examples to illustrate what the three stepping commands do.

Step over

Choosing **Thread > Step Over** in a debugger "steps over" the next function call that occurs in that debugger's thread, executing the call in full and then returning control to the debugger. The command operates in the context of the currently selected call frame in the debugger's stack pane.

Consider this stack backtrace:

```
[-] Thread 1: "Main thread"
|image13| [+] concerto
[+] opus
```

The selected frame is *concerto*, the source code for which looks like this:

```
define method concerto () => ()
  first-movement(#"moderato");
  |image14| second-movement(#"adagio-sostenuto");
  third-movement(#"allegro-scherzando");
end method;
```

Where execution was paused in the call to *second-movement*. Choosing **Thread > Step Over** runs through the entire execution of *second-movement* before returning control to the debugger.

Thread > Step Over does an implied **Thread > Step Out** too, so that if when you choose **Thread > Step Over** there is no more code, it steps out rather than continuing the application no longer under the debugger. See *Step out* for details of **Thread > Step Out**.

Step into

Choosing **Thread > Step Into** in a debugger "steps into" the next function call that occurs in that debugger's thread, and then returns control to the debugger before the function begins to execute. This command is not sensitive to the debugger's selected call frame.

Typically, this command causes a new frame to appear at the top of the stack.

Thread > Step Into does an implied **Thread > Step Over** (and hence an implied **Thread > Step Out**), so that if you when choose **Thread > Step Into** and there is nothing to step into, it acts like a **Thread > Step Over** (or a **Thread > Step Out** once you leave the function). See *Step over* for details of **Thread > Step Over** and *Step out* for details of **Thread > Step Out**.

Step out

Choosing **Thread > Step Out** in a debugger "steps out" of the current function call, that is, it resumes execution of the application until a function returns, and then passes control back to the debugger.

This command is sensitive to the debugger's selected call frame: it always steps out of the function running in that frame.

Consider this stack backtrace:

```
[-] Thread 1: "Main thread"
[+] -- presto ()
[+] -- allegro ()
[+] -- moderato ()
|image15| [+] -- andante () <<<<
[+] -- adagietto ()
[+] -- adagio ()
[+] -- largo ()
```

The selected frame is *andante*. Choosing **Thread > Step Out** resumes execution of the thread until *andante* returns.

Debugging client/server applications

If you have a client/server application, where both the client application and server application are written in Dylan, you can debug them in parallel.

Start by opening both projects in the environment. It is not possible to run two instances of the environment, with one debugging the client and the other debugging the server: if any libraries are shared between the applications, both environment instances will attempt to lock the compiler database files for those libraries. Since all applications ultimately use the Dylan library, and most share other libraries—not the least of which in this case being networking libraries—using two Open Dylan processes is never a practical debugging method.

This is not a disadvantage. By running both client and server in one Open Dylan, you can be debugging in the client, and then when the client invokes the server you can smoothly start debugging that instead. This can be very useful for tracking down synchronization bugs.

Once you have both projects open, you can start both applications up. Note that by default the action of starting a project will switch the active project, so the last project you start will be the active one by default. You can change this behavior in the main window with **Options > Environment Options...** so that the active project does not switch in this situation. See *The active project* for more information.

If you need to rebuild a library shared between the client and server, you need to stop both running applications, since Windows forbids writing to a DLL that is currently in use.

Be careful when setting breakpoints if the client and server library share source files. If you set a breakpoint when editing a shared file, the breakpoint will be set in the editor's active project. You can change the active project using the drop-down list in the main window.

Breakpoints set in other windows' source pages (such as in the browser) act on the project associated with that window. Note that this makes it possible to set breakpoints in both the client and the server so that the debugger correctly opens up on the appropriate project as the breakpoints are reached. However, you cannot set the same breakpoint in both projects at once. Instead you have to go into each project and set the breakpoint separately.

Exporting a bug report or a compiler warnings report

You can save a formatted bug report or compiler warnings report for an application by choosing **File > Export...** in the debugger or project window. The bug report includes a stack backtrace for all threads in the application, while the compiler warnings report contains the same information provided in the project window's Warnings page.

The Export... dialog gives you the option of saving the report as either text HTML. If you choose to save the report to a file, an editor window automatically opens to show the file. The saved report contains a section for user-supplied information into which you can type supplemental text.

Note: This is **not** a facility for saving backtraces for any bugs you find in the Open Dylan environment. The debugger cannot introspect on the development environment's threads.

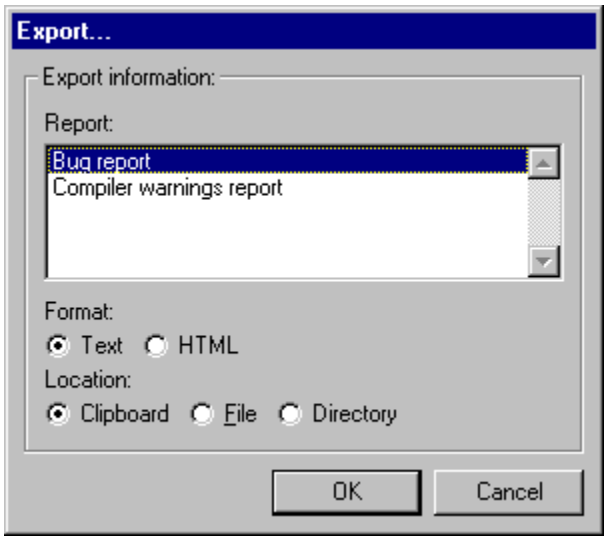

Fig. 8.7: The Export... dialog.

Debugger options

The **View > Debugger Options** command brings up a dialog that controls the options for the debugger. This dialog has three property pages: Stack, Exceptions, and Misc. The options on these pages apply on a per-thread basis—you can have different settings for different threads' debuggers.

Stack options

The Stack page controls the filtering of stack frames from the debugger's backtrace pane.

Show stack frames of types

- Check the boxes to show any of the following frame types: "Dylan function calls", "Foreign function calls", "Cleanup frames", and "Unknown stack frame types". By default, only "Dylan function calls" is selected.
- The selections made here correspond to the "Filtered" rules used in the filtering drop-down list. If you change the types of stack frames to be shown after filtering, the new filtering rules are applied by the filtering drop-down list selections when you next invoke a debugger window. The new rules also persist between sessions.

Show stack frames from modules

- Choose one of "Current module" (show frames whose corresponding definitions are defined in the current module only), "Current module and imported from used modules" (show frames from the current module and the modules it uses; the default), and "All modules" (show frames from all modules in the thread).
- These three options correspond to the "local", "visible" and "all" statements in the filtering drop-down list.

Show stack frames matching

- Enter a string in the "Include" text box; only frame names including this string will be shown in backtraces.
- Enter a string in the "Exclude" text box; frame names including this string will be excluded from backtraces.

Exceptions options

The Exceptions page controls the action taken when a particular exception occurs in the thread. Use the Action list to select an action.

The possible actions are:

- Ignore Throw the exception away and allow the application to continue.
- Report Write the message into the debugger's interaction pane and continue.
- Debug Pause the application. Update the debugger for the thread that signalled the exception. Write the message into a log. Activate any other debugger panels, but without updating them automatically. Allow an arbitrary amount of debugging and continue executing the application once **Application > Resume** is selected.

Miscellaneous options

The Misc page presents miscellaneous, high-level debugger options.

- Use a new debugger for each thread
- When checked, uses a new debugger window for each new thread you choose to debug.
- Confirm before entering the debugger after an error
- When checked (the default), presents an application error dialog asking if you want to abort the current application, continue with a selected restart, debug the application or exit the application. See *A Dylan run-time application error.*.

Expand stack backtrace when first opening debugger

- When checked (the default), expands the stack backtrace one level, to show stack frames. Otherwise just shows the application's thread number and title.
- Expand first stack frame when refreshing debugger

- When checked (the default), the debugger displays the first stack in expanded form and selects the code whenever a debugger appears or is refreshed.

Refresh all open debuggers when entering debugger

- When checked, refreshes stack information in all threads' open debugger windows upon entry to the debugger by any one thread. Otherwise only the debugger window for the thread that has entered the debugger will be refreshed. This option is not checked by default.

Open debugger window on pause button

- When checked (the default), clicking the toolbar Pause button or choosing **Application > Pause** causes the application to enter the debugger as well as pausing execution. Otherwise these actions only cause the application's execution to be paused. This option is checked by default.

Just-in-time debugging

The Windows operating system has the built-in capability to perform "just-in-time" debugging. Just-in-time, or JIT, debugging is where an application crashes while not running under a debugger, and the operating system arranges to start up an available debugger and attach it to the crashed process in order to obtain a backtrace. The system registry contains an entry for the debugger that should be invoked when this happens. Open Dylan is capable of acting as a JIT debugger; during the installation process you have the opportunity to install Open Dylan as your machine's default debugger.

If you set Open Dylan up as your JIT debugger, it is simple to make a connection to the Open Dylan debugger "just in time". When the application signals an error, and that error is unhandled, the operating system displays a dialog giving you a chance to attach to a debugger. If you then click **Debug** in the dialog, the Open Dylan debugger can attach to the application.

That is the procedure for console applications. Attaching the debugger to a DUIM application takes slightly longer. DUIM applications have a *default-handler* method that displays a standard dialog describing the unhandled Dylan error, and offering the chance to terminate the application (*Yes* button), ignore the error (*No* button), or debug the application (*Cancel* button). This is the dialog that we saw in *Problems at run time*.

To attach the debugger in this situation, click **Cancel**. Because the application is not running under a debugger already, the error falls through to be caught by the operating system. At this point, the operating system displays its dialog and you can click **Debug** to make the Open Dylan debugger attach.

Once the debugger attaches successfully, another dialog appears, asking whether you would like to open a project. If the crashed process is a Dylan application for which you have a project, you should take this opportunity to open the related project before you start your debugging.

It is not strictly necessary to open a project, since the debugger will still be able to use whatever debugging information is available in the executable file itself in order to provide a backtrace. But it is worth opening a project because you can then browse the source code or the definitions in the project, and use the interactor to evaluate Dylan expressions.

