Photran 4.0 Developer's Guide

J. Overbey

Contents

1 Introduction **2**

1.1 CDT Terminology 2

1.2 The Model 3

1.3 The CDT Debugger and `gdb` 5

2 Plug-in Decomposition **6**

2.1 The Photran Feature 6

2.2 The Virtual Program Graph (VPG) Feature: Analysis and Refactoring Support 7

2.3 Other Plug-ins 8

3 Parsing and Analysis **9**

3.1 Abstract Syntax Trees: The `FortranWorkspace` 9

3.1.1 Visualizing ASTs 9

3.1.2 Acquiring and Releasing Translation Units 9

3.1.3 The Program Representation: IFortranAST 10

3.1.4 AST Structure 10

3.2 Scope and Binding Analysis 12

3.3 Type Checking 13

4 Refactoring **14**

4.1 Structure of a Refactoring 14

4.2 Checking Preconditions 15

4.3 AST Rewriting 16

4.4 User Interface 16

4.5 An Example: Rename 16

A Getting the Photran 4.0 Sources from CVS **18**

B Creating an Error Parser **21**

C Creating Tests for the Rename Refactoring **23**

Chapter 1

Introduction

Last Updated 4/4/07

Photran is a IDE for Fortran 90/95 and Fortran 77 that is built on top of Eclipse. It is structured as an Eclipse feature, in other words, as a set of plug-ins that are designed to be used together. Starting with version 3.0, it is an extension of CDT, the Eclipse IDE for C/C++. Previous versions of Photran were created by hacking a copy of the CDT to support Fortran instead of C/C++, but now we have developed a mechanism for adding new languages into the CDT, allowing the Fortran support code to be in its own set of plug-ins.

Our purpose in writing Photran was to create a refactoring tool for Fortran. Thus, Photran has a complete parser and program representation. Photran adds a Fortran editor and several preference pages to the CDT user interface, as well as a Fortran Managed Make project type.

This document explains the design of Photran so that you could fix a bug or add a refactoring. You should know how to use Photran and how the CDT works. You need to understand Eclipse and Eclipse plug-ins before you read this document. We recommend *The Java Developer's Guide to Eclipse* for Eclipse newcomers.

1.1 CDT Terminology

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

- **Standard Make projects** are ordinary Eclipse projects, except that the CDT (and Photran) recognize them as being "their" type of project (as opposed to, say, projects for JDT, EMF, or another Eclipse-based tool). The user must supply their own Makefile, typically with targets "clean" and "all." CDT/Photran cleans and builds the project by running `make`.

- **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** is the part of CDT and Photran 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). The CDT provides binary parsers for Windows (PE), Linux (ELF), Mac OS X (Mach), 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 they can recognize, they extract the filename, line number, and error message, and use it to populate the Problems view.
- CDT keeps a **model** of all of the files in a project. The model is essentially a tree of **elements**, which all derive from a (CDT Core) class `ICElement`. It is described in the next section.

1.2 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:

```
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, for example extension
            - 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 find in the proje

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)
For a complete interface/view of the language, clients should use the __AST__ interface.
```

