Photran Developer's Guide

Part II: Specialized Topics

December 6, 2011 Revision

N. Chen
J. Overbey

Contents

1 Interactions with CDT **2**

1.1 Introduction . . . 2

1.2 C Projects vs. Photran Projects . . . 3

1.3 CDT Terminology . . . 3

1.4 The Model . . . 4

1.5 Reusing UI Elements . . . 6

1.6 The CDT Debugger and `gdb` . . . 6

2 The Abstract Syntax Tree and Virtual Program Graph **7**

2.1 How to Get Acquainted with the Program Representation . . . 7

2.1.1 Visualizing ASTs . . . 7

2.1.2 Visually Resolving Bindings . . . 7

2.1.3 Visualizing Enclosing Scopes . . . 8

2.1.4 Visualizing Definitions . . . 8

2.2 Abstract Syntax Trees . . . 8

2.2.1 Simple AST Example . . . 8

2.2.2 AST Structure for DO-Loops . . . 9

2.3 Virtual Program Graph . . . 10

2.3.1 Acquiring and Releasing ASTs . . . 10

2.3.2 Scope and Binding Analysis . . . 11

2.3.3 Scheduling and (Avoiding) Concurrent Access . . . 13

2.4 Common Tasks . . . 13

3 Refactorings **14**

3.1 Introduction . . . 14

3.2 Overview . . . 14

3.3 Getting Started . . . 15

3.4 How to Create a Refactoring . . . 15

3.5 Changing Source Code: AST Rewriting . . . 16

3.5.1 Whitetext Affixes . . . 16

3.5.2 Mutable AST . . . 16

3.5.3 Modifying, Moving, Removing, and Replacing AST Nodes . . . 17

3.5.4 Inserting new AST Nodes . . . 17

3.5.5 Committing Changes . . . 17

3.6 Caveats . . . 17

3.6.1 *Token* or *TokenRef*? . . . 18

3.7 Testing a Refactoring . . . 18

4 The Fortran Editor **20**

4.1 Fortran Text Editor . . . 20

4.2 Contributed *SourceViewerConfiguration* . . . 20

4.3 Fortran Editor Tasks: VPG & AST Tasks . . . 20

A Adding New Fortran Syntax **22**

B Regenerating the Help Plug-in **23**

C Manual Tests for the GNU Managed Build Toolchain **24**

D Release and Deployment Procedure **26**

D.1 Preparing for a Release Build . . . 26

D.2 After the Release Has Been Built . . . 27

Chapter 1

Interactions with CDT

Revision: $Id: cha-cdt.ltx-inc,v 1.1 2010/05/21 20:12:20 joverbey Exp - based on 2008/08/08 nchen

1.1 Introduction

The C/C++ Development Tools (CDT)[1] provide support for C and C++ development in Eclipse. CDT uses *make* to compile projects, and it includes an integrated debugger which is actually a graphical interface to *gdb*.

In 2006, we contributed a patch to CDT[2] which provides an extension point allowing its core infrastructure to support languages other than C and C++. This is appropriate for any language for which code is compiled using *make* and debugged using *gdb*.[3]

Photran builds upon the CDT by

- plugging into this extension point (org.eclipse.cdt.core.language). This allows CDT to recognize Fortran source files as "valid" source files in C/C++ projects, and it allows Photran to provide the outline structure for these files which is displayed in the Navigator and the Outline view (this structure is called the "model" and is discussed below).
- subclassing many of CDT's user interface elements (or copying and modifying them when necessary) to provide an IDE for Fortran that looks and works similarly to CDT.
- contributing *error parsers* which allow CDT to recognize error messages from many popular Fortran compilers and display them in the Problems view.
- contributing new project templates and toolchains, which are shown in the New C/C++ Project dialog (and, of course, the New Fortran Project dialog).

[1] See http://www.eclipse.org/cdt/

[2] See https://bugs.eclipse.org/133386

[3] The exact mechanism is described in J. Overbey and C. Rasmussen, "Instant IDEs: Supporting New Languages in the CDT," *Eclipse Technology eXchange Workshop at OOPSLA 2005 (eTX 2005),* San Diego, CA, October 17, 2005, and a tutorial on the topic was given at EclipseCon 2006. The details (and code) are now outdated, but the concepts are still applicable.

1.2 C Projects vs. Photran Projects

Prior to version 6.0, C projects and Fortran projects were treated identically. As of version 6.0, there is an important distinction:

- every Fortran project is also a C project, but
- not every C project is a Fortran project.

In technical terms, Fortran projects are C projects that also have a Fortran nature. They must be C projects in order to be "recognized" by CDT, but they must also have the Fortran nature so that Photran can display project properties for them and index them without affecting C/C++ projects that do not contain Fortran code.

1.3 CDT Terminology

The following are CDT terms that will be used extensively when discussing Photran.

- There are two types of projects in CDT:[4]
  - **Standard Make projects** require users to supply their own Makefile, typically with targets "clean" and "all." CDT/Photran cleans and builds the project by running `make`. ("Standard Make" is actually old terminology; recent versions of CDT call these "Makefile Projects." They are created by choosing Makefile Project > Empty Project and selecting "– Other Toolchain –" in the New C/C++ Project dialog.)
  - **Managed Make projects** are similar to standard make projects, except that CDT/Photran automatically generates a Makefile and edits the Makefile automatically when source files are added to or removed from the project. The **Managed Build System** (MBS) is the part of CDT that handles all of this.
- **Binary parsers** are able to detect whether a file is a legal executable for a platform (and extract other information from it). CDT provides binary parsers for Windows (PE), Linux (ELF), Mac OS X (Mach O), and others. Photran does not provide any additional binary parsers.
- **Error parsers** are provided for many compilers. CDT provides a gcc error parser, for example. Photran provides error parsers for Lahey Fortran, F, g95, and others. Essentially, error parsers scan the output of `make` for error messages for their associated compiler. When they see an error message that they recognize, they extract the filename, line number, and error message, and use it to populate the Problems View. See Appendix **??** for an example on how to create an error parser.
- CDT keeps a **model** of all of the files in a project. The model is essentially a tree of **elements**, which all inherit the (CDT Core) interface `ICElement`. It is described in the next section.

[4] In earlier versions of CDT, these were actually distinct. CDT overhauled their build system in version 4(?), and these are now treated uniformly in implementation. Nevertheless, this distinction was made frequently in historical discussions about CDT, and it is important to be aware of. It is also a useful distinction to make from the user's perspective, since the idea of hand-writing a makefile seems to intimidate many Fortran programmers.

1.4 The Model

The Fortran Projects view in Photran is essentially a visualization of the CDT's *model*, a tree data structure describing the contents of all Fortran Projects in the workspace as well as the high-level contents (functions, aggregate types, etc.) of source files.