(Of course, if the crashed application is not a Dylan application at all, and also does not contain any embedded Dylan components, then you will not be able to open a project.)

Once you have made your selection and have clicked *OK*, the Open Dylan debugger window appears. The debugger provides a full description of the state of the program at the point of the crash. You can then start to debug the application in the normal way.

CHAPTER

NINE

REMOTE DEBUGGING

Running a Dylan application on a remote machine

Open Dylan offers a facility for running, debugging, and interacting with a Dylan application, DLL, or OLE control ("program" hereafter) running on a remote machine—that is, a networked machine other than the one running the Open Dylan IDE.

The ability to do these things on the remote machine is a simple extension of the standard features described in Debugging and Interactive Development, which covers debugging and interaction techniques. The techniques for running, debugging, and interacting with a program are exactly the same as for the local machine, but there are a few initial configuration issues to cover.

Installing the program and debugging server on the remote machine

To do anything with a Dylan program on a remote machine, the program has to be installed there. It may be there already—if, for instance, you are working with another Dylan developer—but if it is not, you must install the program on that machine.

To install the debugging server and your program on a remote machine, perform these steps on the target machine:

1. Install the Open Dylan runtime system.

   The runtime installer executable, the name of which begins "*hdrun* . . . ", resides in the *Extras* folder at the top-level of your Dylan CD-ROM.

2. Install the debugging server application.

   The debugging server installer executable, the name of which begins with "*hddbg* . . . ", also resides in the *Extras* folder.

3. Copy the *bin* folder for your project onto the remote machine.

Starting the debugging server

Whenever you try to invoke the program on the remote machine, Open Dylan expects to be able to talk to the debugging server application on the remote machine. If this server is not running on the remote machine, it is not possible to run the Dylan program there either.

The debugging server must be started manually on the remote machine itself, so it is worth installing it there. (See *Installing the program and debugging server on the remote machine* for information about using the debugging server installer.) Of course, if the remote machine already has the Open Dylan installed on it, you do not need to install the debugging server.

To start the debugging server choose **Start > Programs > Open Dylan > Start Debug Server**. You can also invoke it with *debugger-server.exe*, which is available in the *Bin* subfolder of the Open Dylan installation folder.

If the debugging server starts up correctly, a Open Dylan Debugger Server window opens. The window provides up-to-date information about active local debugger processes and remote connections. The upper pane of the window shows all locally running processes being debugged on remote machines, and the lower pane shows the remote machines that are connected to the server. A status bar at the bottom of the window displays a summary.

You can set (and later change) a password for the debugging server by using the **Change Password** button on the Debugger Server window. The password can be anything. By default, no password is required.

To exit the Debugging Server, click the **Exit** button or simply close the window.

At this point, you can return to the machine running the Open Dylan IDE, where you will be ready to run Dylan programs remotely.

Starting an application remotely

Running and debugging an application on a remote machine is identical in almost every respect to using your local machine. You can use the three menu items **Application > Start**, **Application > Debug** and **Application > Interact** to launch the application in the normal way. The only difference is that for remote startup you must specify a remote machine in the project's debug settings.

1. Select **Project > Settings...** and select the Debug page.

   In the "Remote machine" section, an option box displays the name of the machine on which the project's application is to be run. By default, the selection will be the local machine. Before you can select a remote machine, you must first establish a debugging connection.

2. Click the **Open New Connection...** button.

3. In the dialog that appears, enter the network address of the machine to which you wish to connect, and enter the password for the debugging server (if one is set), and click **OK**.

   The address is whatever the Windows operating system needs to identify the machine on the network. A computer hostname is likely to be sufficient.

   The password is the password established when you first started the debugging server (see *Starting the debugging server*).

   If the connection is successful, the newly connected machine is added to those listed in the "Remote machine" option box. You can now select this machine.

   If the connection does not succeed, you should ensure that you have successfully started the debug server program on the remote machine. (See *Starting the debugging server* above.)

4. Select the remote machine in the option box.

   Having selected a remote machine, you must also ensure that Open Dylan can locate the program (EXE or DLL) on the filesystem of the remote machine.

5. Specify the path to the program in the "Command line" section of the Debug page.

   The path should be fully qualified, including the name of the remote machine. For example:

```
\\spiral\c_drive\apps\reversi\release\reversi.exe
```

   where *spiral* is a machine name, and *c_drive* is the share name of the drive containing the program *release* folder.

You are now ready to run and debug the application remotely. **Application > Start** starts your application running on the remote machine. All the usual debugging and interaction facilities will now be available.

Details about the connection to a remote machine are stored and saved with the project. Open Dylan tries to re-establish the remote connection automatically when you next open and try to run the project.

Attaching to running processes

The three commands **Application > Start**, **Application > Debug** and **Application > Interact** are all used to launch a program from within Open Dylan. But if the program is running already, perhaps even before you started up the Open Dylan environment, the environment does not know about the process and therefore it has no visible presence in the environment.

Open Dylan allows you to "attach" to such a running process, thereby bringing it under Open Dylan's control just as if it had been started by the environment. It is very simple to do this:

1. Open the project whose application, DLL, or control is running.
2. Choose **Application > Attach...** from the project window, or choose **Tools > Attach Debugger...** from the main window.

   A dialog listing all available running processes appears.
3. Select the process to which you want to attach, and click **OK**.

After a few moments, the debugger attaches to the running process, and all the normal debugging and interaction facilities become available, just as with **Application > Start**.

Note: Open Dylan does not currently offer any facility for detaching from a process. Once it has been attached to the Open Dylan debugger, and all of your debugging work is finished, you will need to close the program down using the **Application > Stop** command.

The process to which you attach need not be running on the local machine. You can also attach to a process that is running on a remote machine provided that the debugging server application (see *Starting the debugging server*) is running on that machine. The process list dialog has an option box that allows you to select the machine whose process list you want to view. There is also an **Open New Connection...** button for creating new connections to remote machines, which works in the same way as described in *Starting an application remotely*.

CHAPTER

TEN

DISPATCH OPTIMIZATION COLORING IN THE EDITOR

This chapter is about source-code coloring in the editor that shows where and how optimizations have taken place.

Note: Optimization work is best done in Production mode. See *Project settings*.

About dispatch optimizations

When you call a generic function in Dylan, the method that will be executed has to be selected from the set of methods defined on that generic function.

The method is selected by comparing the types of the arguments passed in the generic function call to the parameter lists of the methods available; the method whose parameter list is closest to the types of the arguments passed in the call is the one that will be selected. (Note that there may be situations where no one method is more applicable than another, or even where there is no applicable method at all. These situations, which may be detected either at compile time or at run time, signal an error.)

The process of selecting the right method to call is known as *method dispatch*. The algorithm for selecting a method is described in Chapter 6 of the DRM.

Method dispatch can, in principle, occur entirely at run time; but in some circumstances, the Open Dylan compiler can work out at compile time precisely which method needs to be called, and so optimize away the need for run-time dispatch, making delivered applications faster. Depending on the circumstances, these *dispatch optimizations* can consist of replacing the generic function call with a direct call to the correct method; replacing the generic function call with a class slot access; or inlining the call completely.

Optimization coloring

When you compile a project, the Open Dylan compiler records the kinds of optimizations it performed for each source code file in the project. It also records cases where compile-time optimization was for one reason or another not possible.

The Open Dylan editor provides a way to see this information, by choosing **View > Color Dispatch Optimizations**. This command colors a source code file so that you can see where the compiler managed to optimize method dispatch, and also places where you may be able to make changes that will make dispatch optimizations possible next time you compile the project.

For instance, you will see a generic function call colored in blue where the compiler worked out the exact method to call and emitted a direct call to that method in the compiled code. *Dispatch optimization colors and their meanings* contains a full list of colors and their meanings.

Editing colored source code

Dispatch optimization coloring in a file shows only what the compiler achieved the last time you compiled the project containing the file. If you edit colored source code, the changes you make could affect the optimizations the compiler can carry out upon it the next time you compile the project.

For this reason, when **View > Color Dispatch Optimizations** is on and you edit a colored line, the editor resets the entire line to black. All the other lines in the file keep their color.

In this situation, you can re-color the line by doing **View > Refresh**. Note that once you have edited the text, the coloring information may no longer fit it, but you may still find it useful. Alternatively you may prefer to turn off coloring altogether once the coloring information for part of the file has been invalidated.

Effect of compilation mode on dispatch optimizations

The Open Dylan compiler tries to optimize method dispatch whether compiling a project in Interactive Development mode or in Production mode. Because the compiler does more optimization in Production mode, you will see more coloring information after a Production-mode build.

Dispatch optimization colors and their meanings

The following table shows the colors used to indicate different kinds of dispatch optimization.

Color	Meaning	Recommended Action
Magenta	Call not optimized because the compiler could not determine all the applicable methods.	Happens when the generic function is open. Where possible, turn open protocols into sealed protocols.
Red	Call not optimized despite the compiler finding all the applicable methods. Can happen when the type of an argument is not specific enough.	Where possible, add type specializers to the bindings of the arguments to the call.
Blue	Call optimized. The compiler found the appropriate method, and emitted a direct call to it.	None required.
Green	Call optimized. The compiler found that this call was an access to a slot whose position in a class is fixed. It replaced the function call with (faster) code to access that slot value.	None required.
Dark Gray	Call optimized. The compiler inlined the code of the appropriate method.	None required.
Light Gray	This code was eliminated completely. The compiler either determined that this call would never be made or that it would not make any difference to the outcome of other code with which it was associated, or it managed to evaluate the call directly.	None required. (Unless the code should have been called.)

Where possible, add type specializers to the bindings of the arguments to the call.

Optimizing the Reversi application

In this section we look at the dispatch optimization color information for part of the Reversi application and see what we can do to optimize it.

Before doing that, we should build the Reversi application in Production mode so we know that the application has been optimized as much as possible.

Open the Reversi project.

1. Choose **Project > Settings** and, on the Compile page, set the compilation mode to "Production mode".
2. Choose **Project > Clean Build**.
3. When the build is complete, go to the Sources page and open the file *game.dylan*.

An editor window showing *game.dylan* appears.

1. In the editor window, turn on the **View > Color Dispatch Optimizations** check item.

We can now see color information showing how dispatch optimizations were or were not carried out during the last build.