From another one 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.
```

1.3 The CDT Debugger and `gdb`

- The so-called CDT debugger is actually just a graphical interface to `gdb`, or more specifically to `gdb/mi`. If something doesn't work, try it in `gdb` directly, or using 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 debug UI: You will *not* find code for them in the Photran UI plug-in.
- `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

Plug-in Decomposition

Last Updated 4/4/07

2.1 The Photran Feature

The following projects comprise the “base” of Photran. All sources in these projects are Java 4.

- org.eclipse.photran-feature

  This is the Eclipse feature for Photran, used to build the Zip file distributed to users. (A feature is a grouping of related plug-ins.)

- org.eclipse.photran.cdtinterface

  This contains all of the components (core and user interface) related to integration with the CDT. It includes

  - ILanguage for Fortran (i.e., the means of adding Fortran to the list of languages recognized by the CDT)
  - Fortran model elements, and icons for the Outline and Projects views
  - Simple lexical analyzer-based model builder, and an extension point for contributing more sophisticated model builders
  - Fortran perspective, Fortran Projects view, and new project wizards

- org.eclipse.photran.core

  This is the Photran Core plug-in. It contains much of the Fortran-specific “behind the scenes” functionality:

  - Utility classes
  - Error parsers for Fortran compilers
  - Fortran 95 lexical analyzer
  - Workspace preferences

- org.eclipse.photran.core.errorparsers.xlf

  Error parser for the XLF compiler. Managed by Craig Rasmussen at LANL.

- org.eclipse.photran.managedbuilder.core, org.eclipse.photran.managedbuilder.gnu.ui, org.eclipse.phot
  org.eclipse.photran.managedbuilder.ui

  Support for Managed Build projects using the GNU toolchain. Managed by Craig Rasmussen at LANL.

- org.eclipse.photran.core.intel, org.eclipse.photran.intel-feature, org.eclipse.photran.managedbuilder.in

  Support for Managed Build projects using Intel toolchains. Maintained by Bill Hilliard at Intel.

- org.eclipse.photran.ui

  This contains the Fortran-specific components of the user interface:

  - Editor
  - Preference pages

2.2 The Virtual Program Graph (VPG) Feature: Analysis and Refactoring Support

The following projects support parsing, analysis, and refactoring of Fortran sources. They are written in Java 5. The Virtual Program Graph is described in more detail later.

- org.eclipse.photran.vpg.core

  This contains the parsing, analysis, and refactoring infrastructure.

  - Fortran parser and abstract syntax tree (AST)
  - Fortran preprocessor (to handle INCLUDE lines)
  - Parser-based model builder
  - Virtual Program Graph library (vpg-eclipse.jar)
  - Photran's Virtual Program Graph (VPG)

- Utility classes (e.g., `SemanticError`, `LineCol`)
- Project property pages
- Binding analysis (equivalent to symbol tables)
- Type checker (incomplete)
- Refactoring/program transformation engine
- Refactorings

- org.eclipse.photran.core.vpg.tests

  JUnit Plug-in tests for the VPG core plug-in.

- org.eclipse.photran.ui.vpg

  UI contributions that depend on the org.eclipse.photran.core.vpg plug-in. Currently, this is the Open Declaration action, a project property page where the user can customize the search path for Fortran modules and include files, and all of the actions in the Refactoring menu.

2.3 Other Plug-ins

The following projects are in CVS but are not distributed to users:

- org.eclipse.photran-dev-docs

  Developer documentation, including this document (`dev-guide/*`), CVS instructions (`dev-guide/cvs-instructions.pdf`), the materials from our presentation at EclipseCon 2006 on adding a new language to the CDT, and a spreadsheet mapping features in the Fortran language to JUnit tests (`language-coverage/*`).

- org.eclipse.photran-samples

  A Photran project containing an assortment of Fortran code.

Chapter 3

Parsing and Analysis

Last Updated 4/11/07

This information is out of date and is being updated (8/07).

3.1 Abstract Syntax Trees: The `FortranWorkspace`

3.1.1 Visualizing ASTs

Photran can display ASTs rather than the ordinary Outline view. This behavior can be enabled from the Fortran workspace preferences (click on Window > Preferences). Clicking on an AST node in the Outline view will move the cursor to that construct's position in the source file.

3.1.2 Acquiring and Releasing Translation Units

`FortranWorkspace` **is a singleton object responsible for constructing ASTs for Fortran source code. ASTs are retrived by invoking**

```
public IFortranAST acquireTU(IFile file,
                             boolean shouldImportModules,
                             boolean shouldDieWhenModuleNotFound,
                             boolean isFixedForm,
                             IProgressMonitor progressMonitor) throws Exception
```

or one of its less verbose overloads.

ASTs are retained in memory until `releaseTU` or `releaseAll` is called. Until a release method is called, subsequent requests for the same `IFile` will return the same (pointer-identical) AST.

If an AST is constructed with the `shouldImportModules` option set, the ASTs for all of the files containing loaded modules may be in memory. In this case, it is particularly important to call `releaseAll` when you are done.

3.1.3 The Program Representation: IFortranAST

The `acquireTU` methods return an object implementing `IFortranAST`.