Alain Magloire (CDT) described the model, A.K.A. the `ICElement` hierarchy, in the thread "Patch to create ICoreModel interface" on the cdt-dev mailing list (April 1, 2005):

```
So I'll explain a little about the ICElement and what we get out of it for
C/C++.

The ICElement hierarchy can be separated in two:
(1) - how the Model views the world/resources (all classes above ITranslationUnit)
(2) - how the Model views the world/language (all classes below ITranslationUnit).

How we(C/C++) view the resources:
- ICModel  --> [root of the model]
  - ICProject --> [IProject with special attributes/natures]
    - ISourceRoot --> [Folder with a special attribute]
      - ITranslationUnit --> [IFile with special attributes, e.g. extensions *.c]
      - IBinary --> [IFile with special attributes, elf signature, coff etc]
      - IArchive --> [IFile with special attributes, "<ar>" signature]
      - ICContainer -> [folder]

There are also some special helper classes
  - ILibraryReference [external files use in linking ex:libsocket.so, libm.a, ...]
  - IIncludeReference [external paths use in preprocessing i.e. /usr/include, ...]
  - IBinaryContainer [virtual containers regrouping all the binaries found
    in the project]

This model of the resources gives advantages:
- navigation of the binaries,
- navigation of the include files not part of the workspace (stdio.h, socket.h, etc)
- adding breakpoints
- search
- contribution on the objects
etc.....

[...]

(2) How we view the language.

Lets be clear this is only a simple/partial/incomplete view of the language. For
example, we do not drill down in blocks, there are no statements(if/else
conditions) etc .... For a complete interface/view of the language, clients
should use the __AST__ interface.
```

From another of Alain's posts in that thread:

```
Lets make sure we are on the same length about the ICElement hierarchy.
It was created for a few reasons:

- To provide a simpler layer to the AST.  The AST interface is too complex
  to handle in most UI tasks.
- To provide objects for UI contributions/actions.
- The glue for the Working copies in the Editor(CEditor), IWorkingCopy class
- The interface for changed events.
- ...

Basically it was created for the UI needs: Outliner, Object action contributions,
C/C++ Project view and more.

The CoreModel uses information taken from:
- the Binary Parser(Elf, Coff, ..)
- the source Parser(AST parser)
- the IPathEntry classes
- the workspace resource tree
- The ResolverModel (*.c, *.cc extensions), ...

to build the hierarchy.
```

The CDT model should **not** be confused with the Abstract Syntax Tree (AST) that is discussed in Chapter 2. They are **not** identical. It is helpful to think of the CDT model as containing a *partial/simplified view* of a program's structure (representing only the high-level/organizational elements in the source code: program units, subprogram declarations, etc.) **as well as** a model of the current workspace resources (Fortran projects, Fortran source files, binary executables). In other words, the CDT model knows about the resources and the organizational units in the source code. The AST, on the other hand, completely models *everything* in a single source ilfe—including low-level elements like individual statements and expressions—but knows nothing about projects, folders, etc.

By conforming to the CDT model, Photran is able to reuse various UI elements for *free*. For instance, the Outline View for Photran is managed by CDT; Photran just needs to provide a CDT-compatible model to represent its project and source files.

The `FortranLanguage` class (in the org.eclipse.photran.cdtinterface project) is responsible for initializing concrete classes that will build up the model that CDT expects.

There are **two** options for creating suitable *model builders*:

1. The org.eclipse.photran.cdtinterface plug-in project defines the `org.eclipse.photran.cdtinterface.modelbuilder` extension point that other plug-ins can extend. Plug-ins extending that extension point are responsible for providing a suitable model builder. Using this option, it is possible to have multiple model builders. The model builder to use can be selected in the workspace preferences (under Fortran > CDT Interface).
2. If there are no plug-ins that extend the `org.eclipse.photran.cdtinterface.modelbuilder` extension point, then Photran falls back on a default implementation provided by the `EmptyFortranModelBuilder` class (which, not surprisingly, builds an empty model).

The Photran VPG (see Section 2.3) inside the org.eclipse.photran.cdtinterface.vpg project uses the first option to contribute a model builder. The relevant classes are under the `org.eclipse.photran.internal.core.model` package (notably, `FortranModelBuilder`.)

As mentioned in the post by Alain, all model elements must implement the `ICElement` interface for CDT to recognize them. In Photran, the `FortranElement` class implements the `ICElement` interface and serves as the base class for all Fortran elements such as subroutines, functions, modules, variables, etc. Each subclass of `FortranElement` corresponds to an element that can be displayed in the Outline View.

1.5 Reusing UI Elements

Various UI elements in Photran are also reused from the CDT.

- Many elements are reused directly.
  - Photran frequently instantiates or references classes from CDT and uses them as-is. For example, the `FortranProjectWizard` uses CDT's `ICProjectDescription`.
  - It is also common for plugin.xml files in Photran to reference contributions from CDT. For example, the XML declaration for the Fortran perspective includes the line
    ```
    <actionSet id="org.eclipse.cdt.make.ui.makeTargetActionSet"/>
    ```
    which adds some build-related CDT actions to the menus and toolbars in the Fortran perspective.
  - In many cases, the XML describing a particular element in Photran is essentially the same as the corresponding XML in CDT, except the name and/or icon is changed. This is the case for, e.g., the *Local Fortran Application* launch configuration.
- Many elements are reused through subclassing. For example, Photran's `NewSourceFolderCreationWizard` subclasses an equivalent class in CDT but overrides a method in order to change the title and icon for the wizard.
- As a last resort, some parts of Photran are copied from CDT and then modified as necessary. This is the case with the `FortranPerspectiveFactory` class, for example: There was no way to accomplish the desired effect through subclassing (without modifying CDT), so we (unfortunately) had to copy a class from CDT and then modify it appropriately for Photran.

1.6 The CDT Debugger and `gdb`

Currently, Photran re-uses the CDT debugger as-is and does not contribute any enhancements to it. Here is a brief summary of how the debugger works:

- The so-called CDT debugger is actually just a graphical interface to `gdb`, or more specifically to `gdb/mi`. So, if something doesn't appear to work, it is advisable to try it in `gdb` directly or to use another `gdb`-based tool such as DDD.
- The debugger UI "contributes" breakpoint markers and actions to the editor. The "set breakpoint" action, and the breakpoint markers that appear in the ruler of the CDT (and Photran) editors are handled **entirely** by the CDT debug UI; there is no code for this in Photran. The "set breakpoint" action is enabled by calling setRulerContextMenuId("#CEditorRulerContext"); in the `AbstractFortranEditor` constructor.
- `gdb` reads debug symbols from the executable it is debugging. That is how it knows what line it's on, what file to open, etc. Photran has *nothing* to do with this: These symbols are written entirely by the compiler. Moreover, the compiler determines what shows up in the Variables View. If the debugger views seem to be a mess, it is the compiler's fault, not Photran's.

Chapter 2

The Abstract Syntax Tree and Virtual Program Graph

Revision: $Id: cha-ast-vpg.ltx-inc,v 1.1 2010/05/21 20:12:20 joverbey Exp - based on 2008/08/08 nchen

2.1 How to Get Acquainted with the Program Representation

These features work only on files that are located inside a Fortran project with analysis and refactoring enabled. See the Photran user documentation (Photran Advanced Features Manual) for instructions.

2.1.1 Visualizing ASTs

When Photran parses a file, it produces an abstract syntax tree (AST). This is the central data structure used when implementing refactorings, program analyses, etc.

