Photran 6.0 Developer's Guide

Part II: Specialized Topics

N. Chen
J. Overbey

Contents

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 The details (and code) are now outdated, but the concepts are still applicable.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) model that is discussed in Section **??**. They are **not** identical. It is helpful to think of the CDT model as containing a *partial/simplified view* of the AST model to represent the high-level/organizational elements in the source code (program units, subprogram declarations, etc.) **in addition** to 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 the source file (but nothing about the resources), including low-level elements that the user is unlikely to be interested in knowing about (assignment nodes, variable declarations). While low-level, these elements are useful for refactoring and program analysis.

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 sec:asts.)

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:

```
program main
    integer a
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 specification 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 gramar, 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()
```

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

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 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

[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.

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:

```
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 3.4.1 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)
```

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

```
{
  // search through each ScopingNode
}
```

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.

Chapter 3

Refactorings

Revision: $Id: cha-refactoring.ltx-inc,v 1.1 2010/05/21 20:12:20 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].

As mentioned in Chapter **??**, the purpose in writing Photran was to create a refactoring tool for Fortran. Because Photran is structured as a plug-in for Eclipse, we can take advantage and reuse many of the language-neutral support that Eclipse provides for refactoring. This makes it possible to create refactoring tools that *resemble* the Java Development Tools that most Eclipse programmers are already familiar with.

However, implementing first-class support for Fortran refactoring is not an easy task. It requires having an accurate representation of the underlying Fortran source files so that our tools can perform proper program analysis to construct our automated refactoring. The VPG (see Chapter **??**) is our initial step in providing such a representation; the VPG will be improved in future versions of Photran to provide support for many different types of refactoring and program analysis.

In this chapter, we describe how to add automated refactorings for Fortran using the underlying infrastructure provided by Eclipse (and Photran) as well as the analysis tools provided by the VPG.

3.2 Structure of a Fortran Refactoring

Refactorings in Photran are subclassed from either `SingleFileFortranRefactoring` or `MultipleFileFortranRefactor`
Both of these are subclasses of `AbstractFortranRefactoring`, which is in turn a subclass of the `Refactoring` class provided by the Eclipse Language ToolKit (LTK)[2].

The LTK is a language-neutral API for supporting refactorings in the Eclipse environment. It provides a generic framework to support the following functionalities:

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

[2] See The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs for an introduction to the LTK.

1. Invoking the refactoring from the user interface (UI).
2. Presenting the user with a wizard to step through the refactoring.
3. Presenting the user with a preview of the changes to be made.

In other words, the LTK provides a common UI for refactorings: This allows refactorings for Java, C/C++, and Fortran to all have the same look and feel.

A concrete Fortran refactoring must implement the following *four* methods:

Listing 3.1 Abstract methods of `AbstractFortranRefactoring` class

```
public abstract String getName();

protected abstract void doCheckInitialConditions(RefactoringStatus status,
  IProgressMonitor pm) throws PreconditionFailure;

protected abstract void doCheckFinalConditions(RefactoringStatus status,
  IProgressMonitor pm) throws PreconditionFailure;

protected abstract void doCreateChange(IProgressMonitor pm) throws
  CoreException, OperationCanceledException;
```

`getName` simply returns the name of the refactoring: “Rename,” “Extract Subroutine,” “Introduce Implicit None,” or something similar. This name will be used in the title of the wizard that is displayed to the user.

Initial conditions are checked before any wizard is displayed to the user. An example would be making sure that the user has selected an identifier to rename. If the check fails, a `PreconditionFailure` should be thrown with a message describing the problem for the user.

Final conditions are checked after the user has provided any input. An example would be making sure that the new name that that user has provided is a legal identifier.

The actual transformation is done in the `doCreateChange` method, which will be called only after the final preconditions are checked. For more information, see Section 3.3.

The `AbstractFortranRefactoring` class provides 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`. It is worth reading through the source code for `AbstractFortranRefactoring` before writing your own utility methods.

3.3 Creating Changes: AST Rewriting

After determining the files that are affected and the actual changes that are required for a particular refactoring, manipulating the source code in the `doCreateChange` method is conceptually straightforward.

Instead of manipulating the text in the files directly (by doing a textual find & replace) we use a more scalable approach: manipulating the Abstract Syntax Tree (AST) of the source code. This allows us to make changes based on the program's semantics and its syntactic structure. This section assumes some familiarity with the AST used in Photran. For more information about the AST, refer to Section 2.3.