```
public interface IFortranAST extends Iterable/*<Token>*/
{
    ///////////////////////////////////////////////////////////////////////////
    // Visitor Support
    ///////////////////////////////////////////////////////////////////////////

    public void visitTopDownUsing(ASTVisitor visitor);
    public void visitBottomUpUsing(ASTVisitor visitor);
    public void visitOnlyThisNodeUsing(ASTVisitor visitor);

    ///////////////////////////////////////////////////////////////////////////
    // Other Methods
    ///////////////////////////////////////////////////////////////////////////

    public ASTExecutableProgramNode getRoot();

    public Token findTokenByOffsetLength(int offset, int length);
    public Token findFirstTokenOnLine(int line);
}
```

The `getRoot` method returns the root of the AST, while the `find...` methods provide an efficient means to search for tokens based on their lexical positioning.[1]

3.1.4 AST Structure

The Fortran AST comprises 381 classes. The class names of AST nodes all begin with `AST`, and all are located in `org.eclipse.photran.internal.core.parser`.

[1] In addition to the AST, Photran internally maintains an array of the `Tokens` in a program (in a `TokenList` object); the `find...` methods perform a binary search on this array.

The AST classes are generated automatically from the parsing grammar, `fortran95.bnf`.

Ordinary AST Nodes

The children of any AST node are retrieved by calling one of its `getAST...` methods, which return `null` if that child is not present. For example, `fortran95.bnf` defines a <*ProgramUnit*> as follows.

```
# R202
<ProgramUnit> ::=
    <MainProgram>
  | <FunctionSubprogram>
  | <SubroutineSubprogram>
  | <Module>
  | <BlockDataSubprogram>
```

An `ASTProgramUnitNode` object, then, provides the following interface.

```
public class ASTProgramUnitNode extends ParseTreeNode
{
    public ASTMainProgramNode getASTMainProgram() { ... }
    public ASTFunctionSubprogramNode getASTFunctionSubprogram() { ... }
    public ASTSubroutineSubprogramNode getASTSubroutineSubprogram() { ... }
    public ASTModuleNode getASTModule() { ... }
    public ASTBlockDataSubprogramNode getASTBlockDataSubprogram() { ... }
}
```

List Nodes (Recursive Productions)

Recursive productions are treated specially, since they are used frequently to express lists in the grammar. The recursive member is labeled in the grammar with an @ symbol. For example,

```
<Body> ::=
    <BodyConstruct>
  | @:<Body> <BodyConstruct>
```

indicates that a <*Body*> consists of several <*BodyConstruct*>s. They can be iterated with code such as the following.

```
for (int i = 0; i < astBodyNode.count(); i++)
    doSomething(astBodyNode.getASTBodyConstruct(i);
```

Tokens

`Tokens` form the leaves of the AST. They record, among other things,

- The terminal symbol in the grammar that the token is an instance of (`getTerminal()`)
- The actual text of the token (`getText()`)
- The line, column, offset, and length of the token text in the source file (`getLine()`, `getCol()`, `getOffset()`, `getLength()`)
- Scope and binding (`getScope()`, `getBinding()`, discussed below)

Most of the remaining fields are used internally for refactoring.

3.2 Scope and Binding Analysis

As mentioned above, every `Token` has a `Scope` associated with it, and identifier tokens (`Tokens` for which `token.getTerminal() == Terminal.T_IDENT`), which represent functions, variables, etc. in the Fortran grammar, also have a `Binding` associated with them.[2]

A `Scope` is essentially a collection of `Bindings`. To a typical user of an AST, it is not very useful.

`Binding`, on the other hand, is very useful. `Binding` is an abstract class; every `Binding` is either a `Definition` or a `Reference`. Some `References` are also `Declarations`. Any binding can be resolved to all of its potential definitions by invoking its `resolve()` method (or a similarly-named method). The exact source token can be located by calling the `findSourceToken` method on the `Binding`. (Note that a binding may resolve to a definition in another source file; this can be checked by calling the `getContainerFile` method on a `Binding`.)

For debugging, the `printGlobalSymbolTableOn` method may be invoked on any `Binding` to output that file's symbol table. (Photran provides an item in its Refactoring menu which dumps the symbol table for the file in the editor to standard output using this method.)

[2]The `getScope()` and `getBinding()` methods are declared to return an `Object`; this is because the `Token` class is located in the org.eclipse.photran.core plug-in and compiled using Java 4, so that it can be used to build the Outline view, while `Scope` and `Binding` are located in org.eclipse.photran.core.analysis, compiled using Java 5, which already has a dependency on org.eclipse.photran.core.

3.3 Type Checking

The type of any expression (`ASTExprNode`) can be determined by invoking the static method `TypeChecker.getTypeOf(node)`. At the time of writing, this behavior is not completely implemented. (TODO)

Chapter 4

Refactoring

Last Updated 4/11/07

This information is out of date and is being updated (8/07).

4.1 Structure of a Refactoring

Refactorings in Photran are subclassed from `FortranRefactoring`, which is in turn a subclass of the `Refactoring` class provided by the Eclipse Language ToolKit (LTK).

Refactorings—that is, subclasses of `FortranRefactoring`—must implement three methods:

```
public String getName();
protected abstract void doCheckInitialConditions() throws PreconditionFailure;
protected abstract void doCheckFinalConditions() throws PreconditionFailure;
protected abstract void doCreateChange() throws CoreException, OperationCanceledException;
```

`getName` simply returns the name of the refactoring: "Rename," "Extract Subroutine," "Introduce Implicit None," or something similar.

Initial conditions are checked before any dialog 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 a string is a legal identifier.

The actual transformation is done in the `doCreateChange` method, which will be called only after the final preconditions are checked.

4.2 Checking Preconditions

The `FortranRefactoring` class provides a large number of `protected` utility methods, mostly for precondition testing. Some representative examples follow. This is by no means an exhaustive list, and others will be added as more refactorings are written.

Methods that serve directly as preconditions are often named `ensure...`. For example:

- `protected void ensureIsIdentifierToken(Token t) throws PreconditionFailure`
- `protected void ensureIsValidIdentifier(String name) throws PreconditionFailure`
- `protected void ensureIsBoundIdentifier(Token t) throws PreconditionFailure`
- `protected Definition ensureIsUniquelyDefinedIdentifer(Token t) throws PreconditionF`
- `protected void ensureNameWillNotConflictInScope(Binding binding, String name) throws PreconditionFailure`

Other methods are more primitive and are used to build such precondition checks. For example:

- `void failIf(boolean test, String msg)` throws a `PreconditionFailure` if the given boolean expression evaluates to false.
- `Token findEnclosingToken(IFortranAST ast, final OffsetLength selection)` finds the `Token` containing the given characters in the source file.
- `Token findEnclosingIdentifier(IFortranAST ast, OffsetLength selection)` does the same but throws a `PreconditionFailure` if the token is not an identifier.
- `StatementSequence findEnclosingStatementSequence(IFortranAST ast, OffsetLength selection)` finds the smallest sequence of statements containing the given characters in the source file. (This is useful for, say, Extract Subroutine.)

4.3 AST Rewriting

After it is determined what files are affected by a refactoring, manipulating the source code in the `doCreateChange` method is conceptually straightforward.

1. To change the text of a single token, simply call its `setText` method.
2. To remove part of an AST, call the static method `SourceEditor.cut(node)`, which will remove the given node (and all of its children), returning it.
3. To insert a node into an ast, call one of the `SourceEditor.paste...` methods. The pasted node will be automatically reindented to match its surroundings.

After all of the changes have been made to a file's AST, `addChangeFromModifiedAST(IFile file)` should be invoked to commit the change, after which it is safe to call `FortranWorkspace.getInstance().release....`

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()`, `getScope()`, `getBinding()`, etc.) will return *incorrect* values. This also means that the `find...` methods in `FortranWorkspace`, as well as `findSourceToken` in the `Binding` class will *not* work correctly. Instead, pointers to the nodes and tokens to manipulate should be obtained *prior to* making any modifications to the AST.

4.4 User Interface

Adding a refactoring to the user interface is best done by following an example. 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 in the org.eclipse.photran.refactoring.ui plug-in. Then, the action delegate must be created to populate the user interface of the refactoring wizard dialog; `RenameAction` (in the `org.eclipse.photran.internal.refactoring.ui` package) can be used as a starting point.

4.5 An Example: Rename

The rename refactoring, implemented as `RenameRefactoring` in the `org.eclipse.photran.intern` package, is quite brief and should serve as a model for building future Fortran

refactorings.

Appendix A

Getting the Photran 4.0 Sources from CVS

Last Updated 4/4/07

BEFORE YOU BEGIN: Make sure you are running Eclipse 3.3 and a Java 5 or later JVM.

Part I. Check out the CDT 4.0 sources from CVS

If you already have CDT 4.0 installed and do not need to edit the CDT source code, Part I can be skipped.

1. **In Eclipse, switch to the CVS Repository Exploring perspective.**