Photran can display its abstract syntax tree (AST) for a file in place of the ordinary Outline view. This behavior can be enabled from the Fortran workspace preferences:

- Click on Window > Preferences in Windows/Linux, or Eclipse > Preferences in Mac OS X.
- Select "Fortran" on the left hand side of the preference dialog (do not expand it).
- Select "(Debugging) Show entire abstract syntax tree rather than Outline view"

Clicking on a `Token` in the Outline view will move the cursor to that construct's position in the source file.

Figure 2.1 is an example of this display. (It is explained further in Section 2.2.)

2.1.2 Visually Resolving Bindings

Each use of an identifier in a Fortran program corresponds to a declaration. For example, the use of the variable *count* in `print *, count` might correspond to a declaration like `integer :: count`. We say that the use of the

identifier is **bound** to its declaration, and determining the declaration that corresponds to a particular use is called **resolving** the name binding.

To visualize what declaration a particular variable use is bound to, click on an identifier in a Fortran editor (position the cursor over it), and press F3 (or click Navigate > Open Declaration, or right-click and choose Open Declaration.) The binding will be resolved and the declaration highlighted. If there are multiple bindings, a pop-up window will open and one can be selected. If the identifier is bound to a declaration in a module defined in a different file, an editor will be opened on that file.

2.1.3 Visualizing Enclosing Scopes

Every declaration in a Fortran program exists in a particular *lexical scope.* For example, if a subroutine definition includes a variable declaration like `integer :: i`, then that variable is only visible within the subroutine, and that subroutine is its *enclosing scope.* In the AST, the subroutine is represented by a *ASTSubroutineSubprogramNode*, a class which inherits from *ScopingNode*, and it would an the ancestor of the *ASTTypeDeclarationStmtNode* representing the variable declaration.

To view the enclosing scope for a particular token in the AST, click on any token in the Fortran editor, and click Refactor > (Debugging) > Select Enclosing Scope. The entire range of source text corresponding to that token's enclosing `ScopingNode` will be highlighted.

2.1.4 Visualizing Definitions

For every declaration in a Fortran program, Photran maintains a *Definition* object which summarizes the information available about that symbol. For example, by invoking methods on the *Definition* for a symbol, you could determine that the symbol is a local variable named *matrix* and that it is a two-dimensional, allocatable array. Of course, you could figure this out by traversing the AST, but that can be very tedious (and expensive).[1]

To get a sense of what symbols have *Definition* objects (these are things like variable declarations, subprogram declarations, common block names, etc.), open a file in the Fortran editor, and click Refactor > (Debugging) > Display Symbol Table for Current File. Indentation shows scope nesting, and each line summarizes the information in a `Definition` object.

2.2 Abstract Syntax Trees

2.2.1 Simple AST Example

The Fortran grammar is very lengthy, containing hundreds of rules. Even the simplest Fortran program has a fairly non-trivial AST. For instance this simple Fortran program:

```
1 program main
2     integer a
3 end
```

[1] Actually, Photran *does* have to traverse the AST to figure this out, but it only does this once per file, and then it saves the *Definition* to disk (in the "VPG database"). When you ask Photran for a *Definition*, it simply loads the information from this database.

generates the AST shown in Figure 2.1.

Figure 2.1 AST for simple Fortran program as viewed through the Outline View

Fortunately, it is not necessary to know every construct in the grammar. For most refactoring and program analysis tasks, it is sufficient to rely on the information that the VPG provides (e.g., *Definition* objects) and to construct a Visitor to visit only the nodes of interest and "collect" the information that is required. To get a sense of what AST nodes are involved in the particular task you want to accomplish, we recommend enabling AST visualization (described above), writing some sample programs that exercise the relevant parts of the Fortran grammar, and then observing the AST that is displayed.

2.2.2 AST Structure for DO-Loops

Due to a deficiency in the parser, DO-constructs are not recognized as a single construct; DO and END DO statements are recognized as ordinary statements alongside the statements comprising their body. There is a package in the core.vpg plug-in called org.eclipse.photran.internal.core.analysis.loops which provides machinery to fix this, giving DO-loops a "proper" AST structure.

If you call `LoopReplacer.replaceAllLoopsIn(ast)`, it will identify all of the new-style DO-loops and change them to `ASTProperLoopConstructNode`s, which a more natural structure, with a loop header, body, and END DO statement. Once this is done, visitors must be implemented by subclassing one of the Visitor classes *in the org.eclipse.photran.internal.core.analysis.loops package;* these have a callback method to handle `ASTProperLoopConstructNode`s.

The AST visualization (described above) shows the AST *after* `LoopReplacer.replaceAllLoopsIn(ast)` has been invoked.

2.3 Virtual Program Graph

In Photran, it is *almost* never necessary to call the lexer, parser, or analysis components directly. Instead, Photran uses a **virtual program graph**, or VPG, which provides the façade of a whole-program abstract syntax tree (AST) with embedded analysis information.

To the programmer building a refactoring, the VPG appears quite simple.

1. When the programmer requires an AST, the programmer asks the VPG to provide it. He does not parse the file directly.
2. Methods on (certain) AST nodes provide name binding information, control flow information, source/AST rewriting, etc. For example, the programmer can ask an AST node for its control flow successors, or he can tell it to remove itself from the AST, to reindent itself, etc.[2]
3. The VPG maintains a database containing the analysis information (e.g., what names are references to what other names). Analyses are run when the user saves a file, and the result is saved to the database. Then, when requests are made for analysis information (e.g., asking for all of the references to a particular name), it can simply be loaded from the database. It is the programmer's responsibility to make sure that the database is up-to-date before he attempts to access it, and that any task that requires database information is scheduled so that it will not attempt to read from the database while analysis information is being updated.

Photran's VPG is implemented by the class `PhotranVPG`. This is a *singleton* object whose instance is available via `PhotranVPG.getInstance()`. The remaining subsections describe how the above tasks are implemented in Photran.

2.3.1 Acquiring and Releasing ASTs

ASTs are retrieved by invoking either of these methods:

```
public IFortranAST acquireTransientAST(IFile file)
public IFortranAST acquirePermanentAST(IFile file)
```

The returned object is an `IFortranAST`, an object which has a method for returning the root node of the AST as well as methods to quickly locate tokens in the AST by offset or line information. A *transient AST* can be garbage collected as soon as references to any of its nodes disappear. A *permanent AST* will be explicitly kept in memory until a call is made to either of the following methods:

Listing 2.1 Releasing the Fortran AST

```
public void releaseAST(IFile file)
public void releaseAllASTs()
```

Often, it is better to acquire a transient AST and rely on the garbage collector to reclaim the memory once we are done using it. However, there are times when acquiring a permanent AST would be more beneficial performance-wise. For instance, if we will be using the same AST multiple times during a refactoring, it would be better to just acquire

[2] Control flow analysis is not (yet) implemented in Photran. But, in theory, this is how it should work...

a permanent AST. This prevents the garbage collector from reclaiming the memory midway through the refactoring once all references to the AST have been invalidated. While it is always possible to reacquire the same AST, doing so can be an expensive operation since it requires *lexing*, *parsing* **and** finally *reconstructing* the AST from scratch.