3.3.1 Common Methods for Manipulating the AST

In the following paragraphs, we describe some of the approaches that are currently being used in Photran for manipulating the AST.

Changing the Text of `Tokens`

To change the text of a single token, simply call its `setText` method. This is used in `RenameRefactoring` to rename tokens while preserving the "shape" of the AST.

Listing 3.2 Use of `setText` in `RenamingRefactoring` (see RenameRefactoring.java)

```
private void makeChangesTo(IFile file, IProgressMonitor pm) throws Error
{
  try
  {
    vpg.acquirePermanentAST(file);

    if (definitionToRename.getTokenRef().getFile().equals(file))
        definitionToRename.getTokenRef().findToken().setText(newName);

    for (PhotranTokenRef ref : allReferences)
        if (ref.getFile().equals(file))
            ref.findToken().setText(newName);

    addChangeFromModifiedAST(file, pm);

    vpg.releaseAST(file);
  }
  catch (Exception e)
  {
    throw new Error(e);
  }
}
```

Removing/replacing AST Nodes

To remove or replace part of an AST, call `replaceChild`, `removeFromTree` or `replaceWith` on the node itself. These methods are defined in the `IASTNode` interface that all nodes implement. Line 107 of Listing 3.4 shows an example of the `removeFromTree` method.

In addition, if the *specific* type of the AST is known, then it is possible to just call its *setter* method to directly replace particular nodes. For more information on the available setters for each node type, see Section **??**.

Listing 3.3 AST manipulation methods in `IASTNode` (see Parser.java) that all AST nodes implement

```
public static interface IASTNode
{
  void replaceChild(IASTNode node, IASTNode withNode);
  void removeFromTree();
  void replaceWith(IASTNode newNode);
  ...
}
```

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 The `AbstractFortranRefactoring` class provides convenience methods for constructing new AST nodes. These methods should be treated as part of the API for Fortran refactorings . For instance, the `parseLiteralStatement` methods constructs a list of AST nodes for use in the "Intro Implicit None" refactoring.

Inserting the new AST node Inserting the new AST node can be accomplished using the approach discussed previously in *Removing/replacing AST Nodes*.

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. Refer to line 111 of Listing 3.4.

3.3.2 Committing Changes

After all of the changes have been made to a file's AST, `addChangeFromModifiedAST` has to be invoked to actually commit the changes. This convenience function creates a new `TextFileChange` for the *entire* content of the file. The underlying Eclipse infrastructure performs a `diff` internally to determine what parts have actually changed and present those changes to the user in the preview dialog.

Listing 3.4 Inserting new declarations into an existing scope (see `IntroImplicitNoneRefactoring.java`)

```java
protected void doCreateChange(IProgressMonitor progressMonitor) throws
CoreException, OperationCanceledException
{
  assert this.selectedScope != null;

  for (ScopingNode scope : selectedScope.getAllContainedScopes())
  {
    if (!scope.isImplicitNone()
      && !(scope instanceof ASTExecutableProgramNode)
      && !(scope instanceof ASTDerivedTypeDefNode))
    {
      ASTImplicitStmtNode implicitStmt = findExistingImplicitStatement(scope);
      if (implicitStmt != null) implicitStmt.removeFromTree();

      IASTListNode<IBodyConstruct> newDeclarations = constructDeclarations(scope);
      scope.getBody().addAll(0, newDeclarations);
      Reindenter.reindent(newDeclarations, astOfFileInEditor);
    }
  }

  this.addChangeFromModifiedAST(this.fileInEditor, progressMonitor);
  vpg.releaseAllASTs();
}
```

3.4 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 `Tokens` (`getLine(), getOffset(),` etc.) will return *incorrect* or *null* values.

Therefore, *all program analysis should be done first*; pointers to all relevant **tokens** should be obtained (usually as `TokenRefs`) *prior* to making any modifications to the AST. In general, ensure that all analysis (and storing of important information from `Tokens`) should be done in the `doCheckInitialConditions` and `doCheckFinalConditions` methods of your refactoring before the `doCreateChange` method.

3.4.1 `Token` or `TokenRef`?

`Tokens` 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.

`TokenRefs` 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.

