# Photran 4.0 Developer's Guide

N. Chen
J. Overbey

# Contents

# Chapter 1

# Introduction

Photran is an 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 C/C++ Development Tools (CDT), the Eclipse IDE for C/C++. Previous versions of Photran were created by hacking a copy of CDT to support Fortran instead of C/C++, but now we have developed a mechanism for adding new languages into 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.

## 1.1 How To Read This Guide

This document explains the design of Photran so that interested contributors could fix a bug or add a refactoring. Contributors should know how to use Photran and how CDT works. There is a short *Getting Started Guide* on the Photran website.

Contributors also need to understand Eclipse and how to build Eclipse plug-ins before they read this document. We recommend reading *Building Commercial-Quality Plug-ins* and *The Java Developer's Guide to Eclipse* for Eclipse newcomers.

This is a *duplex* guide. The main chapters provide general descriptions of the various components and how they interact. The appendix describes concrete examples from Photran so that contributors can familiarize themselves with actual code and implementation details. This guide complements the source code in the repository; it is <u>not a substitute</u> for reading the actual source code.

Last significant update: 08/08/08

# Chapter 2

# Interactions with CDT

## 2.1 Introduction

The C/C++ Development Tools (CDT)[1] enables Eclipse to function as a first-class C/C++ IDE. CDT provides features that a programmer expects from an IDE such as project management, automated build, integrated debugging, etc. In addition, CDT also provides extension points for writing IDEs for other programming languages that follow the C/C++ edit-compile-debug-compile cycle closely; Fortran is one such language.

Photran builds upon the CDT by leveraging its extension points. As such, it needs to follow certain conventions and expectations of the CDT. In this chapter, we discuss those conventions and expectations.

## 2.2 CDT Terminology

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

- **Standard Make projects** are ordinary Eclipse projects, except that CDT and Photran recognize them as being "their own" type of project (as opposed to, say, projects for JDT, EMF, or another Eclipse-based tool). Users must supply their own

Last significant update: 08/08/08

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

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). 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 that they recognize, they extract the filename, line number, and error message, and use it to populate the Problems View. See Appendix B 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 derive from the (CDT Core) class `ICElement`. It is described in the next section.

## 2.3 The Model

*This section describes the CDT model in detail. Understanding the CDT model is useful for contributors who are interested in modifying the UI and how Fortran projects are managed in the Fortran perspective. Contributors who are interested in creating refactorings and program analysis tools should familiarize themselves with the Abstract Syntax Tree (AST) and Virtual Program Graph (VPG) described in Chapter 4.*

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) describes 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, 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 4.3. They are **not** identical. It is best to think of the CDT model as containing a *partial/simplified view* of the AST model to represent the *interesting* elements in the source code (program names, function names, subroutine names) **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 language and the resources.* 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 is responsible for initializing concrete classes that will build up the model that CDT expects. For more information, refer to the FortranLanguage.java file in the `org.eclipse.photran.cdtinterface` plug-in project.

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. The default implementation is provided by the `SimpleFortranModelBuilder` class. It relies on simple lexical analysis to build the model. This simple model builder might not be able to accurately handle the more complicated features of the Fortran language.

The Photran VPG (see Section 4.2) inside the `org.eclipse.photran.core.vpg` project uses the first option to contribute a model builder. The relevant classes are under the `org.eclipse.photran.internal.core.model` package i.e. `FortranModelBuilder`, `FortranModelBuildingVisitor` and `FortranParseTreeModelBuildingVisitor`.

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.

## 2.4 Reusing UI Elements

Various UI elements in Photran are also reused from the CDT through subclassing. For instance, the `NewProjectDropDownAction` class shown in Listing 2.1 is a subclass of `AbstractWizardDropDownAction` declared in CDT. `AbstractWizardDropDownAction` provides most of the implementation and our subclass just provides the Photran-specific details such as the actual action that will be invoked.

Our `NewProjectDropDownAction` is invoked through the right-click menu by going to New... > Other > Fortran. It creates a new Fortran project in the current workspace.

In addition, we could also customize the icons for each UI element by modifying the appropriate attributes in the plugin.xml file in the `org.eclipse.photran.cdtinterface`

**Listing 2.1** `NewProjectDropDownAction` class

```java
public class NewProjectDropDownAction extends AbstractWizardDropDownAction
{
  public NewProjectDropDownAction()
  {
    super();
    PlatformUI.getWorkbench().getHelpSystem().setHelp(this,
        ICHelpContextIds.OPEN_PROJECT_WIZARD_ACTION);
  }

  protected IAction[] getWizardActions()
  {
    return FortranWizardRegistry.getProjectWizardActions();
  }
}
```

project.