1. Go to the definition of the method `<reversi-game>`.

You can use **Edit > Find** or the "binoculars" toolbar button to do this.

This is the definition of `<reversi-game>` :

```
define class <reversi-game> (<object>)
  slot reversi-game-board :: <reversi-board> = make(<reversi-board>);
  slot %player :: <player> = #"black",
    init-keyword: player:;
  slot %players :: <integer> = 1,
    init-keyword: players:;
  slot black-algorithm :: <algorithm> = default-algorithm-for-player(#"black"),
    init-keyword: black-algorithm:;
  slot white-algorithm :: <algorithm> = default-algorithm-for-player(#"white"),
    init-keyword: white-algorithm:;
  slot reversi-game-update-callback :: <function> = always(#f),
    init-keyword: update-callback:;
  slot reversi-game-message-function :: false-or(<function>) = #f,
    init-keyword: message-function:;
end class <reversi-game>;
```

There are three different colorings in this definition. The call to the function *always*, a Dylan language built-in function, is in light gray. That means the call has been eliminated completely from the compiled application. A call to the function *always* is defined to return a function object that always returns the value passed in the call to *always*. So here, the function object would always return `#f`. Unsurprisingly, the compiler evaluated this call completely, avoiding the need for run-time method dispatch.

The two calls to *default-algorithm-for-player*, a Reversi application method from *algorithms.dylan*, are colored in blue, signifying that the compiler managed to determine precisely which method to call, and inserted a direct call to that method in the compiled application. Again, the need for run-time method dispatch was averted.

Investigation shows that there is only one method on *default-algorithm-for-player*, which makes blue optimization simple here. The generic function for *default-algorithm-for-player* is defined implicitly, in the single `define method default-algorithm-for-player` call. Recall from the DRM (chapter 6) that implicitly defined generic functions are sealed by default. That fact allows the compiler to conclude that this method is the only method there will ever be on *default-algorithm-for-player*, making the optimization possible.

The third coloring is magenta, in the call to `make` on `<reversi-board>`, in the *reversi-game-board* slot definition. Here, then, is a generic function call that was not optimized. Magenta coloring means that for this call to `make`, the compiler could not determine the complete set of methods from which it could attempt to select the appropriate method to call. We will now make changes to the Reversi sources to optimize this call.

The problem here is that the compiler cannot be sure that additional methods on `make` might not be added at run time. By defining a sealed domain on make for `<reversi-board>`, we can clear this up.

1. Add the following to *game.dylan* :

```
define sealed domain make(subclass(<reversi-board>));
```

With this information, the compiler knows it has access to the complete set of methods on `make` for this class, and therefore can attempt to do the method selection itself.

We can recompile the application to see what effect our change has had.

1. Save *game.dylan* with **File > Save**.
2. Rebuild the application, and refresh the color information for *game.dylan* with **View > Refresh**.

The refreshed coloring shows the call to `make` on `<reversi-board>` in the *reversi-game-board* slot definition in light gray. This coloring means that the compiler determined which `make` method to call, computed the result of the call—a `<reversi-board>` object—and inlined the object.

Looking further down *game.dylan*, notice that the definition of *reversi-game-size-setter* also calls `make` on `<reversi-board>`, a call that is also colored light gray.

We can now look at other possible optimizations in *game.dylan*.

1. Go to the definition of the method *initialize-board*.

The definition of *initialize-board* is:

```
define method initialize-board (board :: <reversi-board>) => ()
  let squares = reversi-board-squares(board);
  for (square from 0 below size(squares))
    squares[square] := #f
  end;
  for (piece in initial-pieces(board))
    let square = piece[0];
    squares[square] := piece[1]
  end;
end method initialize-board;
```

In this method there is a green-colored call to *reversi-board-squares* on the parameter *board*, an instance of `<reversi-board>`. Green coloring denotes an access to a slot whose position in a class is fixed. This optimization was possible because the *reversi-board-squares* method is just the implicitly defined accessor for the slot *reversi-board-squares* :

```
define class <reversi-board> (<object>)
  slot reversi-board-size :: <integer> = $default-board-size,
    init-keyword: size:;
  slot reversi-board-squares :: <sequence> = #[];
end class <reversi-board>;
```

The compiler achieved this optimization because it knew three things. First, it knew that the generic function implicitly defined by the accessor method was sealed. (As normal Dylan methods, accessor methods implicitly define a generic function if one does not already exist; such a generic function is sealed because implicitly defined generic functions are sealed by default.) Second, the compiler knew the type of *board* in the call to the accessor method. Third, the compiler knew that the class `<reversi-board>` was sealed, because classes are sealed by default.

We can now move on to some other optimization. The call *size(squares)* in *initialize-board* is colored in magenta. There are several similar magenta colorings in *game.dylan*, where the compiler could not optimize a method call on the value returned from *reversi-board-squares* : calls to *element*, *element-setter*, *empty?*, and *size*. In all cases this is because the type of *reversi-board-squares* is `<sequence>`, which is an open class.

We could seal domains on `<sequence>` to get optimizations here. But the DRM defines `<sequence>` as an open class, and it is not good practice to seal protocols that do not belong to your library or libraries. However, we can change the type of *reversi-board-squares* to be in a domain which is already sealed. Changing the slot type to `<simple-object-vector>` gives us a sealed type as well as preserving the protocol in use, so that we do not have to change any of the calls being made.

1. Go to the definition of `<reversi-board>`.
2. Change the type of *reversi-board-squares* to be `<simple-object-vector>`.
3. Save *game.dylan* with **File > Save**.
4. Rebuild the application, and refresh the color information for *game.dylan* with **View > Refresh**.
5. Go back to the definition of *initialize-board*.

The *size(squares)* call is now colored green. Green coloring means the compiler determined that the call was equivalent to a slot access—particularly, an access to slot having a fixed offset from the memory address at which its class is located. The compiler removed the need for run-time method dispatch by replacing the call with code to access the location that would contain the slot value.

This particular optimization was possible because *size* is a slot accessor for instances of `<simple-object-vector>`, and, of course, because `<simple-object-vector>` is sealed.

You could examine the effects of this change on other calls that use the return value of *reversi-board-squares*. Some calls turn blue. Some calls to *element-setter* remain magenta because the compiler does not know the type of the index. Constraining the type of the index would improve such a call, turning it blue or even dark gray (inlined).

CHAPTER

ELEVEN

DELIVERING DYLAN APPLICATIONS

Applications you have written using Open Dylan need access to Open Dylan's run-time libraries in order to run. This applies to all applications, whether executables or DLLs. The run-time libraries are normal Win32 DLLs, stored in the top-level installation folder *Redistributable*.

When you run an application on a machine where Open Dylan is installed, the libraries will be found in *Redistributable*, whether you run the application from the command line or from within the Open Dylan environment using commands such as **Application > Start**.

But to deliver your Dylan application to a customer or other third party, you will need to include in your distribution the Open Dylan run-time libraries that the application uses.

This chapter discusses two methods of delivering Dylan applications with the necessary run-time libraries: using the environment to build a Release folder, and using Open Dylan's stand-alone run-time library installer.

Building a release folder

To create a single folder containing everything necessary for your application to run on a customer's machine, use the **Project > Make Release** command in the project window.

This command takes the compiled application files associated with the project, and whichever Open Dylan run-time libraries (DLLs) are necessary for the application to run, and copies them into the *Release* subfolder of the project's own folder.

You can then distribute the entire *Release* folder as a stand-alone application. Read the Open Dylan license agreement for details of the legal side of redistributing the Open Dylan run-time libraries.

Using the run-time library installer

An alternative to building a single release folder is to use Open Dylan's run-time library installer, a self-extracting executable that installs all the Open Dylan run-time libraries (DLLs) in a central location.

The default location is

```
C:\Program Files\Common Files\Harlequin\System
```

The run-time library installer also sets the PATH environment variable to include this folder.

Your distributable application should then consist of a copy of your compiled application files from the project's *Bin* folder, and the run-time installer.

The run-time library installer is included on CD-ROM editions of Open Dylan, and can also be downloaded from Harlequin's World Wide Web site. You can distribute the run-time installer to customers, or allow them to download it themselves.

About the run-time library DLLs

The run-time library DLLs have 8.3 format names and include a version number. This version number will be incremented for new releases of the run-time libraries.

CHAPTER

TWELVE

THE INTERACTIVE EDITOR

The Open Dylan environment has a text editor that is specifically designed to make it easy to write and interact with your Dylan code. One of its most unique features is that it actually lets you compile selections of code as you develop your application. This is why we say the editor is *interactive*.

You can use the editor in one of two styles: Emacs or Windows. Using the editor options, as described in ‘ <../../../../../../ldisk2/doc/dylan/product/guide/environment/src/appx.htm#52521>‘_, you can choose from either an Emacs or Windows style keyboard layout and default settings.

The editor allows you to perform a wide range of operations by using menu commands, as well as keyboard commands. These operations range from simple tasks such as navigating around a file to more complex actions that have been specifically designed to ease the task of writing and working with Dylan code, such as compiling selected blocks of code, setting breakpoints, editing multiple sources, and browsing objects. *You can do all of these things while your application is running.*

This chapter describes the editor and gives you a general overview of how to use it. Some familiarity with Emacs and Windows usage is presumed.

While basic use of the editor is quite intuitive, by becoming familiar with the menus and options, you can more effectively use the editor to perform advanced operations.

Invoking the editor and displaying files

The editor window lets you read and edit text files stored in your filesystem. You can invoke the Open Dylan editor in a variety of ways:

- Click the New Text File (
- Double-click on a file listed in a project window and an editor window opens and displays the file.
- Click the Edit Source (
- Choose a file name from the *File* menu on a project window, an existing editor window, or the main window. (The main window’s *File* menu only lists files if they have been viewed previously.)
- Choose *Edit Source* from the popup (shortcut) menu for an object.

When the editor opens, a buffer containing the text of the current file is displayed, and you can move around it and change its contents as you wish, then save it back to the original file (assuming that you have permission to write to it). *Editor window showing the game.dylan file from the Reversi project.* shows a file opened in an editor window.