2. **Right-click the CVS Repositories view; choose New, Repository Location**
3. **In the dialog box, enter the following information, then click Finish:**

Host name:	**dev.eclipse.org**
Repository path:	**/cvsroot/tools**
Username:	**anonymous**
Password:	**(no password)**
Connection type:	**pserver**

4. **Right-click on :pserver:anonymous@dev.eclipse.org:/cvsroot/tools, and choose Refresh Branches...**
5. **In the dialog box, scroll down, check the box next to org.eclipse.cdt, and click Finish. When prompted, click on Search Deeply. You may have to wait for a few minutes for processing to complete and the dialog to disappear.**

6. Now, in the CVS Repositories view

    - Expand ":pserver:anonymous@dev.eclipse.org:/cvsroot/tools"
    - Then expand "Versions"
    - Then expand "org.eclipse.cdt CDT_4_0_0"
    - Then expand "all"

7. Click on the first entry under "all" (it should be org.eclipse.cdt), then shift-click on the last entry under "all" (it should be org.eclipse.cdt-feature). All of the intervening plug-ins should now be selected. Right-click on any of the selected plug-ins, and select Check Out from the pop-up menu. (Check out will take several minutes.)

8. You now have the CDT source code. Make sure it compiles successfully (lots of warnings, but no errors).

    Part II. Check out the Photran sources from CVS

9. In Eclipse, switch to the CVS Repository Exploring perspective.

10. Right-click the CVS Repositories view; choose New, Repository Location

11. Enter the following information, then click Finish:
    If you are a Photran committer:

Host name:	dev.eclipse.org
Repository path:	/cvsroot/technology
Username/passwd:	(your eclipse.org committer username and password)
Connection type:	extssh
Otherwise:	
Host name:	dev.eclipse.org
Repository path:	/cvsroot/technology
Username:	anonymous
Password:	(no password)
Connection type:	pserver

12. Expand the node for dev.eclipse.org:/home/technology, then expand HEAD (in the CVS Repositories view), then expand org.eclipse.photran

13. Check out the following projects under org.eclipse.photran:

    - org.eclipse.photran.core
    - org.eclipse.photran.core.vpg
    - org.eclipse.photran.core.vpg.tests
    - org.eclipse.photran.errorparsers.xlf

- org.eclipse.photran.intel-feature
- org.eclipse.photran.managedbuilder.core
- org.eclipse.photran.managedbuilder.gnu.ui
- org.eclipse.photran.managedbuilder.intel.ui
- org.eclipse.photran.managedbuilder.ui
- org.eclipse.photran.managedbuilder.xlf.ui
- org.eclipse.photran.ui
- org.eclipse.photran.ui.vpc
- org.eclipse.photran-dev-docs
- org.eclipse.photran-feature

(The debug and launch plug-ins are not part of Photran 4.0 and will not compile. The analysis and refactoring plug-ins have been deprecated; they do not contain any files, since that functionality is in the VPG plug-ins.)

The sources should all compile (albeit with lots of warnings).

Note. ***Some JUnit tests for the parser and refactoring engine require closed-source code that is not available in CVS. A warning will appear in the JUnit runner if this code is not available.***

Appendix B

Creating an Error Parser

Last Updated 4/4/07

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 74 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.er` 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 to not have to be implemented in the Photran Core plug-in. In fact, they do not have to be implemented in 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.

Appendix C

Creating Tests for the Rename Refactoring

Last Updated 4/12/07

This information is out of date and is being updated (8/07).

JUnit tests for the Rename refactoring are located in the org.eclipse.photran.refactoring.tests plug-in. A Rename test has two components:

1. one or more Fortran files, which contain the code to be refactored, and
2. a JUnit test suite class, which creates tests attempting to rename the identifiers in that file.

The Fortran files are stored as .f90 files in the rename-test-code folder. The JUnit tests are similarly-named Java classes in the `org.eclipse.photran.refactoring.tests.rename` package.

A sample JUnit test suite is the following. The more complex tests follow a similar structure. Here, the `vars` array records all of the identifiers and the line/column positions on which they occur. The test suite constructor attempts to rename each identifier to z and also to `a_really_really_long_name`.