## 2.5 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 3

# Plug-in Decomposition

## 3.1 Introduction

This chapter presents a high-level overview of the different projects and plug-ins in Photran. It serves as a guide for developers reverse-engineering Photran to *guess-and-locate* where certain components are. It also serves as a guide for contributers on *where* to put their contributions.

The following sections are grouped by feature. A feature is a collection of related Eclipse plug-ins that the user can install as a whole.

## 3.2 Base Photran Feature: org.eclipse.photran-feature

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

- **org.eclipse.photran.cdtinterface**

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

Last significant update: 08/08/08

  - FortranLanguage 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

  For more information about CDT, see Chapter 2.

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

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

- **org.eclipse.photran.ui**

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

  - Fortran Editors
  - Preference pages

## 3.3 Virtual Program Graph (VPG) feature: org.eclipse.photran.vpg-feature

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 in Chapter 4.

- **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)
  - Refactoring/program transformation engine
  - Refactorings

- **org.eclipse.photran.core.vpg.tests**,
  **org.eclipse.photran.core.vpg.tests.failing**

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

  All tests in org.eclipse.photran.core.vpg.tests should pass. Test suites and test cases are placed in the "failing" plug-in project until all of their tests pass.

  These plug-ins *must* be run as a "JUnit Plug-in Test" ()**not** a "JUnit Test"). In the launch configuration, the environment variable TESTING must be set to some non-empty value. (This ensures that the VPG will not try to run in the background and interfere with testing.)

- **org.eclipse.photran.ui.vpg**

  UI contributions that depend on the `org.eclipse.photran.core.vpg` plug-in. Currently, this includes (1) the Open Declaration action, a project property page where the user can customize the search path for Fortran modules and include files, and (2) all of the actions in the Refactoring menu. Search and other VPG-dependent features will be placed here in the future.

## 3.4 XL Fortran Compiler Feature: org.eclipse.photran.xlf-feature

The following are plug-ins to support the XL Fortran compiler.

- **org.eclipse.photran.core.errorparsers.xlf**,
  **org.eclipse.photran.managedbuilder.xlf.ui**

  Support for Managed Build projects using XL toolchains. Managed by Craig Rasmussen at LANL.

## 3.5 Intel Fortran Compiler Feature: org.eclipse.photran.intel-feature

The following are plug-ins to support the Intel Fortran Compiler.

- **org.eclipse.photran.core.intel**, **org.eclipse.photran.managedbuilder.intel.ui**

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

## 3.6 Non-plug-in projects

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 4

# Parsing and Program Analysis

## 4.1 Parsing

Before any program analysis can be done, the source code of the Fortran program has to be parsed. Photran provides support for both fixed-form (Fortran 77) and free-form Fortran (Fortran 90 & 95) source code. The parser in Photran is generated using the Ludwig parser/AST generator by Jeff Overbey. It is based on the *fortran95.bnf* grammar file.

Figure 4.1 shows the lexer & parser tool chain in Photran. Preliminary support for the preprocessor (`INCLUDE` directives) has also been implemented in the current version of the tool chain.