Display conventions

The editor separates each definition in a source file with a gray line. Printed in the middle of each line is the name of the definition below it. These code separators also appear above top-level expressions wrapped with *begin* … *end*. A

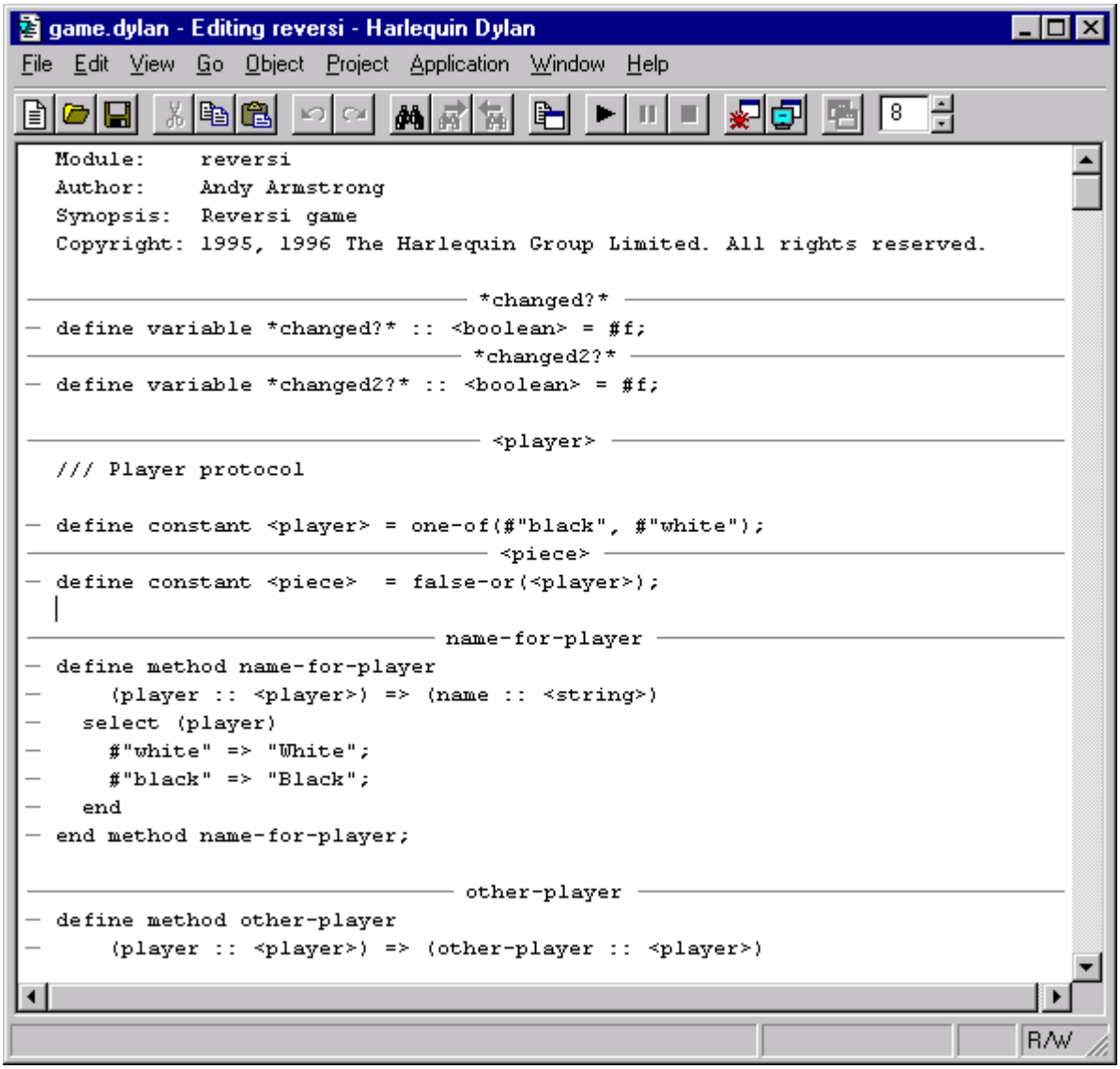

Fig. 12.1: Editor window showing the game.dylan file from the Reversi project.

file has to have been compiled for code separators to show.

The code separators are just a visual aid, and are not part of the file itself. If you opened the source file in a different editor you would not see the separators. The compiler also ignores them.

When you add a new definition, or a new *begin ... end* top-level form, the code separators will only be updated if you manually refresh the editor window (**View > Refresh**), move the cursor past an existing separator, or perform some other operation that forces the editor to redisplay.

If you have opened a source file in the editor by double-clicking on an item in the debugger's source pane, Open Dylan positions the insertion point at the line containing the problem. If the problem identified by the debugger spans a range of lines, the entire range is highlighted in the editor. Likewise, if the problem is an undefined binding and you have opened the source from the project window's Warnings tab page, the binding in question is highlighted.

Changing the editor options and layout

By default the editor uses Windows-style defaults and therefore associates each opened file with a new editor window. However, you can change this and edit many different files at once in the same editor by using the **View > Editor Options...** Display tab page. Alternatively, you can use the Restore tab page to switch to Emacs defaults, which changes this default (and others). For details about the Editor Options dialog, its tab pages and settings, see ' <../../../../../../ldisk2/doc/dylan/product/guide/environment/src/appx.htm#52521>'_.

The editor offers a number of different viewing options for the window's layout.

- The status bar, located at the bottom of the editor window, lets you examine any output messages from the environment. You can show or hide the status bar using **View > Status Bar**. This is where you will see messages displayed about your interactive development, such as "Compiling definition..." when you compile a selection.
- The standard Dylan toolbar can be viewed or hidden with **View > Toolbar**.
- The color dispatch optimizations feature, as described in Dispatch Optimization Coloring in the Editor, shows you where and how to optimize your code and is controlled by **View > Color Dispatch Optimizations**.

The editor window's context: the active project

In an editor window, the toolbar and menu commands act upon the application of the active project. For instance, the **Project**, **Build**, and **Application** menus are not available in the editor if the source file being edited is not part of the active project—the project whose name is visible in the main window's drop-down list. See *The active project* for more details.

Menu commands and special features

The editor offers all of the standard **File** and **Edit** menu commands a user expects to find on a text editing window (such as New, Open, Cut, Copy, Paste, Find/Replace, and so forth). The editor also has the standard Open Dylan menus, such as **Go**, **Project**, **Application**, and **Window**. This section describes the additional menu commands provided by the Open Dylan editor.

The **Go** menu commands open an appropriate window, usually the browser or the project window, to show you the corresponding item. For instance, **Go > Breakpoints** opens the project window for the active project and displays the Breakpoints tab page. Likewise, when you choose **Go > Edit Compiler Warnings** an editor window opens on the source code corresponding to the first compiler warning for the project. The cursor is automatically positioned at the problem area in the code. In Emacs mode, you can use Ctrl+. (Ctrl+ period) to view the source for the next compiler warning, and so forth.

The **Object** menu commands require that you place the cursor in an element name in the editor window. The commands then allow you to browse that element or edit the related code. For instance, if the cursor is placed in a class name, **Object > Edit Subclasses** opens an editor window to display a composite buffer containing the subclasses of that class.

The editor's **Project** menu contains two special commands beyond the standard Project commands on other Dylan windows:

Compile Selection Allows you to compile a selection of code independently while an application is running. This is one of the editor's special interactive capabilities. For details and an example of interactive development using Compile Selection, see *Using the editor for interactive development*.

Macroexpand Selection When the cursor is placed in a macro in the editor window, choosing this command expands the macro code in the buffer so that you can see the actions it performs. Use **Edit > Undo** (or the toolbar/keyboard equivalents) to return to the original contents of the buffer.

The editor's **Application** menu is the same as for the debugger or the project window, except that it contains extra breakpoint commands. These breakpoint commands are also part of the shortcut menu. See *Breakpoint options* for details.

If you have Microsoft Visual SourceSafe installed, the editor displays a **SourceSafe** menu. This menu is Open Dylan's interface to source control. For more information, see *Source control with Visual SourceSafe*.

Shortcut menus

The editor provides a shortcut menu whose items vary depending on where your cursor is when you right-click. The most basic shortcut menu pops up if you right-click in a file when the cursor is **not** in the middle of a code element (like a method or a class name). This menu contains the items: Edit Source, Cut, Copy, Paste, Delete.

A more extensive shortcut menu pops up if you right-click when the cursor is in (or on either side of) a code element. In addition to the commands in the basic shortcut menu, this menu contains the following commands:

Describe Opens a window that lists the element's module, library, source file, and describes the element.

Browse Opens a browser window on the object.

Browse Type Opens a browser window on the type of the object.

Edit Source Takes you to the portion of code in the source file where that element is originally defined.

Edit Clients Opens an editor window that displays the users of the selected definition.

Edit Used Definitions Opens an editor window that displays definitions used by the selected definition.

Show Documentation Opens the Open Dylan HTML Help.

If you right-click when the cursor is in a method name, the shortcut menu also contains tracing commands and breakpoint manipulation commands. For details about these shortcut commands, see *Breakpoint options*.

Breakpoint commands are also available if you right-click when your mouse pointer is over the leftmost column of the editor window (see *Breakpoint options*). Underscores in the leftmost column indicate lines where you could add a breakpoint (see *Editor window showing the game.dylan file from the Reversi project.*).

Using the editor for interactive development

You will recall from *An example interaction with Reversi* that we were able to change the shapes of the Reversi game pieces while the application was running simply by providing new definitions in the interactor. Now imagine that you were developing the Reversi application and wanted to interact with it as you coded the sources. The Open Dylan

editor allows you to compile pieces of your code and see the results in the running application by using **Project > Compile Selection**.

In the following example we interact with the Reversi application from an editor window opened on one of the Reversi sources.

Open the Reversi project, and choose **Application > Start**.

Make some moves on the board.

Open the file *board.dylan* in the editor by double-clicking it in the reversi project window.

Find the variable definition:

```
define variable *reversi-piece-shape* = #"circle";
```

Change the word *circle* to *square*, so that the line reads:

```
define variable *reversi-piece-shape* = #"square";
```

Select the line of code and choose **Project > Compile Selection**.