```
public class Rename2 extends RenameTestSuite
{
    ///////////////////////////////////////////////////////////////////////////
    //
    // RECORD POSITIONS OF ALL IDENTIFIERS IN RENAME2.F90, AND
    // GROUP THEM ACCORDING TO WHICH ONES SHOULD BE RENAMED TOGETHER
```

```
    //
    ////////////////////////////////////////////////////////////////////////////

    private String filename = "rename2.f90";

    private Ident[] vars = new Ident[]
    {
        var(filename, "Main", new LineCol[] { lc(2,9), lc(27,13) }),
        var(filename, "one", new LineCol[] { lc(4,16), lc(12,14), lc(16,11), lc(20,11) }),
        var(filename, "two", new LineCol[] { lc(5,27), lc(10,13), lc(13,14), lc(17,14) }),
        var(filename, "three", new LineCol[] { lc(6,16), lc(14,9), lc(18,9) }),
        var(filename, "four", new LineCol[] { lc(10,21), lc(15,14), lc(19,14) })
    };

    ////////////////////////////////////////////////////////////////////////////
    //
    // TEST CASES
    //
    ////////////////////////////////////////////////////////////////////////////

    public static Test suite() throws Exception
    {
        return new Rename2();
    }

    public Rename2() throws Exception
    {
        startTests("Renaming program with comments and line continuations");
        for (String name : new String[] { "z", "a_really_really_long_name" })
            for (Ident var : vars)
                addSuccessTests(var, name);
        endTests();
    }
}
```

The `addSuccessTests` method adds several test cases to the suite: it simulates the user clicking on each occurrence of the identifier and asking to rename that instance. (Of course, no matter which occurrence is clicked on, all instances should be renamed... but this has occasionally not happened.)

If the rename should not have succeeded—that is, a precondition would not be met—`addPreconditionTests` should have been called rather than `addSuccessTests`.

`Rename3` is a slightly more complicated example, which renames identifiers spanning multiple files. In this case, a large boolean matrix is used to record which identifiers should be renamable to which other identifiers:

```
    private boolean[][] expectSuccess = new boolean[][]
    {
```

```
// IMPORTANT:
// * Modules can't be renamed, hence the rows of "false" for moduleA, moduleB, and moduleC
// * Everything except myProgram and external should probably be renameable to myProgram, but tl

/* vvv can be renamed to >>>    myProgram, aRenamed3, bRenamed3, contained, external, ...
/* myProgram */ new boolean[] { false,     true,      true,      true,      false,    ...
/* aRenamed3 */ new boolean[] { false,     false,     false,     false,     false,    ...
/* bRenamed3 */ new boolean[] { false,     false,     false,     false,     false,    ...
/* contained */ new boolean[] { false,     false,     false,     false,     false,    ...
/* external  */ new boolean[] { false,     false,     false,     false,     false,    ...
...
```