For most refactorings, using either `Tokens` or `TokenRefs` does not make much of a difference. However, in a refactoring like "Rename Refactoring" 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 `TokenRefs` 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`.

3.5 Examples

The "Rename", "Introduce Implicit None" and "Move COMMON To Module" refactorings found in the `org.eclipse.photran.in` package inside the `org.eclipse.photran.core.vpg` project are non-trivial but readable and should serve as a model for building future Fortran refactorings.

An example of a simpler but rather *useless* refactoring is presented in Appendix B. It should be taken as a guide on the actual steps that are involved in registering a new refactoring with the UI and also how to actually construct a working Fortran refactoring.

3.6 Common Tasks

In this section, we briefly summarize some of the common tasks involved in writing a new Fortran refactoring.

In an AST, how do I find an ancestor node that is of a particular type?
Sometimes it might be necessary to traverse the AST *upwards* to look for an ancestor node of a particular type. Instead of traversing the AST manually, you should call the `findNearestAncestor(TargetASTNode.class)` method on a `Token` and pass it the **class** of the ASTNode that you are looking for.

How would I create a new AST node from a string?
Call the `parseLiteralStatement(String string)` or `parseLiteralStatementSequence(String string)` method in `AbstractFortranRefactoring`. The former takes a `String` that represents a single statement while the latter takes a `String` that represents a sequence of statements.

How do I print the text of an AST node and all its children nodes?
Call the `SourcePrinter.getSourceCodeFromASTNode(IASTNode node)` method. This method returns a `String` representing the source code of its parameter; it includes the user's comments, capitalization and whitespace.

Chapter 4

Photran Editors

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 Editors

There are **two** different text editors in Photran. This is necessary to support both the fixed-form Fortran 77 standard and the free-form Fortran 90 & 95 standard.

Fortran 77 is known as fixed-form Fortran because it requires certain constructs to be *fixed* to particular columns. For instance, Fortran statements can only appear between columns 7 - 72; anything beyond column 72 is ignored completely. This requirement is an artifact of the days when punched cards were used for Fortran programming. However, Fortran 77 compilers still enforce this requirement. The fixed-form editor in Photran helps the programmer remember this requirement by displaying visual cues to denote the column partitions.

Fortran 90/95 adopted the free-form format that most programmers today are accustomed to. Nonetheless, because Fortran 77 is still considered a subset of Fortran 90/95, it is possible to write programs following the fixed-form format. As such, the free-form editor maintains some visual cues on column numbering (although using a more subtle UI).

The UML class diagram in Figure 4.1 shows the hierarchy of the editors in Photran.

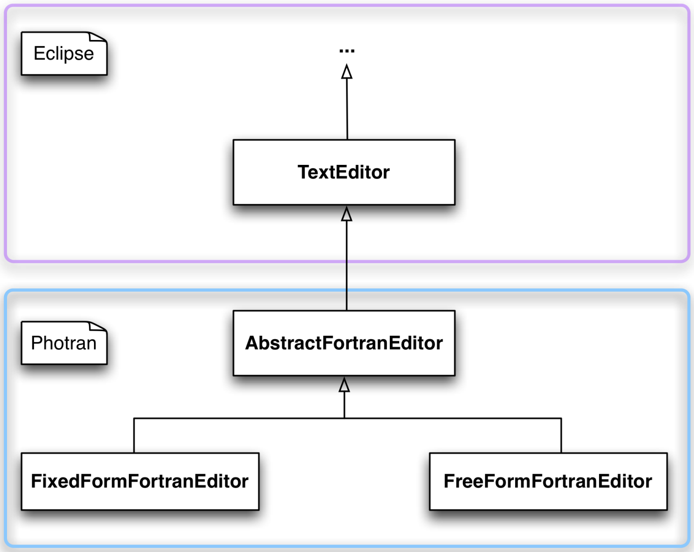
Both the `FixedFormFortranEditor` and `FreeFormFortranEditor` concrete classes inherit from `AbstractFortranEdit` Most of the actual work is done inside `AbstractFortranEditor`; its subclasses just specify how to decorate the UI.

In general, the implementation of `AbstractFortranEditor` closely follows the standard implementation of text editors in Eclipse. The following section highlights some of the Photran-specific mechanisms of the text editor. For more information on how text editors work in Eclipse, please consult the Eclipse references mentioned in Chapter **??**.