Notice that the status bar says "Compiling region…" and then "*reversi-piece-shape* successfully downloaded".

Make a few moves on the board.

The new moves and any refreshed area of the board display square game pieces.

Unlike the interactor, which compiles and executes the code you enter in the context of the paused thread to which the debugger is connected, the editor compiles the code you select in the context of a special interaction thread that it chooses automatically. This prevents unnecessary tampering with user threads.

Source control with Visual SourceSafe

To simplify the process of working with files under source control, the Open Dylan editor provides an interface to Microsoft's Visual SourceSafe, an external source code control system. This section describes the editor interface to Visual SourceSafe. (For information on using Visual SourceSafe, see Visual SourceSafe documentation.)

What is the editor's source control interface?

A source code control system provides one or more repositories (databases) where developers place source code and related files for a project. Files in the database can be accessed by several developers simultaneously by copying them from the database to their local machine. A developer can "check out" one or more files from the database in order to make changes and, when finished, "check in" the updated files for use by other developers. The database maintains a history of the changes made to each file, making it possible to retrieve older versions if necessary. It also provides mechanisms to resolve conflicts when two or more developers are making changes to the same files at the same time.

The Open Dylan source control interface provides access to a subset of Visual SourceSafe features, which are described in *The SourceSafe menu commands*. To perform more complex actions, you must use the source control system's native interface.

Open Dylan detects a Visual SourceSafe installation on a machine and automatically makes its interface to source control available by creating an additional **SourceSafe** menu on editor windows (see *The SourceSafe menu on a Open Dylan editor window.*). Therefore, you must have Visual SourceSafe installed in order to see the **SourceSafe** menu.

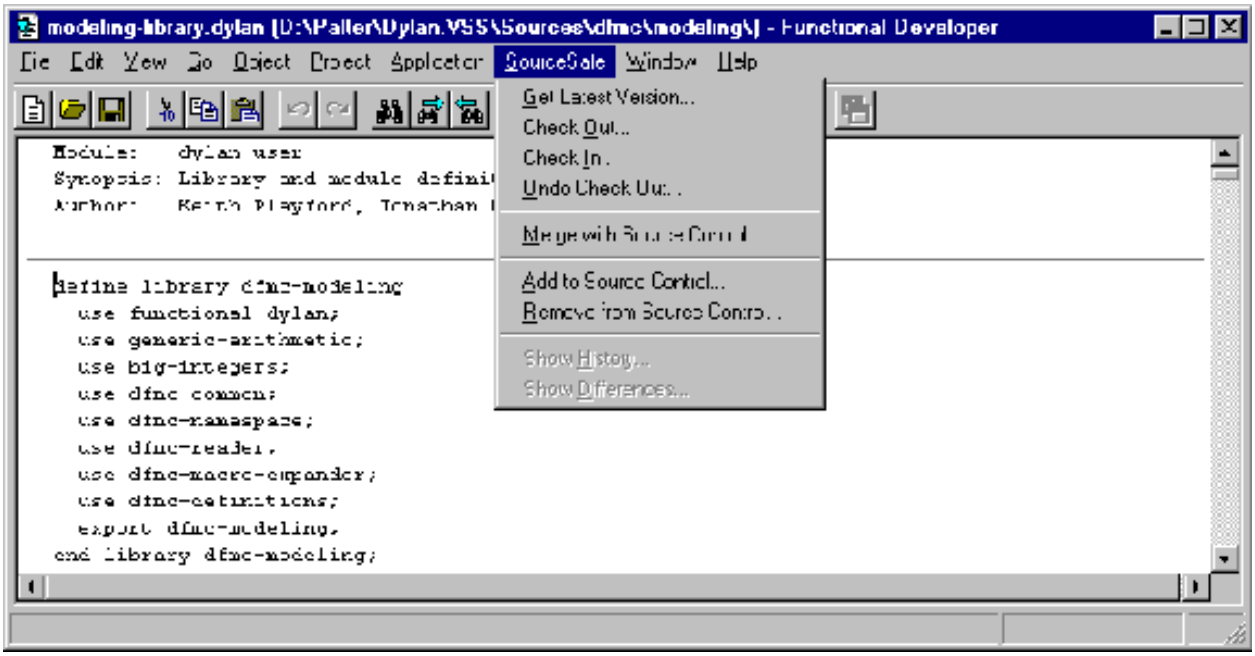

Fig. 12.2: The SourceSafe menu on a Open Dylan editor window.

The SourceSafe menu commands

The **SourceSafe** menu provides the following commands:

Get Latest Version... Copies the latest version of a file in the database onto the developer's machine.

Check Out... Copies the latest version of a file in the database onto the developer's machine. In addition, the database is updated to note that the file is being edited by the developer.

Check In... Copies a file from the developer's machine back into the database, creating a new version of the file, and notes that the file is no longer being edited by the developer.

Undo Check Out... Notes in the database that a file is no longer being edited by the developer and does not change the latest version of the file. In addition, the latest version of the file is copied from the database onto the developer's machine; any changes the developer may have made to the local copy of the file are abandoned.

Merge with Source Control... Merges the changes made by the developer to the local copy of a file with the latest version of the file in the database and replaces the local copy of the file with the merged edition; the file remains checked out by the developer.

Add to Source Control... Creates the first version of a file in the source control database using the copy on the developer's machine as the initial content.

Remove from Source Control... Removes a file and its history from the database.

Show History... Displays the list of changes made to a file as recorded in the database.

Show Differences... Compares the latest version of a file in the database against a copy on the developer's machine and displays the differences, if any. This command can be used to check to see if others may have made changes to a file that should be merged into the developer's copy before it is checked into the database.

Using the editor's source control interface

The first time you choose one of the **SourceSafe** menu items, the editor prompts you for the name of the SourceSafe database, as shown in *The Select Database dialog.*.

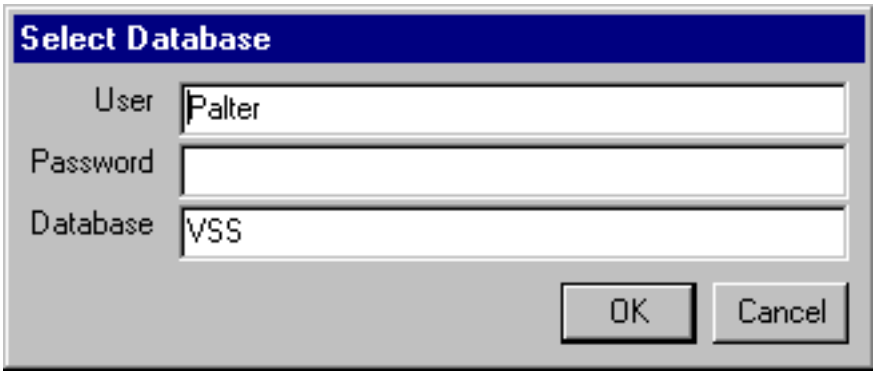

Fig. 12.3: The Select Database dialog.

Your site's Visual SourceSafe administrator supplies the name of your SourceSafe database. (The Open Dylan interface offers the name of the last database used in the Visual SourceSafe explorer as the default.)

The editor then requests the identity of the file (or files) to be manipulated by a source control operation, as shown in *The Select Project and File dialog.*.

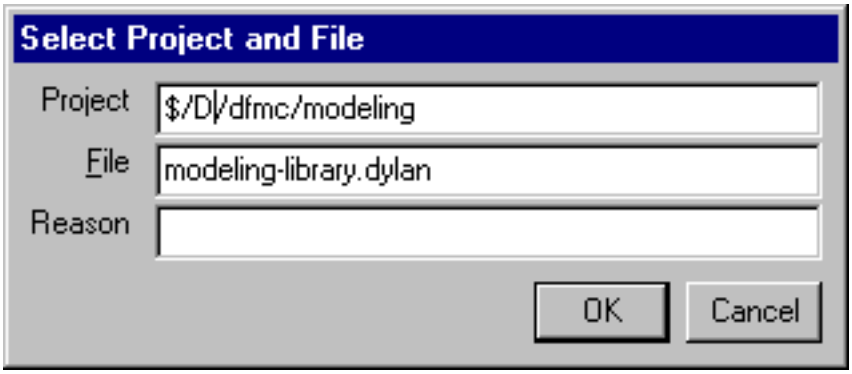

Fig. 12.4: The Select Project and File dialog.

SourceSafe organizes its database into a hierarchical collection of projects. Each project can hold both files and projects (in other words, subprojects). The project named *$/* refers to the root of the SourceSafe database. In *The Select Project and File dialog.* the developer has selected a project three levels below the root.

When using Visual SourceSafe, you may omit the file name in all operations (except for **SourceSafe > Add to Source Control...** and **Remove from Source Control...**) to cause the operation to be performed on all files in the project. For instance, to check out all the files in a project, leave the File field empty in the Select Project and File dialog.

The Reason field is provided mainly as a convenience. Not all operations prompt you for a Reason; in such cases the dialog only asks for the project and file names. When a Reason is requested, it is optional and may be left blank.

CHAPTER

THIRTEEN

CREATING COM PROJECTS

Working with COM type libraries

Microsoft COM interfaces and implementations of those interfaces are described on disk by *type libraries*. Type libraries can reside in stand-alone files with the extension .TLB, or within files containing executable code such as .DLL, .EXE, and .OCX files. Often the file that implements a COM component also contains the type library representing that component.

Open Dylan includes a tool that can read the contents of a type library and generate Dylan code to act as a client to the COM interfaces and classes the type library describes, or to act as a server implementing the type library's interfaces and classes.

You can invoke this type library tool from a Open Dylan project by including a *specification file* in the project. A specification file describes what type library to translate and in what way the translation will be used. Specification files have the extension .SPEC. You can find more details about specification files in section *The type library tool and specification files*.

To make using the tool easier, the New Project wizard allows you to create a project containing a specification file.

An example COM server and client

The following example explains how the type library tool works and how to use the New Project wizard features that support it.

In the example, we will create COM server and client applications. We will write a simple encryption engine as a COM server, and write a simple client to that interface. In order to make things simpler, we have provided a COM type library describing the encryption interface.

Creating the server stubs library

First we use the New Project wizard to create a Dylan library defining server-side stubs for the encryption interface.