Only one AST for a particular file is in memory at any particular point in time, so successive requests for the same `IFile` will return the same (pointer-identical) AST until the AST is released (permanent) or garbage collected (transient).

The `acquireTransientAST` and `acquirePermanentAST` methods return an object implementing `IFortranAST`. This interface has several methods, notably including the following:

- The `getRoot` method returns the root of the AST, while the `find...` methods provide efficient means to search for tokens based on their lexical positioning in the source code.
- The `accept` method allows an external visitor to traverse the AST. This method is usually used when it is necessary to "collect" information about certain nodes.
- Because `IFortranAST` extends the `Iterable` interface, it is possible to use the *foreach* loop to conveniently iterate through all the tokens in the AST e.g.
  for (Token token : **new** IterableWrapper<Token>(ast))

2.3.2 Scope and Binding Analysis

Currently, the only semantic analysis performed by Photran is binding analysis: mapping *identifiers* to their *declarations*. Compilers usually do this using symbol tables but Photran uses a more IDE/refactoring-based approach.

Certain nodes in a Fortran AST represent a lexical scope. All of these nodes are declared as subclasses of `ScopingNode`:

- ASTBlockDataSubprogramNode
- ASTDerivedTypeDefNode
- ASTExecutableProgramNode
- ASTFunctionSubprogramNode
- ASTInterfaceBlockNode[3]
- ASTMainProgramNode
- ASTModuleNode
- ASTSubroutineSubprogramNode

Each of the subclasses of `ScopingNode` represents a scoping unit in Fortran. The `ScopingNode` class has several public methods that provide information about a scope. For example, one can retrieve a list of all of the symbols declared in that scope; retrieve information about its `IMPLICIT` specification; find its header statement (e.g. a `FUNCTION` or `PROGRAM` statement); and so forth.

The enclosing scope of a `Token` can be retrieved by calling the following method on the `Token` object:

[3] An interface block defines a nested scope only if it is a named interface.Anonymous (unnamed) interfaces provide signatures for subprograms in their enclosing scope.

```
public ScopingNode getEnclosingScope()
```

Identifier tokens (`Tokens` for which `token.getTerminal() == Terminal.T_IDENT`), which represent functions, variables, etc. in the Fortran grammar, are *bound* to a declaration[4]. Although, ideally, every identifier will be bound to exactly one declaration, this is not always the case: the programmer may have written incorrect code, or Photran may not have enough information to resolve the binding uniquely). So the `resolveBinding` method returns a *list* of `Definition` objects:

```
public List<Definition> resolveBinding()
```

A `Definition` object contains many public methods which provide a wealth of information. From a `Definition` object, it is possible to get a list of all the references to a particular declaration (using `findAllReferences`) and where that particular declaration is located in the source code (using `getTokenRef`). Both of these methods return a `PhotranTokenRef` object. See Section 3.6.1 for a comparison between `Token` and `TokenRef`.

Obtaining the `Definition` of a variable

If you have a reference to the `Token` object of that variable (for instance through iterating over all `Tokens` in the current Fortran AST) then use:

```
// myToken is the reference to that variable
List<Definition> bindings = myToken.resolveBinding();

if(bindings.size() == 0)
  throw new Exception(myToken.getText() + " is not declared");
else if (bindings.size() > 1)
  throw new Exception(myToken.getText() + " is an ambiguous reference");

Definition definition = bindings.get(0);
```

If you do **not** have a reference to a `Token` but you know the name of the identifier, you can first construct a *hypothetical* `Token` representing an identifier and search for that in a *particular* `ScopingNode` (possibly obtained by calling the static method `ScopingNode.getEnclosingScope(IASTNode node)`).

```
Token myToken = new Token(Terminal.T_IDENT, "myNameOfIdentifier");
List<PhotranTokenRef> definitions = myScopingNode.manuallyResolve(myToken);
```

If you want to search for the identifier in **all** `ScopingNodes` for the current source file, then retrieve all the `ScopingNodes` and manually iterate through each one. Remember that the root of the AST is a `ScopingNode` and you may obtain the root of the AST through the `getRoot` method declared in `IFortranAST`.

```
List<ScopingNode> scopes = myRoot.getAllContainedScopes();

for (ScopingNode scopingNode : scopes)
{
  // search through each ScopingNode
}
```

[4]The introduction to VPGs earlier in this chapter (URL above) provides an example visually.

Examples in `FortranEditorASTActionDelegate` subclasses

The following subclasses of `FortranEditorASTActionDelegate` all contain short working examples of how to use the binding analysis API in Photran:

- DisplaySymbolTable
- FindAllDeclarationsInScope
- OpenDeclaration
- SelectEnclosingScope

2.3.3 Scheduling and (Avoiding) Concurrent Access

It is important to note that, because `PhotranVPG` is a singleton object, it may not be accessed concurrently by multiple threads.

Most actions that require an AST will be subclasses of `FortranEditorActionDelegate`. (All refactoring actions in Photran are descendants of `FortranEditorActionDelegate`, for example.) These are always scheduled in a way that avoids this problem.

Otherwise, the thread must either be scheduled using a `VPGSchedulingRule` or it must lock the entire workspace. See `EclipseVPG#queueJobToEnsureVPGIsUpToDate` as an example on how to use the `VPGSchedulingRule` and `FortranEditorActionDelegate#run` as an example of how to lock the entire workspace.

As a guideline, contributors who are interested in accessing the VPG should consider structuring their contributions as descendants of `FortranEditorActionDelegate`. However, if that approach is not feasible, then they should consider using `VPGSchedulingRule` before resorting to locking the entire workspace.

2.4 Common Tasks

In the org.eclipse.photran.internal.core.refactoring package, there is a class called *_AST_VPG_HOWTO* which contains several small "snippets" illustrating how to perform common tasks with the AST/VPG. This includes how to traverse the AST using a visitor, how to find nodes of a particular type, how to find all of the references to a particular name, etc.

Chapter 3

Refactorings

Revision: $Id: cha-refactoring.ltx-inc,v 1.3 2010/06/02 21:25:28 joverbey Exp - based on 2008/08/08 nchen

3.1 Introduction

A refactoring is a program transformation to improve the quality of the source code by making it easier to understand and modify. A refactoring is a special kind of transformation because it preserves the *observable behavior* of your program – it neither removes nor adds any functionality.[1].

The main purpose of writing Photran was to create a refactoring tool for Fortran. The Eclipse Platform provides some language-neutral components that allow Photran's refactorings to resemble the refactorings in the Java Development Tools, at least in terms of their user interface. However, the *language infrastructure* for refactoring—the program representation and the APIs for manipulating Fortran source code—are all unique to Photran.

Most of the code in a refactoring is dedicated to program analysis and source code transformation; the APIs for doing this were described in the previous chapter on the AST and VPG. In this chapter, we describe how automated refactorings are structured and how they can be added to the Eclipse API.