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(SourceViewerConfig`

Figure 4.1 Photran editor class hierarchy



`sourceViewerConfiguration)` method in an Eclipse text editor.

Photran provides an additional layer of flexibility by allowing its `SourceViewerConfiguration` to be contributed from other plug-ins. A plug-in that is interested in contributing a `SourceViewerConfiguration` to the Photran editors must extend the
`org.eclipse.photran.ui.sourceViewerConfig` extension point defined in the
`org.eclipse.photran.ui.vpg` plug-in.

At run-time, Photran *dynamically* instantiates a suitable `SourceViewerConfiguration` by searching through the list of plug-ins that extend the `org.eclipse.photran.ui.sourceViewerConfig` extension.

Currently, there are **two** `SourceViewerConfigurations` in Photran: the contributed `FortranVPGSourceViewerConfigura`
and the default (but less feature-full) `FortranModelReconcilingSourceViewerConfiguration`.

4.3 Fortran Editor Tasks: VPG & AST Tasks

Many actions that a user can invoke actually depend on the current text in the text editor. For instance, auto-completion depends on the text that has been entered so far to provide feasible completion choices. A good auto-completion strategy constantly augments and updates its database of feasible completion choices as the user enters or modifies text; it does not have to wait until the user saves the current file.

Another important feature of the text editor that requires constant updates is syntax highlighting. Syntax highlighting has to work almost instantaneously based on what the user has entered. It is not acceptable for the user to experience lengthy delays between typing a character and waiting for it to syntax highlight correctly.

Eclipse utilizes a *reconciler* to correctly and instantly perform syntax highlighting. The reconciler runs in a background thread in Eclipse, constantly monitoring the text that the user enters and updating the syntax highlighting as necessary. Every text editor in Eclipse – including Photran's – has a corresponding reconciler.

Photran takes advantage of its existing reconciler (`FortranVPGReconcilingStrategy`) and adds additional Fortran editor tasks that should run each time its reconciler runs. The list of tasks to run is stored in the singleton `FortranEditorTasks` object.

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 **??**.

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)

```
public class FortranEditorTasks
{
  public static FortranEditorTasks instance(AbstractFortranEditor editor)

  public synchronized void addASTTask(IFortranEditorASTTask task)

  public synchronized void addVPGTask(IFortranEditorVPGTask task)

  public synchronized void removeASTTask(IFortranEditorASTTask task)

  public synchronized void removeVPGTask(IFortranEditorVPGTask task)

  public Runner getRunner()

  ...
}
```

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`

Appendix A

Creating an Error Parser

Revision: $Id: app-error-parsers.ltx-inc,v 1.3 2010/04/28 18:12:52 joverbey Exp

Error parsers scan the output of `make` for error messages for a particular compiler. When they see an error message they can recognize, they extract the filename, line number, and error message, and use it to populate the Problems view.

For an example, see `IntelFortranErrorParser`. (It's a mere 60 lines.)

To create a new error parser, do the following.

- We will assume that your error parser class will be in the `errorparsers` folder in the `org.eclipse.photran.core` plug-in and added to the `org.eclipse.photran.internal.errorparsers` package.
- Define a class implementing `IErrorParser`
- Implement `public boolean processLine(String line, ErrorParserManager eoParser)` which should always return false because ErrorParserManager appears not to use the result in a rational way
- In org.eclipse.photran.core's `plugin.xml`, find the place where we define all of the Fortran error parsers. Basically, copy an existing one. Your addition will look something like this:

```
<extension
      id="IntelFortranErrorParser"
      name="Photran Error Parser for Some New Fortran Compiler"
      point="org.eclipse.cdt.core.ErrorParser">
   <errorparser
         class="org.eclipse.photran.internal.errorparsers.MyErrorParser">
   </errorparser>
</extension>
```

- Your new error parser will appear in the error parser list in the Preferences automatically, and it will be automatically added to new projects. For existing projects, you will need to open the project properties dialog and add the new error parser to the project manually.

Note. Error parsers do not have to be implemented in the Photran Core plug-in. In fact, they do not have to be implemented as part of Photran at all. If you create a brand new plug-in, you can specify `org.eclipse.cdt.core` as a dependency, include the above XML snippet in your plug-in's `plugin.xml`, and include your custom error parser class in that plug-in. The plug-in system for Eclipse will recognize your plug-in, detect that it extends the `org.eclipse.cdt.core.ErrorParser` extension point, and add it to the list of implemented error parsers automatically.