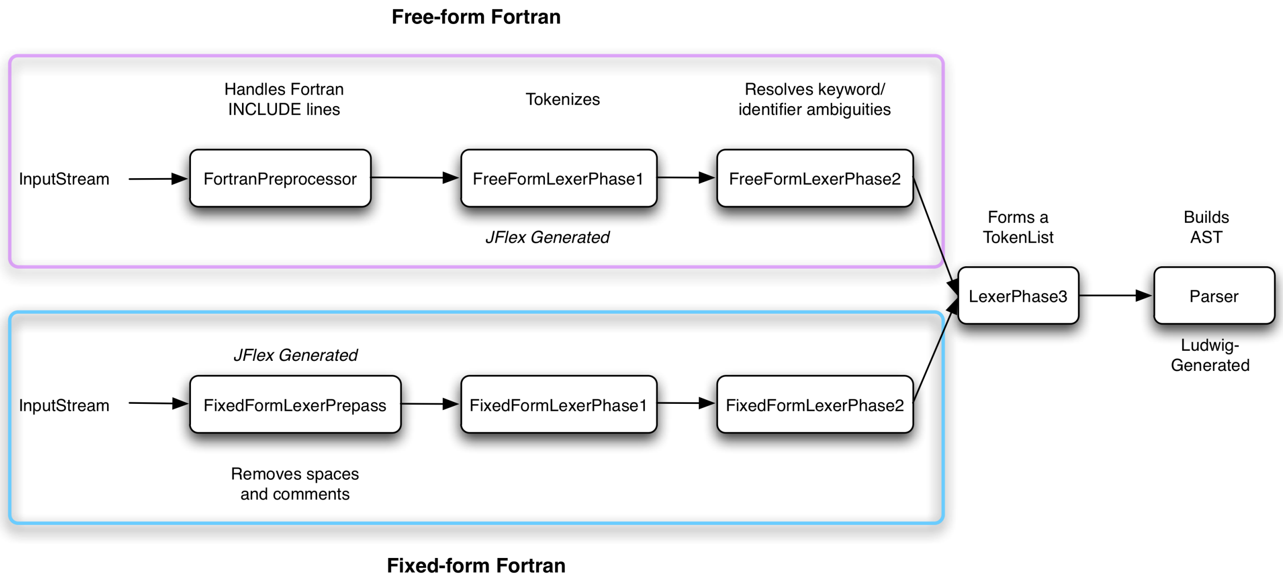
## 4.2 Virtual Program Graph

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

An overview of VPGs is available at the following link: `http://jeff.over.bz/software/vpg/doc/`. It provides the background necessary for the remaining sections of this chapter.

Last significant update: 08/08/08

**Figure 4.1** Photran preprocessor/lexer/parser chain



### 4.2.1 Using the Photran VPG

#### Acquiring and Releasing Translation Units

`PhotranVPG` is a *singleton* object responsible for constructing ASTs for Fortran source code. ASTs are retrieved by invoking either of these methods:

**Listing 4.1** Acquiring the Fortran AST

```
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 4.2** Releasing the Fortran AST

```
public void releaseAST(IFile file)

public void releaseAllASTs()
```

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

**Caveat**

It is important to note that, because `PhotranVPG` is a singleton object, one must be careful about accessing it from multiple threads. Most of the time, when an AST is required, it will be from an action contributed to the Fortran editor; in this case, the action will usually be a descendant of `FortranEditorActionDelegate`, and synchronization will be handled automatically. For instance, all refactoring actions in Photran are descendants of `FortranEditorActionDelegate` and their accesses to `PhotranVPG` are being synchronized automatically.

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` since that is the simplest mechanism (all synchronization is already taken care of automatically). However, if such an approach is not feasible, then they should consider using `VPGSchedulingRule` before resorting to locking the entire workspace.

### 4.2.2 The Program Representation: IFortranAST

The `acquireTransientAST` and `acquirePermanentAST` methods return an object implementing `IFortranAST`. Listing 4.3 shows the methods in IFortranAST.

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.

**Listing 4.3** `IFortranAST` (see IFortranAST.java)

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

  public void accept(IASTVisitor visitor);

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

  public ASTExecutableProgramNode getRoot();

  public Iterator<Token> iterator();
  public Token findTokenByStreamOffsetLength(int offset, int length);
  public Token findFirstTokenOnLine(int line);
  public Token findTokenByFileOffsetLength(IFile file, int offset, int length);
}
```

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. For more information see Section 4.3 on what nodes can be visited.

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))
```

## 4.3 AST Structure

Photran's (rewritable) AST is generated along with the parser, so the structure of an AST is determined by the structure of the parsing grammar (see the *fortran95.bnf* file). The generated classes are located in the `org.eclipse.photran.internal.core.parser` package in the `org.eclipse.photran.core.vpg` project. The easiest way to visualize the structure of a particular file's AST is to view it in the Outline view (see Section 4.5). However determining all possible ASTs for a particular construct requires scrutinizing the parsing grammar file.

### 4.3.1 Ordinary AST Nodes

Generally speaking, there is one AST node for each nonterminal in the grammar and one accessor method for each symbol on its right-hand side (unless the symbol name is prefixed with a hyphen, in which case it is omitted). For example, the following specification[1]

```
# R533
<DataStmtSet> ::= <DataStmtObjectList> -:T_SLASH <DataStmtValueList> -:T_SLASH
```

generates the AST node class `ASTDataStmtSetNode` shown in Listing 4.4. Notice the *presence* of the `getDataStmtObjectList` and `getDataStmtValueList` getters methods and the *absence* of any method for accessing T_SLASH.

> The convention is to generate a class with the name `AST`<nonterminal name>`Node` that extends `ASTNode`. For instance # R533 will generate the `ASTDataStmtSetNode` class.

The following sections describe additional annotations that can be used to modify the standard convention when necessary. These annotations are not considered part of the standard BNF notation but they are supported by the underlying Ludwig parser generator.

**Listing 4.4** `ASTDataStmtSetNode` generated from # R533

```
public class ASTDataStmtSetNode extends ASTNode
{
  public IASTListNode<IDataStmtObject> getDataStmtObjectList() {...}

  public void setDataStmtObjectList(IASTListNode<IDataStmtObject> newValue) {...}

  public IASTListNode<ASTDataStmtValueNode> getDataStmtValueList() {...}

  public void setDataStmtValueList(IASTListNode<ASTDataStmtValueNode> newValue) {...}

  ...
}
```

#### Annotation #1: `(list)`

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 the `(list)` annotation. For example, the following specification

[1] All grammar specifications are taken from the *fortran95.bnf* file. The # RXXX number provides a reference to the actual specification in the grammar file.