3.2 Overview

This section is excerpted from M. Méndez, J. Overbey, A. Garrido, F. Tinetti, and R. Johnson, "A Catalog and Classification of Fortran Refactorings" (2010).

One of Photran's design objectives has been to make adding new refactorings as painless as possible. Often, it is possible to add a new refactoring by implementing a single Java class and adding one line of XML to a configuration file.

Photran divides refactorings into two categories. An **editor-based refactoring** requires the user to select part of a Fortran program (for example, he may select the header of a do loop) in a text editor in order to initiate the refactoring. A **resource refactoring** applies to entire files; the user can select several files at once, or even entire folders or projects, and the refactoring will be applied to all of the selected Fortran source files at once.

To create a new refactoring, the developer must decide whether it will be an editor-based refactoring or a resource

[1] For more information see *Refactoring: Improving the Design of Existing Code*

refactoring. Photran provides different superclasses for each. The developer then creates a concrete subclass and adds a line of XML to a configuration file to make Photran aware of the new refactoring. The concrete subclass must define methods which:

- provide the name of the refactoring. This becomes its label in the Refactor menu, and it is also used to describe the refactoring in the Edit > Undo menu item and in other user interface elements.
- check initial preconditions. These are usually simple checks which verify that the user selected the correct construct in the editor, that the file is not read-only, etc.
- acquire user input. For example, a refactoring to add a parameter to a subprogram must ask the user to supply the new parameter's name and type.
- check final preconditions. These validate user input and perform any additional checks necessary to ensure that the transformation can be performed, the resulting code will compile, and it will retain the behavior of the original program.
- perform the transformation. Once all preconditions have been checked, this method determines what files will be changed, and how.

Thanks to Java's reflective facilities, much of the user interface for a refactoring comes "for free." Photran automatically adds the refactoring to the appropriate parts of the Eclipse user interface, and it provides a wizard-style dialog box which allows the user to interact with the refactoring. This dialog includes a *diff*-like preview, which allows the user to see what changes the refactoring will make before committing it.

3.3 Getting Started

Perhaps the best way to create a new refactoring is to start with an existing refactoring and modify it to do what you want. The following refactorings are very simple and should be used as starting points:

- *InterchangeLoopsRefactoring* – An editor-based refactoring that swaps the headers of two perfectly-nested `DO`-loops
- *RepObsOpersRefactoring* – A resource refactoring that replaces old-style operators like `.LT.` and `.EQ.` with their newer, symbolic counterparts (i.e., < and ==).
- *KeywordCaseRefactoring* – A resource refactoring that changes all keywords to either upper- or lower-case.

All of these refactorings make fairly simple replacements in an AST. A slightly less trivial example is *IntroImplicitNoneRefactoring*, which illustrates how to insert new code into an AST.

Hint: *To find any of these classes quickly, click on Navigate > Open Type in the Java perspective, and start typing the class name.*

3.4 How to Create a Refactoring

1. Decide whether your refactoring will be an editor-based refactoring or a resource refactoring, and then create a new subclass of either *FortranEditorRefactoring* or *FortranResourceRefactoring*.[2] Use one of the simpler

[2]Both of these classes inherit from of an Eclipse Platform class called *Refactoring*. The *Refactoring* class is provided by a Platform component called the Eclipse Language ToolKit, or LTK. The LTK is a language-neutral API for supporting refactorings in the Eclipse environment. It

refactorings listed above as a starting point, and look at the JavaDoc for each method you must override.

2. Add the refactoring to the `plugin.xml` file. Be sure that you give the correct fully-qualified name for your refactoring class, and be sure you correctly identify it as either a resource refactoring or an editor refactoring based on its superclass!

3. When you run Photran, your refactoring should now be visible in the user interface.

Refactoring classes inherit a large number of `protected` utility methods common among refactorings, such as a method to determine if a token is a uniquely-bound identifier, a method to parse fragments of code that are not complete programs, and a *fail* method which is simply shorthand for throwing a *PreconditionFailure*. Before writing your own utility methods, use content assist to scan through the list of methods available, or look at a refactoring with similar preconditions to determine if the functionality you need already exists.

3.5 Changing Source Code: AST Rewriting

Instead of manipulating the text in the source files directly, Photran's refactorings modify source code by manipulating the program's abstract syntax tree (AST), and then reproducing source code from the modified AST. The previous chapter provided a general introduction to Photran's AST. (If you have not read it, please do so before continuing.) The remainder of this section will discuss how the AST is used in refactorings.

3.5.1 Whitetext Affixes

Photran's AST is *concretized* as described in J. Overbey and R. Johnson, "Generating Rewritable Abstract Syntax Trees" (SLE 2008). Its API appears to be that of an ordinary AST, but in fact every token returned by the lexer is actually stored in the AST. Moreover, every token has whitetext (comments, whitespace, line continuations, etc.) affixed to it.

In general, all of the comments, whitespace, etc. are affixed to the *following* token. (This works well in Photran since it treats a newline as its own token.) This whitetext can be retrieved by invoking `token.getWhiteBefore()`. However, if there is whitetext at the *end* of a file, it is affixed to the *last* token in the file and can be retrieved by invoking `token.getWhiteAfter()`.

3.5.2 Mutable AST

Photran's AST is *mutable.* When you want to change the source code produced by the AST, you modify the AST itself. That is, you add, remove, or replace parts of the AST as necessary to effect the change you want. (Note that this is *different* from the Eclipse JDT and CDT, which use an *ASTRewriter* to *describe* source code changes in terms of AST nodes but do not actually modify the AST.)

Note that the *toString()* method can be invoked on any AST node to display its source code. This means that, if you are stepping through a refactoring in the debugger, you can watch the AST (or any subtree) to see the effects of each change that is made to it.

---

provides the wizard dialog used by most refactorings, including the diff view shown in the preview page. This allows refactorings for Java, C/C++, and Fortran to all have the same look and feel. See "The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs" for an introduction to the LTK.

3.5.3 Modifying, Moving, Removing, and Replacing AST Nodes

In the org.eclipse.photran.internal.core.refactoring package, there is a class called *_AST_VPG_HOWTO* which contains several small "snippets" illustrating how to perform common tasks with the AST/VPG. This includes how to traverse the AST using a visitor, how to find nodes of a particular type, how to find all of the references to a particular name, etc. It also includes several examples of how to move, copy and delete nodes in an AST.

3.5.4 Inserting new AST Nodes

Some refactorings require inserting new AST nodes into the current program. For instance, the "Intro Implicit None Refactoring" inserts new declaration statements to make the type of each variable more explicit.

There are *three* steps involved in inserting a new AST node:

1. Constructing the new AST node.
2. Inserting the new AST node into the correct place.
3. Re-indenting the new AST node to fit within the current file.

Constructing the new AST node A refactoring class inherits several convenience methods for constructing new AST nodes. These are all named *parseLiteral...*. For instance, the *parseLiteralStatement* methods constructs a list of AST nodes for use in the "Intro Implicit None" refactoring.

Inserting the new AST node Examples for how to insert the new AST node are included in the *_AST_VPG_HOWTO* class mentioned above.