Appendix B

Simple Fortran Refactoring Example

Revision: $Id: app-obfuscate-fortran.ltx-inc,v 1.3 2010/04/28 18:12:52 joverbey Exp - based on 2008/08/08 nchen

B.1 Introduction

Contributing a new refactoring to Photran is best done by following a working example.

This paragraph describes the general approach: First, an action must be added to both the editor popup menu **and** the Refactor menu in the menu bar by modifying the plugin.xml file. Then, the action delegate and its accompanying refactoring wizard have to be coded; these two classes are responsible for populating the user interface of the refactoring wizard dialog. Finally, the actual Fortran refactoring itself has to be coded.

The remaining sections go into the details of each of those steps based on a simple (but not useful) refactoring example: obfuscating Fortran by removing the comments and adding redundant comments to the header. The source code is available from `http://photran.cs.illinois.edu/samples/obfuscator-photran5.zip` (UIUC personnel can find it in the Subversion repository described in the appendix).

B.2 Modifying the plugin.xml

There are **four** extensions points (from the Eclipse core) that our plug-in needs to extend:

`org.eclipse.ui.commands` Creates a new command *category* to represent our refactoring. This category will be referenced by the other extensions in the plugin.xml file.

`org.eclipse.ui.actionSets` This extension point is used to add menus and menu items to the Fortran perspective.

`org.eclipse.ui.actionSetPartAssociations` Allows our refactoring to be visible/enabled in the context of the Fortran editor.

`org.eclipse.ui.popupMenus` Displays our refactoring in the pop-up menu that appears during a right-click.

`org.eclipse.ui.bindings` (Optional) Allows our refactoring to be invoked via keyboard shortcuts. For instance the Fortran Rename Refactoring is bound to the Alt + Shift + R keyboard shortcut, which is the same as the one for the Java Rename Refactoring.

Please refer to the documentation and schema description for each extension point; the documentation is available from Help > Help Contents in Eclipse.

Fortran currently does **not** use the newer `org.eclipse.ui.menus` extension points (introduced in Eclipse 3.3) for adding menus, menu items and pop-up menus.

It is possible to use the newer `org.eclipse.ui.menus` extension point if desired, but this chapter uses the older extension points to remain consistent with how Photran is doing it.

For more information, see the plugin.xml file of our refactoring example.

B.3 Creating an Action Delegate and a Refactoring Wizard

The `org.eclipse.ui.actionSets` and `org.eclipse.ui.popupMenus` extension points that were extended in our plugin.xml file require a reference to action delegate class that we need to provide.

For a Fortran refactoring, our action delegate should extend the `AbstractFortranRefactoringActionDelegate` class **and** implement the `IWorkbenchWindowActionDelegate` and `IEditorActionDelegate` interfaces.

The most important method in our action delegate class is the **constructor**. The constructor has to be done in a particular way so that everything is setup correctly. Listing B.1 shows how the constructor needs to be setup.

Listing B.1 `ObfuscateAction` for our simple refactoring example

```
public class ObfuscateAction extends AbstractFortranRefactoringActionDelegate
  implements IWorkbenchWindowActionDelegate, IEditorActionDelegate {

    public ObfuscateAction() {
      super(ObfuscateRefactoring.class, ObfuscateRefactoringWizard.class);
    }

    ...
}
```

Inside our constructor, we need to call the parent constructor that takes **two** parameters: the class of the actual refactoring object (e.g. ObfuscateRefactoring) and the class of the actual refactoring wizard (e.g. ObfuscateRefactoringWizard). The parent class will dynamically create the refactoring object and refactoring wizard using Java reflection.

Our refactoring wizard needs to be a subclass of `AbstractFortranRefactoringWizard`. The only method that we are required to implement is the `doAddUserInputPages` method. This page is responsible for creating a page for the wizard. For instance, a refactoring such as rename refactoring requires the user to provide a new name. So the `doAddUserInputPages` is responsible for creating the interface for that.

Ideally, if our refactoring does not require the user to provide any input, it should just have an empty `doAddUserInputPages` method. However, because of a bug in the Mac OS X version of Eclipse, it is necessary to add a *dummy* page. Without this dummy page the refactoring will cause the entire Eclipse UI to lock up on Mac OS X. Listing B.2 shows how to add a dummy input page.