Choose **File > New...** from the main window.

1. Select **Project** and click **OK**.

The New Project wizard appears.

1. In the Project Type section, select "Interface to COM Type Library" and click **Next**.

The next page allows you to name a type library to be translated. Most COM components store the location of their type libraries in the Windows Registry. These registered type libraries are listed in the Installed Type Libraries section

of the window. You can select a type library from the list, or click **Browse...** in the Location pane to select a type library file from disk.

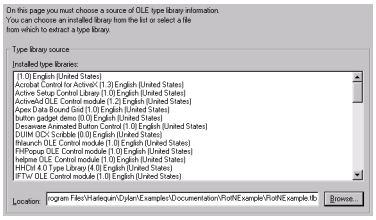

Fig. 13.1: Choosing a type library file to convert.

In this case, we use a type library supplied with Open Dylan.

1. Click **Browse...** and navigate to the Open Dylan examples folder.

The folder required is called Examples and is placed under the top-level Open Dylan folder.

It is usually *C:\Program Files\Open Dylan\Examples*.

1. Go to the *Documentation\RotNExample* subfolder and choose the *RotNExample.tlb* file.
2. Click **Next** to continue to the next page of the wizard.

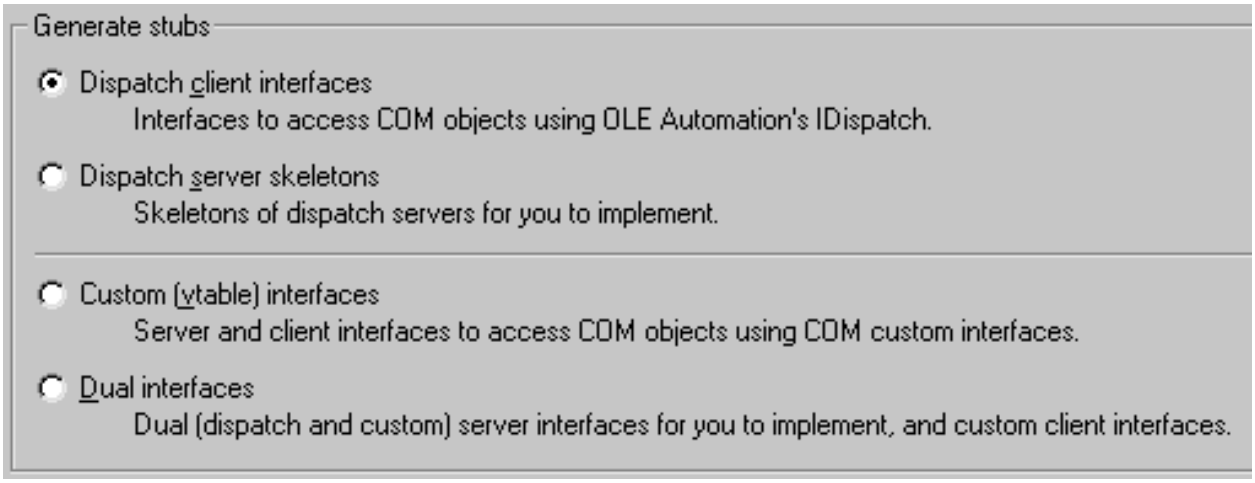

Fig. 13.2: Choosing the kind of skeleton code to generate.

The next page allows you to choose what kind of stubs to generate from the type library. There are two options:

Dispatch client interfaces

- Defines Dylan code to allow you to interface to COM servers.

Dispatch server skeletons

- Defines Dylan code to allow you to create COM servers implementing the interfaces described in the type library.

Because we are writing the server side of the application, we want to generate dispatch server skeleton code.

1. Select "Dispatch server skeletons".
2. Click **Next** to continue to the next page of the wizard.

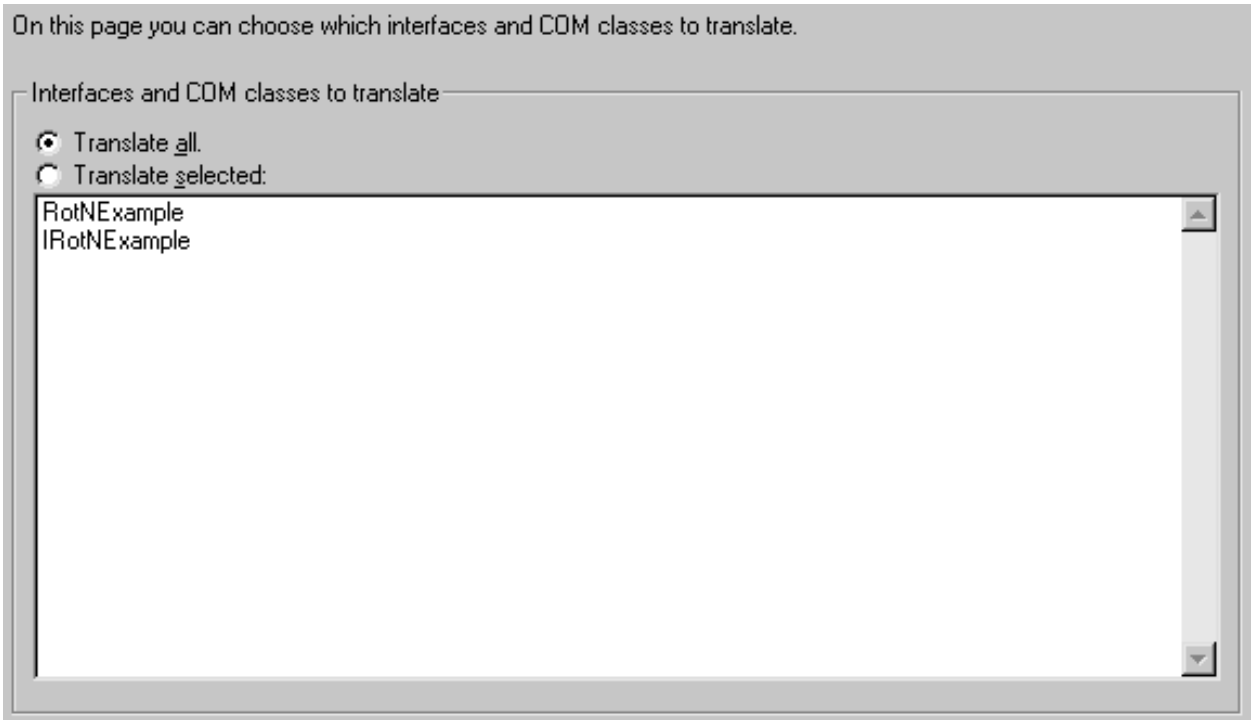

Fig. 13.3: Choosing interfaces and classes from the type library.

The next page presents a list of COM classes and interfaces contained in the selected type library. You can select which to translate by choosing "Translate selected" and then clicking to select individual items, dragging or using the Shift key to select ranges, and using the Ctrl key to select additional items. Choose "Translate all" and all classes and interfaces in the type library will be translated. This is different from selecting all items under "Translate selected" because if classes or interfaces are added to the type library later, they will only be translated if you selected "Translate all".

1. Choose "Translate all", so that both the RotNExample COM class and the IRotNExample interface are translated.
2. Click **Next**.

Now we reach the Project Name and Location page. This and all subsequent pages are the same as those that you see for other kinds of project in the New Project wizard. Follow the remaining steps to finish defining the server stubs project.

1. Change the name of the project to *RotNExample-server-stubs*.
2. Choose a suitable Location for the project.
3. Make sure that the "Dynamic Link Library (DLL)" option is chosen in the Project Settings and Templates section.
4. Make sure that the "Include any available templates" option is *not* checked.
5. Click **Next** to continue.

We are now at the Use Libraries page. We are only defining the stubs for the server, so we do not need any unusual libraries.

1. Choose the "Minimal" option.
2. Click **Next** to continue.

We are now at the final page of the New Project wizard.

1. Make any changes you want to the Source File Headers section.
2. Click **Finish**.

The new project appears.

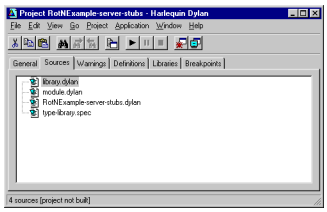

Fig. 13.4: The server stubs project.

In addition to the usual files, this project contains a file named *type-library.spec*. This is a specification file. It provides information to the type library tool.

1. Double-click on the specification file.

An editor window opens on the specification file.

The specification file looks something like this:

```
Origin: COM-type-library
Type-library: C:\\Program Files\\Open Dylan\\Examples\\...\\RotNExample.tlb
Module: type-library-module
Module-file: type-library-module.dylan
Generate: dispatch-servers
Stub-file: stubs.dylan
```

As you can see, the specification file contains all the information provided to the type library portion of the New Project wizard.

However, no skeleton code has yet been generated. The type library tool, which generates the skeleton code, only runs when you build the project.

1. Build the project with **Project > Build**.

The build process adds some new files to the project. These files were generated by the type library tool.

The file *type-library-module.dylan* defines a module in which the translated code resides. This module exports all translated symbols. If you look at *module.dylan*, you will see that the main module re-exports all of these symbols.

The file *stubs.dylan* contains the Dylan code generated by the type library tool. It defines a class for an implementation of the IRotNExample interface, and generic functions for the methods and properties of the interface:

```
/* Dispatch interface: IRotNExample version 0.0
 * GUID: {822ED42A-3EB1-11D2-A3CA-0060B0572A7F}
 * Description: An example interface for Open Dylan's Getting
 * Started manual. */

define open dispatch-interface <IRotNExample> (<simple-dispatch>)
  uuid "{822ED42A-3EB1-11D2-A3CA-0060B0572A7F}";
  virtual property IRotNExample/key :: type-union(<integer>,
    <machine-word>), name: "key", disp-id: 12288;
  function IRotNExample/encrypt (arg-pre :: <string>) =>
    (arg-result :: <string>), name: "encrypt", disp-id: 24576;
  function IRotNExample/decrypt (arg-pre :: <string>) =>
```

```
    (arg-result :: <string>), name: "decrypt", disp-id: 24577;
end dispatch-interface <IRotNExample>;
define open generic IRotNExample/key (this :: <IRotNExample>) =>
  (arg-result :: type-union(<integer>, <machine-word>));
define open generic IRotNExample/key-setter (arg-result ::
  type-union(<integer>, <machine-word>), this :: <IRotNExample>)
  => (arg-result :: type-union(<integer>, <machine-word>));
define open generic IRotNExample/encrypt (this :: <IRotNExample>,
  arg-pre :: <string>) => (result :: <HRESULT>, arg-result ::
  <string>);
define open generic IRotNExample/decrypt (this :: <IRotNExample>,
  arg-pre :: <string>) => (result :: <HRESULT>, arg-result ::
  <string>);
```

This defines a class `<IRotNExample>` which implements the given interface. The implementation is not complete until methods are provided for the generics. This can be done by adding methods in the same library, or by defining a subclass of `<IRotNExample>` in another library and defining methods on the subclass. We will take the latter approach.