Re-indenting the new AST node It might be necessary to re-indent the newly inserted AST node so that it conforms with the indentation at its insertion point. The *Reindenter* utility class provides the static method *reindent* to perform this task.

3.5.5 Committing Changes

After all of the changes have been made to a file's AST, *addChangeFromModifiedAST* has to be invoked to inform the Eclipse refactoring infrastructure that the source code should be replaced with the revised source code in the modified AST. *If you forget to invoke this method, your refactoring will appear to do nothing!*

3.6 Caveats

CAUTION: Internally, the AST is changed only enough to reproduce correct source code. After making changes to an AST, most of the accessor methods on *Token*s (*getLine(), getOffset(),* etc.) will return *incorrect* or *null* values.

Therefore, *all program analysis should be done first*. Affected *Token*s should be recorded as *TokenRef*s and affected *Definition*s should be stored *prior* to making any modifications to the AST. In general, ensure that all analyses (and storing of important information from *Token*s) is done in the *doCheckInitialConditions* and *doCheckFinalConditions* methods of your refactoring *before* the *doCreateChange* method is invoked and before the AST is modified in any way.

3.6.1 *Token* or *TokenRef*?

*Token*s form the leaves of the AST – therefore they exist as part of the Fortran AST. Essentially this means that holding on to a reference to a *Token* object requires the entire AST to be present in memory.

*TokenRef*s are lightweight descriptions of tokens in an AST. They contain only three fields: filename, offset and length. These three fields uniquely identify a particular token in a file. Because they are not part of the AST, storing a *TokenRef* does not require the entire AST to be present in memory.

If a refactoring only modifies one or two files, using either *Token*s or *TokenRef*s does not make much of a difference. However, in a refactoring like "Rename" that could potentially modify hundreds of files, it is impractical to store all ASTs in memory at once. Because of the complexity of the Fortran language itself, its ASTs can be rather large and complex. Therefore storing references to *TokenRef*s would minimize the number of ASTs that must be in memory.

To retrieve an actual *Token* from a *TokenRef*, call the *findToken()* method in *PhotranTokenRef*, a subclass of *TokenRef*.

To create a *TokenRef* from an actual *Token*, call the *getTokenRef* method in *Token*.

3.7 Testing a Refactoring

Writing JUnit tests for a refactoring is simple (starting with Photran 6 anyway). Any of the tests suites in the org.eclipse.photran.internal.tests.refactoring package can be used as examples.

Each refactoring has a corresponding test suite in that package. The test suite is a class which inherits from *PhotranRefactoringTestSuiteFromMarkers*. The test suite is constructed by importing files from a directory in the source tree, searching its *markers* (comma-separated lines starting with !<<<<<), and adding one test case to the suite for each marker.

Markers are expected to have one of the following forms:

1. !<<<<<) *line, col, ...*
2. !<<<<<) *fromLine, fromCol, toLine, toCol, ...*

That is, the first two fields in each marker are expected to be a line and column number; the text selection passed to the refactoring will be the offset of that line and column. The third fourth fields may also be a line and column number; then, the selection passed to the refactoring will extend from the first line/column to the second line/column.

The line and column numbers may be followed by an arbitrary number of fields that contain data specific to the refactoring being invoked. Many refactorings don't require any additional data; the Extract Local Variable test suite uses one field for the new variable declaration; the Add ONLY to USE Statement test suite uses these fields to list the module entities to add; etc.

The final field must be either `pass`, `fail-initial`, or `fail-final`, indicating whether the refactoring should succeed, ail its initial precondition check, or fail its final precondition check. If the refactoring is expected to succeed, the Fortran program will be compiled and run before and after the refactoring in order to ensure that the refactoring actually preserved behavior.

To create a test suite for a new refactoring:

1. In the org.eclipse.photran.internal.tests.refactoring package, copy an existing test suite class, like DataToParameterTestSuite.java.
2. Change the name of the Refactoring class in the list of type parameters and in the constructor.
3. If your markers will contain information other than fromLine, fromCol, toLine, toCol, and pass/fail, override the *configureRefactoring* method. See *ExtractLocalVariableTestSuite* for an example.
4. Inside the *refactoring-test-code* folder, create a new subfolder named after your refactoring. We will call this the *base folder.* In the test suite class, change the string variable `DIR` to point to the base folder.
5. Inside the base folder, create one or more subfolders. Each subfolder will be imported into the runtime workspace as a single Eclipse project, so it may contain only one Fortran file, or it may contain several. The refactoring will be run according to the markers in its files. Remember, markers have the basic form `!<<<<<<` *startLine, startCol, endLine, endCol, result.*
6. If there is only one marker anywhere in the subfolder, and it indicates that the refactoring should succeed, you may also create a *.result* file. For example, if the folder contains *hello.f90*, it may also contain *hello.f90.result*. This file should contain the *exact text* the file is expected to have after the refactoring has been performed. The *.result* file will be compared character-by-character with the result of the refactoring, so spaces, tabs, and newlines *do* matter; the test will fail even if a single space is incorrect. (This is because a production refactoring tool must produce correctly-indented code and handle comments correctly, and this is an easy way to test for this.) Note that you will probably need to include the marker in both the original file and the result file, based on how comments are handled by the refactoring (since it is, after all, simply another comment in the Fortran code).

Chapter 4

The Fortran Editor

Revision: $Id: cha-editor.ltx-inc,v 1.1 2010/05/21 20:12:20 joverbey Exp - based on 2008/08/08 nchen

4.1 Fortran Text Editor

The Fortran code editor is implemented in a class called, not surprisingly, *FortranEditor*. In general, *FortranEditor* closely follows the standard implementation of text editors in Eclipse (see *The Java Developer's Guide to Eclipse,* Chapter ?? or another reference for more information.) This chapter highlights some of the Photran-specific details of the text editor implementation.

4.2 Contributed *SourceViewerConfiguration*

Text editors in Eclipse rely on a *SourceViewerConfiguration* to enhance the current editor with features such as auto indenting, syntax highlighting and formatting. By default, most of these features are already provided by the concrete *SourceViewerConfiguration* class. However, it is possible to provide a custom implementation of a *SourceViewerConfiguration*. This is done by calling the *setSourceViewerConfiguration(SourceViewerConfiguration sourceViewerConfiguration)* method in an Eclipse text editor.

Photran's text editor is contained in the org.eclipse.photran.ui plug-in. However, several of the features contributed by the *SourceViewerConfiguration* require the Fortran parser, which is contained in the org.eclipse.photran.ui.vpg plug-in. Therefore, the ui.vpg plug-in *contributes* a *SourceViewerConfiguration* to the org.eclipse.photran.ui plug-in via an extension point. Specifically, it contributes the class *FortranVPGSourceViewerConfigurationFactory* to the *org.eclipse.photran.ui.sourceViewerConfig* extension point, and that class is responsible for creating the actual *SourceViewerConfiguration*.

4.3 Fortran Editor Tasks: VPG & AST Tasks

When the user modifies the text in an editor, often several other data structures and views need to be updated as well. For example, the Outline view should summarize the current text in the editor, and the list of choices in the content assistant (auto-completion) should be consistent with the current text in the editor. Both of these need to be updated as the user types—not just when the file is saved.