Listing B.2 Adding a `dummy` wizard input page

```
protected void doAddUserInputPages() {
    addPage(new UserInputWizardPage(refactoring.getName()) {

  public void createControl(Composite parent) {
      Composite top = new Composite(parent, SWT.NONE);
      initializeDialogUnits(top);
      setControl(top);

      top.setLayout(new GridLayout(1, false));

      Label lbl = new Label(top, SWT.NONE);
      lbl.setText("Click OK to obfuscate the current Fortran file.
      To see what changes will be made, click Preview.");

  }
    });
}
```

B.4 Creating the Actual Refactoring

Section 3.2 gives a good overview of the **four** methods that a Fortran refactoring needs to implement. And Section 3.4 gives an overview of things to avoid while performing a refactoring. Our example refactoring conforms to the lessons in both those sections.

Here we briefly describe the four methods in our example:

getName This just returns the text "Obfuscate Fortran Code" describing our refactoring. This text will be used as the title of the refactoring wizard dialog.

doCheckInitialConditions Our simple refactoring does not have any *real* initial conditions. Our refactoring can proceed as long as the current file can be parsed as valid Fortran source code. This is automatically checked by the `FortranRefactoring` parent class.

Instead we use this method as a hook to perform some simple program analysis – acquiring the names of all the functions and subroutines in the current file. We will print these names later as part of the header comment.

doCheckFinalConditions Since we do not require the user to provide any additional input, there are no final conditions to check.

doCreateChange The actual refactoring changes are constructed in this method.

We iterate through every token in the current file to check if it has a comment string. Comment strings are acquired by calling `Token#getWhiteBefore()` and `Token#getWhiteAfter()`. Following the advice of Section 3.4, we store a list of all the tokens (call this list TokensWithComments) that contain comment strings. Once we have iterated through all the tokens, we proceed to remove the comments for tokens in our TokensWithComments list. Removing comments is done by calling `Token#setWhiteBefore()` and `Token#setWhiteAfter()` with blank strings as parameters.

Finally, we create a header comment that just lists all the functions and subroutines in the current source file and add that to the preamble of the main program.

For more information, please consult the source code for our example.

Appendix C

Adding New Fortran Syntax

Revision: $Id: app-new-syntax.ltx-inc,v 1.2 2010/04/28 18:12:52 joverbey Exp

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

[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.

(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

Appendix D

Regenerating the Help Plug-in

Revision: $Id: app-generating-help.ltx-inc,v 1.2 2010/04/28 18:12:52 joverbey Exp

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 E

Release and Deployment Procedure

Revision: $Id: app-deploy.ltx-inc,v 1.6 2010/04/28 18:12:52 joverbey Exp

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.

E.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 D 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.

E.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. Create a maintenance branch for the org.eclipse.photran module in CVS. For example, a photran_5_0 branch was created after the Photran 5.0.0 release and was subsequently used to build Photran 5.0.1, 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, photran-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.

Appendix F

Release History

Revision: $Id: app-history.ltx-inc,v 1.6 2010/04/28 18:12:52 joverbey Exp

Photran	Date	Platform	CDT	Notes
1.2	Jan 2005	2.1	1.2	First public version; hacked CDT clone
2.1	Feb 2005	3.0	2.1	First version available at eclipse.org
3.0b1	Aug 2005			
3.0b2	Nov 2005			
3.0	Jan 2006	3.1	3.0	CDT extension; required modified CDT
3.1b1	Jul 2006			
3.1b2	Apr 2007	3.1	3.1.1	Extension of stock CDT
4.0b1	Jun 2007	3.2.2	3.1.2	*Rename, intro implicit none*
4.0b2	Oct 2007	3.3.1	4.0.1	
4.0b3	Nov 2007	3.3.1.1	4.0.1	
4.0b4		3.4	5.0	First version release via an update site
4.0b5	Feb 2009	3.4	5.0.1	*Move saved vars to common*
	Mar 2009	3.4	5.0.1	First automated integration build under PTP
	Sep 2009	3.5	6.0.0	
5.0.0	Dec 2009	3.5	6.0.1	First official release at eclipse.org
6.0.0	Jun 2010	3.6	7.0.0	Part of the Helios release train (TENTATIVE)