There is also generated code corresponding to the COM class RotNExample from the type library:

```
/* COM class: RotNExample version 0.0
 * GUID: {C44502DB-3EB1-11D2-A3CA-0060B0572A7F}
 * Description: Implementation of IRotNExample.
 */

define constant $RotNExample-class-id = as(<REFCLSID>,
                                           "{C44502DB-3EB1-11D2-A3CA-0060B0572A7F}");
/* You should define your coclass something like this:
define coclass $RotNExample-type-info
  name "RotNExample";
  uuid $RotNExample-class-id;
  default interface <IRotNExample>;
end coclass;
*/
```

Note that the `define coclass` is within a comment, since you may want to define a COM class based on a subclass of `<IRotNExample>`.

Creating the server

Now we create the actual server application.

Choose **File > New...** from the main window.

1. Select *Project* and click **OK**.

The New Project wizard appears.

1. In the Project Type section, select "GUI Application (EXE)" and click **Next**.
2. Name the project *RotNExample-server*.
3. Make sure that the "Include any available templates" option is *not* checked.
4. Make sure that "Production mode" is selected in the Compilation Mode section of the Advanced Project Settings dialog.

This option will be set already if you have been following all steps from the start of this chapter.

1. Click **Next** to continue.

2. Choose the "Simple" libraries option and click **Next** to continue.
3. Choose the "Standard IO streams and string formatting" option from "IO and system support", and click **Next**.
4. Choose the "Win32 API" option from "GUI support", and click **Next**.
5. Choose the "OLE Automation" option from "OLE Automation support" and click **Next**.
6. Choose the "NoneY" option from "Database support" and click **Next**.
7. Click **Finish**.

The RotNExample-server project window appears.

1. In the new project's window, edit *library.dylan*, and add to the `define library` declaration the following line:

```
use RotNExample-server-stubs;
```

1. Add the same line to the `define module` declaration in *module.dylan*.

To implement the IRotNExample interface, we will create a subclass of `<IRotNExample>`. Because `<IRotNExample>` was created with `define dispatch-interface`, we must use `define COM-interface` to create the subclass.

Note: The remainder of this section of the example involves adding code to *RotNExample-server.dylan*. A version of this file with all the code we add in this section is available in the Open Dylan installation folder, under *Examples\Documentation\RotNExample\RotNExample-server.dylan*. You may want to copy that file into place in your project folder rather than typing code in.

---

1. Add the following code to *RotNExample-server.dylan*.

```
define COM-interface <IRotN-implementation> (<IRotNExample>)
  slot IRotNExample/key ::
  type-union(<integer>, <machine-word>) = 13;
end;
```

If you add this by hand, make sure not to put it after the top-level call to *main*.

We provide here an implementation for the *IRotNExample/key* slot, which was defined as a virtual slot in the superclass. This slot must accept the `<machine-word>` type, since any 32-bit integer which does not fit in the range of a Dylan `<integer>` will be passed as a `<machine-word>`.

The next task is to define the *IRotNExample/encrypt* and *IRotNExample/decrypt* methods. Although it is not obvious from the definition of `<IRotNExample>`, these methods must take as their first parameter the instance of `<IRotN-implementation>` they operate on, and return as a first result a COM error code.

1. Add the following code to *RotNExample-server.dylan*.

```
define method IRotNExample/encrypt
  (this :: <IRotN-implementation>, pre :: <string>)
=> (result :: <HRESULT>, post :: <string>)
  if (instance?(this.IRotNExample/key, <integer>))
    let post = make(<string>, size: pre.size);
    for (char keyed-by index in pre)
      post[index] := rot-char-by-n(char, this.IRotNExample/key);
    end for;
    values($S-OK, post)
  else
    values($E-INVALIDARG, "")
  end if
```

```
end;

define method IRotNExample/decrypt
  (this :: <IRotN-implementation>, pre :: <string>)
=> (result :: <HRESULT>, post :: <string>)
  if (instance?(this.IRotNExample/key, <integer>))
    let post = make(<string>, size: pre.size);
    for (char keyed-by index in pre)
      post[index] := rot-char-by-n(char, -this.IRotNExample/key);
    end for;
    values($S-OK, post)
  else
    values($E-INVALIDARG, "")
  end if
end;
```

Note that this code is careful not to crash when *IRotNExample/key* is a `<machine-word>`. `$S-OK` represents success. `$E-INVALIDARG` is a generic failure representing some kind of invalid argument value.

The above method uses the *rot-char-by-n* function, which we must also add.

1. Add the following code to *RotNExample-server.dylan*.

```
define function rot-char-by-n
  (char :: <character>, n :: <integer>)
=> (r :: <character>)
  let char-as-int = as(<integer>, char);
  local method rot-if-in-range
    (lower :: <integer>, upper :: <integer>) => ()
    if (lower <= char-as-int & char-as-int <= upper)
      char-as-int := lower + modulo(char-as-int - lower + n,
                                    upper - lower + 1);
    end if;
  end method;
  rot-if-in-range(as(<integer>, 'a'), as(<integer>, 'z'));
  rot-if-in-range(as(<integer>, 'A'), as(<integer>, 'Z'));
  as(<character>, char-as-int)
end;
```

This function rotates alphabetic characters forward *n* positions, wrapping around if the character passes "Z". When *n* is 13, this implements the classic Rot13 cipher often used to hide offensive material on USENET.

In order to create our server, we must also create a COM class for it.

1. Add the following code to *RotNExample-server.dylan*.

You may want to copy the `define coclass` code from *stubs.dylan* in the *RotNExample-server-stubs* project and modify it.

```
define coclass $RotNExample-type-info
  name "RotNExample";
  uuid $RotNExample-class-id;
  default interface <IRotN-implementation>;
end coclass;
```

Now we simply have to add a Windows event loop as the main body of the server program.

1. Modify the `main` method in *RotNExample-server.dylan* to look like the following.

```
define method main () => ()
  if (OLE-util-register-only?())
```

```
    register-coclass($RotNExample-type-info,
                     "Harlequin.RotNExample");
  else
    let factory :: <class-factory>
      = make-object-factory($RotNExample-type-info);
    with-stack-structure (pmsg :: <PMSG>)
      while (GetMessage(pmsg, $NULL-HWND, 0, 0))
        TranslateMessage(pmsg);
        DispatchMessage(pmsg);
      end while;
    end with-stack-structure;
    revoke-registration(factory);
  end if;
end method main;
```

With this code in place, if the server is invoked from the command line with */RegServer* as an argument, `OLE-util-register-only?` will return `#t`. The call to `register-coclass` creates a type library (with extension .TLB) and registers the type library and the server itself in the Windows registry.

Note that the server provides no way to exit. We can make it exit whenever our interface object is destroyed. This is a little simplistic, since it does not correctly handle the case in which two servers are created, but it will suffice for our example.

1. Add the following code to *RotNExample-server.dylan*.

```
define method terminate (this :: <IRotN-implementation>) => ()
  next-method();
  PostQuitMessage(0); // Cause main event loop to terminate.
end;
```

The `PostQuitMessage` call causes the next call to `GetMessage` (in the main event loop) to return `#f`, and thus cause the program to exit.

1. Build the project with **Project > Build**.

During the build, you will be prompted for the location of the project file *RotNExample-server-stubs.hdp*.

Creating the client stubs library

Now we create a project for the client-side stubs of the encryption interface.

Choose **File > New...** from the main window.

1. Select **Project** and click **OK**.

The New Project wizard appears.

1. In the Project Type section, select "Interface to COM Type Library" and click **Next**.
2. Click **Browse...** and navigate to the Open Dylan examples folder.

The folder required is called Examples and is placed under the top-level Open Dylan folder.

It is usually *C:\Program Files\Open Dylan\Examples*.

1. Go to the *Documentation\RotNExample* subfolder and choose the *RotNExample.tlb* file.
2. Click **Next** to continue to the next page of the wizard.
3. Select "Dispatch client interfaces" and click **Next** to continue to the next page of the wizard.

4. Choose "Translate all" on the next page, so that both the RotNExample COM class and the IRotNExample interface are translated. Click **Next**.
5. Change the name of the project to *RotNExample-client-stubs*.
6. Choose a suitable Location for the project.
7. Make sure that the "Dynamic Link Library (DLL)" option is chosen in the Project Settings and Templates section.
8. Make sure that the "Include any available templates" option is *not* chosen.
9. Click **Next** to continue.

We are now at the Use Libraries page. We are only defining the stubs for the client, so we do not need any unusual libraries.

1. Choose the "Simple" option and click **Next**.
2. Choose the "Standard IO streams and string formatting" option from "IO and system support", and click **Next**.
3. Choose the "Win32 API" option from "GUI support", and click **Next**.

Note that the "OLE Automation" option on the "OLE Automation support" page is automatically selected. That is what we want.

1. Click **Next**.
2. Choose the "NoneÝ" option from "Database support" and click **Next**.

We are now at the final page of the New Project wizard.

1. Make any changes you want to the Source File Headers section.
2. Click **Finish**.

The new project appears.

As with the server stubs project, we have to build this project to make the type library tool run.