Eclipse editors can have a *reconciler*. When the user has stopped typing for a brief period of time (usually 500ms), the reconciler is run in a background thread. This when these things should be synchronized with the editor.[1]

Photran maintains a list of "tasks" that are run when the editor is reconciled. The list of tasks to run is stored in the singleton *FortranEditorTasks* object, and the tasks are actually run by the *FortranVPGReconcilingStrategy* class.

Currently, there are two kinds of tasks that can be run: Abstract Syntax Tree (AST) editor tasks and Virtual Program Graph (VPG) editor tasks. AST editor tasks depend on information from the AST of the current source file; and VPG editor tasks depend on information from the VPG of the current source file. *FortranEditorTasks* automatically schedules the VPG editor tasks using an instance of *VPGSchedulingRule* to synchronize access to the *PhotranVPG* singleton object. The AST of the current file is computed on-the-fly as the user modifies the source file. The VPG of the current file is based off its previous saved version (so it is less up-to-date). For more information about the AST and VPG, see Chapter 2.

AST editor tasks must implement the *IFortranEditorASTTask* interface and VPG editor tasks must implement the *IFortranEditorVPGTask* interface. Additionally, each task has to register itself with the *FortranEditorTasks* object. A task that no longer needs to run should also be unregistered. Since these tasks run asynchronously, it is important to use proper Java concurrency practices i.e. **synchronized** methods and statements.

Below is the API of the *FortranEditorTasks* class:

Listing 4.1 API of *FortranEditorTasks* (see FortranEditorTasks.java)

```
1  public class FortranEditorTasks
2  {
3    public static FortranEditorTasks instance(AbstractFortranEditor editor)
4
5    public synchronized void addASTTask(IFortranEditorASTTask task)
6
7    public synchronized void addVPGTask(IFortranEditorVPGTask task)
8
9    public synchronized void removeASTTask(IFortranEditorASTTask task)
10
11   public synchronized void removeVPGTask(IFortranEditorVPGTask task)
12
13   public Runner getRunner()
14
15   ...
16 }
```

It is possible for a class to implement both the *IFortranEditorASTTask* and *IFortranEditorVPGTask* interfaces. For example, the *DeclarationView* class registers itself for both kinds of editor tasks and makes use of the information from both as it attempts to present the declaration for the currently selected token of the text editor.

For more information on implementation details, please refer to the following classes:

- *DeclarationView*
- *FortranCompletionProcessorASTTask*
- *FortranCompletionProcessorVPGTask*
- *OpenDeclarationASTTask*

[1] Exercise for the reader: think about (1) why this should be done after a 500ms "break" rather than after every keystroke, and (2) why it should be done in a background thread...

Appendix A

Adding New Fortran Syntax

The process of adding new syntax to Photran is as follows.[1]

1. Modify the lexers and parser:
   (a) Modify the grammar (fortran2008.bnf) to recognize new syntactic constructs, and modify the phase 1 lexers (FreeFormLexerPhase1.flex and FixedFormLexerPhase1.flex) to recognize new keywords
   (b) Add new terminal symbols to Terminal.java
   (c) Run the build-lexer and build-parser scripts to regenerate the lexers and parser
   (d) Modify the phase 2 lexer (FreeFormLexerPhase2.java) to correctly resolve any new keywords as identifiers
2. Modify the syntax highlighting for the Fortran editor:
   (a) Modify the list of keywords in FortranKeywordRuleBasedScanner
   (b) Modify the keyword/identifier resolution rules in SalesScanKeywordRule
3. Modify the model builder (Outline view), if necessary:
   (a) Add new model elements to FortranElement.java, if necessary, and place their Outline view icons in the org.eclipse.photran.cdtinterface/icons/model folder
   (b) Modify FortranModelBuildingVisitor.java to visit the new constructs and add them to the model
4. Modify the name binding analysis, if necessary:
   (a) *IMPORTANT:* If you change any classes that implement IPhotranSerializable, or if you change the ScopingNode class, then be sure to change the VPG database filename in PhotranVPGDB. This will ensure that end users' code is completely reindexed using the new versions of the serialized classes.
   (b) If any new syntactic constructs are subclasses of ScopingNode, there are several methods in the ScopingNode class that will need to be modified to handle the new node type. Currently, these are easy to identify because they all contain large numbers of "instanceof" tests (which is ugly; we will eventually do this the "right" way and dispatch dynamically to the subclasses after the parser generator allows custom subclasses)
   (c) If the new syntax contains identifiers, modify the ReferenceCollector to bind any identifiers in new syntactic constructs to their declarations
   (d) Similarly, if necessary, modify the DefinitionCollector to add any new declarations to the VPG database
   (e) Modify the other collector classes in the same package if necessary

[1] Note that Photran originally handled Fortran 95, and it was later extended to work with Fortran 2003 and then Fortran 2008; the requisite changes to the lexer, parser, and syntax highlighting code are fairly clearly marked.

Appendix B

Regenerating the Help Plug-in

Photran's User's Guide and Advanced Features Guide are maintained on the Eclipse wiki (see the "Documentation" link on Photran's Web site.) However, to accommodate users without Internet access, the documentation is also available in Eclipse Online Help; Photran supplies this via the org.eclipse.photran.doc.user plug-in.

The content in the org.eclipse.photran.doc.user plug-in is *automatically generated* from the markup on the Eclipse wiki. Before a release, assuming the wiki was edited, this content must be regenerated from the revised wiki markup according to the following procedure.

1. Check out the WikiToEclipse project from Subversion. See Appendix **??**.
2. Go to the Eclipse wiki, and "Edit" the Photran User's Guide. Copy the wiki markup into *input/basic* in the WikiToEclipse project. Similarly, edit the Advanced Features Guide, and copy the wiki markup into *input/advanced*. (These are both ordinary text files.)
3. If there are any new images, copy them into *input/images* in the WikiToEclipse project. It is perhaps easiest to view the "Printable" version of each wiki page in Firefox, and save it as "Web Page (complete)" (this will save it and all referenced images); however, this will also include some irrelevant images from the Eclipse Wiki (e.g., the "search" icon) and some CSS and JavaScript, so those should be deleted.
4. Edit the Main class in the WikiToEclipse project. Make sure the plug-in version number is correct.
5. Run the Main class. If the parser fails, there is probably a problem with the wiki markup (e.g., unclosed double-quotes for italics).
6. Overwrite the contents of the org.eclipse.photran.doc.user project from CVS with the generated content. This can be done, for example, by using rsync as follows (don't forget the final slashes on the end of the doc.user directory names!).

```
rsync -av --delete --cvs-exclude \
    /your/workspace/WikiToEclipse/output/org.eclipse.photran.doc.user/ \
    /your/workspace/org.eclipse.photran.doc.user/
```

7. Proofread the generated content by launching a runtime workspace and viewing the online help. A common problem is HTML tags appearing literally in the output because, for example, the tool converted "<ol>" to "<ol>". If this happens, modify the WikiToEclipse class *MultiFileConverter.java* (around line 300) to make the appropriate replacement.