1. Build the project with **Project > Build**.

As before, files are added to the project. The *type-library-module.dylan* file serves the same purpose as with the server stubs. The difference is that *stubs.dylan* contains different code. It defines a dispatch-client class for the *IRotNExample* interface:

```
/* Dispatch interface: IRotNExample version 0.0
 * GUID: {822ED42A-3EB1-11D2-A3CA-0060B0572A7F}
 * Description: An example interface for Open Dylan's
 * Getting Started manual.
 */

define dispatch-client <IRotNExample>
  uuid "{822ED42A-3EB1-11D2-A3CA-0060B0572A7F}";
  property IRotNExample/key :: type-union(<integer>, <machine-word>), name: "key",
  disp-id: 12288;
  function IRotNExample/encrypt (arg-pre :: <string>) =>
    (arg-result :: <string>), name: "encrypt", disp-id: 24576;
  function IRotNExample/decrypt (arg-pre :: <string>) =>
    (arg-result :: <string>), name: "decrypt", disp-id: 24577;
end dispatch-client <IRotNExample>;
```

This defines a class `<IRotNExample>` which allows a client to use the described interface.

There is also generated code corresponding to the COM class RotNExample from the type library:

```
/* COM class: RotNExample version 0.0
 * GUID: {C44502DB-3EB1-11D2-A3CA-0060B0572A7F}
 * Description: Implementation of IRotNExample.
 */

define constant $RotNExample-class-id =
  as(<REFCLSID>, "{C44502DB-3EB1-11D2-A3CA-0060B0572A7F}");
define function make-RotNExample ()
=> (default-interface :: <IRotNExample>)
  let default-interface = make(<IRotNExample>,
                               class-id: $RotNExample-class-id);
  values(default-interface)
end function make-RotNExample;
```

This function creates an instance of the RotNExample COM class, and returns its default (and only) interface.

Creating the client

Now we create the actual client application.

Choose **File > New...** from the main window.

1. Select **Project** and click **OK**.

The New Project wizard appears.

1. In the Project Type section, select "Console Application (EXE)" and click **Next** to continue to the next wizard page.
2. Name the project *RotNExample-client*.
3. Choose a suitable Location for the project.
4. Make sure the "Include any available templates" option is *not* chosen.
5. Click **Next**.
6. Choose the Simple libraries option, and choose the "Standard IO streams and string formatting" and "OLE Automation" options.
7. Proceed to the last page of the wizard and click **Finish**.
8. In the new project, edit *library.dylan*, and add to the `define library` declaration the following line:

```
use RotNExample-client-stubs;
```

1. Add the same line to the `define module` declaration in *module.dylan*.

We now add code to make the client encrypt and decrypt a simple string with the default key of 13 and with the key set to 3.

Note: The remainder of this section of the example involves adding code to *RotNExample-client.dylan*. A version of this file with all the code we add in this section is available in the Open Dylan installation folder, under *Examples\Documentation\RotNExample\RotNExample-client.dylan*. You may want to copy that file into place in your project folder rather than typing code in.

---

1. Modify the `main` method in *RotNExample-client.dylan* to look like the following.

```
define method main () => ()
  with-ole
    format-out("Client connecting to server.\\n");
    let server = make-RotNExample();
    local method encrypt-and-decrypt () => ()
      let plaintext = "And he was going ooo-la, oooooo-la...";
      format-out("Plaintext is %=, encrypting.\\n", plaintext);
      let ciphertext = IRotNExample/encrypt(server, plaintext);
      format-out("Ciphertext is %=, decrypting.\\n", ciphertext);
      let decrypted = IRotNExample/decrypt(server, ciphertext);
      format-out("Decrypted text is %=.\\n", decrypted);
    end method;
    encrypt-and-decrypt();
    server.IRotNExample/key := 3;
    format-out("Set key to %d.\\n", server.IRotNExample/key);
    encrypt-and-decrypt();
    format-out("Client releasing server.\\n");
    release(server);
  end with-ole;
end method main;
```

The `with-ole` macro initializes OLE at entry and uninitializes it at exit.

1. Build the project with **Project > Build**.

During the build, you will be prompted for the location of the project file *RotNExample-client-stubs.hdp*.

Testing the client and server pair

The best way to test the client and server pair is from within the Open Dylan environment, so that we can use the debugger if either application fails.

First we must register the server with the system, so that COM knows where to find the server and what interfaces it supports. In order to do this, we must execute the server with the */RegServer* command line flag. This will cause the server's call `OLE-util-register-only?` to return `#t`, and the server to call `register-coclass`.

Open the RotNExample-server project and build it.

1. Select *Project* > *Settings...* in the RotNExample-server project window.

The Project Settings dialog appears.

1. On the Debug page, put */RegServer* in the Arguments field, and click *OK*.
2. Start the RotNExample-server application.

The server application registers itself and exits immediately. You can tell that the server has exited by watching the stop button in the project window become unavailable.

Now that the server is registered, it can be invoked by the client. But we are going to start the server manually in the environment before starting the client. That way, if the server fails, we can debug it in the environment.

First, however, we need to remove the */RegServer* argument from the project settings, so that the server can run normally.

1. Select **Project > Settings...** in the RotNExample-server project window.

The Project Settings dialog appears.

1. On the Debug page, remove */RegServer* from the Arguments field, and click **OK**.
2. Start RotNExample-server.

3. Start RotNExample-client.

The client should execute, and print something like this:

```
Client connecting to server.
Plaintext is "And he was going ooo-la, oooooo-la...", encrypting.
Ciphertext is "Naq ur jnf tbvat bbb-yn, bbbbbb-yn...",
decrypting.
Decrypted text is "And he was going ooo-la, oooooo-la...".
Set key to 3.
Plaintext is "And he was going ooo-la, oooooo-la...", encrypting.
Ciphertext is "Dqg kh zdv jrlqj rrr-od, rrrrrr-od...",
decrypting.
Decrypted text is "And he was going ooo-la, oooooo-la...".
Client releasing server.
```

Creating vtable and dual interfaces

The New Project wizard can generate custom (vtable) and dual (vtable and dispatch) COM interfaces. They are available on the wizard's stub-selection page, from the options "Custom (vtable) interfaces" and "Dual interfaces".

Because custom and dual interfaces generate both server and client interfaces, it is necessary to ensure that the names of the server and client interfaces do not collide. Thus when either of these is selected an additional page appears for specifying suffixes for the names of the generated interfaces.

Fig. 13.5: Custom and dual interface class suffix selection.

The server class suffix is appended to the name of the interface when generating the server class name. For example, if the server class suffix is *-server* then the server class for an interface *IBar* is named `<IBar-server>`.

The supplied client class suffix is appended to the name of the interface to generate the client class name. If the "Generate client classes with suffix" box is not checked, no client classes are generated, and client methods specialize on `<C-Interface>` instead.

The type library tool and specification files

The type library tool is invoked any time you build a project which includes a a type library tool specification, or .SPEC, file. This is a text file, where the first line must be:

```
Origin: COM-type-library
```

This line identifies that the type library tool should be used to process the file. When it runs, the type library tool will regenerate the module and stub files when they do not exist, or if the .SPEC file has been modified more recently then the last build.

Type library tool specification files can contain the following keywords:

Type-library: SPEC file keyword

```
    Type-library: *typelibrary-path*
```

Required. Specifies the pathname of the type library to translate. If *typelibrary-path* is a relative path, it is considered to be relative to the location of the specification file.

Module-file: SPEC file keyword

```
Module-file: *module-file-to-generate*
```

Required. Specifies the pathname of the module file to generate. The file will be created, if necessary, and added to the project, if necessary.

Module: SPEC file keyword

```
Module: *module-name-to-generate*
```

Required. The name of the module definition to generate. This module definition is placed in the file *module-file-to-generate*.

Stub-file: SPEC file keyword

```
Stub-file: *stub-file-to-generate*
```

Required. Specifies the pathname of the Dylan source file to generate. This file will be created, if necessary, and added to the project, if necessary.

Generate: SPEC file keyword

```
Generate: *type-of-stubs*
```

Required. Determines what type of stubs are generated in the file specified using *Stub-file:*. Possible values of *type-of-stubs* are:

dispatch-clients Dispatch client code is generated, using `define dispatch-client`.

dispatch-servers Dispatch server code is generated, using `define dispatch-interface`.

vtable-interfaces Custom (vtable) interfaces are generated, using `define vtable-interface`. The names of server interfaces are affected by the value of *Server-suffix:* (below). The names of client interfaces and whether client interfaces are generated are affected by the presence and the value of *Client-suffix:* (below).

dual-interfaces Dual (vtable and dispatch) interfaces are generated, using `define vtable-interface`. The names of server interfaces are affected by the value of *Server-suffix:*. The names of client interfaces and whether client interfaces are generated are affected by the presence and the value of *Client-suffix:*.

Server-suffix: SPEC file keyword

```
Server-suffix: *server-suffix*
```

Optional. Only meaningful when *Generate:* 's *type-of-stubs* argument is *vtable-interfaces* or *dual-interfaces*. Specifies a suffix which is appended to generated server interface names. If no value is provided or the *Server-suffix:* keyword is omitted then no suffix is appended.

Client-suffix: SPEC file keyword

```
Client-suffix: *client-suffix*
```

Optional. Only meaningful when *Generate:* 's *type-of-stubs* argument is *vtable-interfaces* or *dual-interfaces*. Specifies a suffix which is appended to generated client interface names. If the *Client-suffix:* keyword is omitted then client classes are not generated (the client-class clause is not provided to the define vtable-interfaces or define dual-interfaces macro invocation). To generate client classes but not append a suffix, an empty value must be provided to the *Client-suffix:* keyword. For example:

```
Client-suffix:
```

Interfaces: SPEC file keyword

```
Interfaces: *interfaces-and-coclasses-to-translate*
```

Optional. If provided, specifies the names of the interfaces and COM classes to translate. If not provided, all interfaces and COM classes are translated. All interfaces or COM classes after the first listed should be provided on a new line, preceded by a tab or spaces. For example:

```
Interfaces: IInterfaceOne
            InterfaceOneCoclass
            IInterfaceTwo
            InterfaceTwoCoclass
```

CHAPTER

FOURTEEN

INDICES AND TABLES

- genindex
- search

Symbols

A

B

C

D

F

I

L

N

O

P

R

S

T

W