Appendix C

Manual Tests for the GNU Managed Build Toolchain

Last modified December 5, 2011

This appendix describes the procedure for manually validating support for the GNU Fortran toolchain in the managed build system.

Setup

1. Create a new managed build Fortran project using the "Executable (Gnu Fortran on [your platform])" project type.
2. In the project properties, under Fortran Build > Settings, check the following.
   (a) In the Tool Settings tab, ensure that "GNU Fortran Compiler" and "GNU Fortran Linker" are listed.
   (b) In the Error Parsers tab, ensure that the "Photran Error Parser for GNU Fortran" is checked.
   (c) In the Binary Parsers tab, ensure that the correct binary parser is checked: PE Windows Parser for Windows, Elf Parser for Linux, or Mach-O 64 for Mac OS X.

Basic Free Form Compilation Test

3. Add a file named `hello.f90` to the project root with the following contents.
```
print *, "Hello"
end program
```
4. Build the program. Ensure that the build completes successfully and that both of the following lines are present in the Console view.
```
gfortran -funderscoring -O0 -g -Wall -c -fmessage-length=0 -o "hello.o" "../hello.f90"
gfortran  -o "Test"  ./hello.o
```
5. Select the Debug/Test binary in the Project Explorer, and run it via Run > Run As > Local Fortran Application. Ensure that the Console view contains the output "Hello".

Basic Error Parser Test

6. Change the second line of hello.f90 to the following.
```
end␣program␣thisisincorrect
```
7. Save the file and build the project. Ensure that an error marker appears next to line 2 and that the Problems view contains "../hello.f90:2.27: Error: Syntax error in END PROGRAM statement at (1)"

Basic Fixed Form Compilation and Error Parser Test

8. If hello.f90 is open in an editor, close that editor.

9. Rename the file `hello.f90` to `hello.f` and change its contents to the following. Spacing must be exact: the second line must begin in column 7 and the third in column 6.
```
c␣Testing
␣␣␣␣␣␣print␣*,
␣␣␣␣␣*␣"Testing"
99999␣end
```

10. Build the project. Ensure that a warning marker appears next to line 4 with "Warning: Label 99999 at (1) defined but not used"

11. Ensure that the program runs and produces the output "Testing" in the Console view.

Compilation Options Test

12. In the project properties, under Fortran Build > Settings, switch to the Tool Settings tab, and change settings as follows.

    (a) Under GNU Fortran Compiler > Symbols, change Symbol Underscoring to "None (-fno-underscoring)".

    (b) Under GNU Fortran Compiler > Directories, add "/home" to the list of Include paths.

    (c) Under GNU Fortran Compiler > Optimization, change the Optimization Level to "Optimize most (-O3)".

    (d) Under GNU Fortran Compiler > Debugging, change Debug Level to "None".

    (e) Under GNU Fortran Compiler > Debugging, check the "Generate gprof information (-pg)" checkbox.

    (f) Under GNU Fortran Compiler > Warnings, check the "Pedantic (-pedantic)" checkbox.

    (g) Under GNU Fortran Compiler > Warnings, check the "Warnings as errors (-Werror)" checkbox.

13. Build the project. Ensure that the Console view includes the following line.
```
gfortran -fno-underscoring -I/home -O3 -pg -pedantic -Wall -Werror -c -fmessage-length=0 -o "hello.o" "../he
```

Dependency Test

14. Change the contents of `hello.f` to the following.
```
c␣Testing
␣␣␣␣␣␣use␣module
␣␣␣␣␣␣call␣hello
99999␣end
```

15. Create a second file named `module.f90` in the project root with the following contents.
```
module␣module
contains
␣␣subroutine␣hello
␣␣␣␣print␣*,␣"Hello␣from␣module"
␣␣end␣subroutine
end␣module
```

16. Build the project. In the Console view, ensure that `module.f90` is compiled *before* `hello.f`. Ensure that the build completes successfully and the resulting executable produces the output "Hello from module".

Appendix D

Release and Deployment Procedure

Last modified April 28, 2010

Most contributors/committers do not need to read this. This explains our entire release and deployment procedure: setting the Photran version number, updating the Web site, etc.

D.1 Preparing for a Release Build

1. Proofread the documentation on the wiki. Make sure version numbers are correct, screenshots and step-by-step instructions are up-to-date, and UI labels are up-to-date (e.g., if the name of a refactoring changed).
2. Regenerate the org.eclipse.photran.doc.user plug-in from the wiki. See Appendix B for details.
3. Update the plug-in version numbers in all projects.
4. Update the feature.xml and feature.properties files for
   - org.eclipse.photran-feature,
   - org.eclipse.photran.intel-feature,
   - org.eclipse.photran.xlf-feature, and
   - org.eclipse.rephraserengine-feature.

   (a) Change the feature version.

   (b) Change the versions of other features/plug-ins it depends on...
      - org.eclipse.photran-feature must specify the correct versions of
        - CDT
        - Rephraser Engine
      - org.eclipse.photran.intel-feature must specify the current version of Photran
      - org.eclipse.photran.xlf-feature must specify the current version of Photran

   (c) the copyright year

   (d) the update site URL
5. Update URLs to reflect the new version numbers...

   (a) Update the Welcome Page URL in org.eclipse.photran.ui/intro/overviewContent.xml

(b) Note that the URL for the release notes shown when the user first installs a new version of Photran is determined by the version of the org.eclipse.photran.ui plug-in (see ShowReleaseNotes.java for details); this is not necessarily the same as the Welcome Page URL.

(c) Be sure the Web site actually contains pages at these URLs; add them if necessary

6. If the VPG database structure (or any of the serialized classes) have changed, update

   - the VPG database filename in the PhotranVPGDB class constructor and
   - the VPG log filename in the PhotranVPGLog class constructor.

   For example, in Photran 4.0 beta 5, the database filename was "photran40b5vpg".

7. Make sure the org.eclipse.photran.cmdline JAR is up-to-date.

8. Make sure the org.eclipse.ptp.releng scripts have the correct versions. *Note that Photran is built from PTP's releng scripts; Photran's own releng scripts are no longer used.*

9. Greg Watson will initiate a PTP build at build.eclipse.org.

D.2 After the Release Has Been Built

1. Update the timeline in this guide's Release History appendix (org.eclipse.photran-dev-docs/dev-guide/app-history.ltx-inc)

2. The PTP release engineering team will create a maintenance branch in Git. For example, a ptp_5_0 branch was created after the Photran 7/PTP 5 release and was subsequently used to build Photran 7.0.1, etc.

3. Add a Bugzilla version for the release and a target for the next expected release

4. Update the Web site:

   (a) Update the home page to mention the release

   (b) Change the update site URL and archived update site link on the Downloads page

   (c) Update the Report a Bug URL in the nav bar to default to the new release version

5. Announce the release, e-mailing the photran, ptp-dev, and ptp-announce mailing lists

6. Copy the Documentation pages on the wiki and update the version numbers for the next expected release

7. Update the Project Plan at the Eclipse Foundation Portal...

   (a) Mark the released version as "released" with the correct date.

   (b) Add a planned/tentative next release at some future date.