```
# R538
(list):<DataStmtValueList> ::=
  |                               <DataStmtValue>
  | <DataStmtValueList> -:T_COMMA <DataStmtValue>
```

means that the AST will contain an object of type `List<ASTDataStmtValueNode>` whenever a `<DataStmtValueList>` appears in the grammar. For instance, # R533 (just described in the previous section) uses the `DataStmtValueList` construct. Notice in Listing 4.4 that the return type of `getDataStmtValueList` is a `List`.

Putting an object that implements the `java.util.List` into the tree (rather than having a chain of nodes) makes it easier to iterate through the list, determine its size and insert new child nodes.

### Annotation #2: `(superclass)`

The `(superclass)` annotation is used to create an interface that is implemented by all symbols on the right-hand side of the specification will implement. For example, the following specifications

```
# R207
(superclass):<DeclarationConstruct> ::=
  | <DerivedTypeDef>
  | <InterfaceBlock>
  | <TypeDeclarationStmt>
  | <SpecificationStmt>

...

# R214
(superclass):<SpecificationStmt> ::=
  | <AccessStmt>
  | <AllocatableStmt>
  | <CommonStmt>
  | <DataStmt>
  | <DimensionStmt>
  | <EquivalenceStmt>
  | <ExternalStmt>
  | <IntentStmt>
  | <IntrinsicStmt>
  | <NamelistStmt>
  | <OptionalStmt>
  | <PointerStmt>
  | <SaveStmt>
  | <TargetStmt>
  | <UnprocessedIncludeStmt>
```

mean that an **interface** – not a class – named `ISpecificationStmt` will be generated for # R214, and `ASTAccessStmtNode`, `ASTAllocatableStmtNode`, `ASTCommonStmtNode`, etc will implement that interface. In addition, because `<SpecificationStmt>` is used inside # R207 which also uses the `(superclass):` annotation, `ISpecificationStmt` also extends the `IDeclarationConstruct` interface from # R207 i.e.

```
public interface ISpecificationStmt extends IASTNode, IDeclarationConstruct
```

So, it is possible for an AST node to implement multiple interfaces based on the specifications in the grammar.

Using the `(superclass)` annotation gives those nonterminals in # R214 nodes a common type; most notably, a `Visitor` can override the `visit(ISpecificationStmt)` method to treat all three node types uniformly.

### Annotation #3: `(bool)`

The `(bool)` annotation indicates that an accessor method will return a **boolean** rather than an actual AST node. For example, the following specification

```
# R511
<AccessSpec> ::=
  | isPublic(bool):T_PUBLIC
  | isPrivate(bool):T_PRIVATE
```

will generate the `ASTAccessSpecNode` class as shown in Listing 4.5.

Notice on lines 8 & 12 that the methods return **boolean** values instead of `ASTNode`s. The **boolean** values are usually used to test the presence of that particular `ASTNode` in the source code.

### Annotation #4: Labels

Specification # R511 also illustrates the use of *labels* in the grammar file: `isPublic(bool):T_PUBLIC` results in a method called `isPublic` instead of `getT_PUBLIC`. The use of labels can greatly enhance the readability of the program by making its intent clearer.

### Annotation #5: `(inline)`

Consider the following specifications for a main program in Fortran:

**Listing 4.5** ASTAccessSpecNode generated from # R511

```
public class ASTAccessSpecNode extends ASTNode
{
  // in ASTAccessSpecNode
  Token isPrivate;
  // in ASTAccessSpecNode
  Token isPublic;

  public boolean isPrivate() {...}

  public void setIsPrivate(Token newValue) {...}

  public boolean isPublic() {...}

  public void setIsPublic(Token newValue) {...}

  ...
}
```

```
# R1101
<MainProgram> ::=
  |               <MainRange>
  | <ProgramStmt> <MainRange>

<MainRange> ::=
  | <Body>              <EndProgramStmt>
  | <BodyPlusInternals> <EndProgramStmt>
  |
```

From the standpoint of a typical Fortran programmer, a main program consists of a Program statement, a body (list of statements), perhaps some internal subprograms, and an End Program statement. This does not match the definition of a `<MainProgram>` in the parsing grammar above: `<Body>` and `<EndProgStmt>` are relegated to a separate `<MainRange>` nonterminal.

The solution is to label the MainRange nonterminal with the `(inline)` annotation, indicating that it is to be in-lined:

```
# R1101
(customsuperclass=ScopingNode):<MainProgram> ::=
  |               (inline):<MainRange>
  | <ProgramStmt> (inline):<MainRange>

<MainRange> ::=
  | <Body>                       <EndProgramStmt>
  | (inline):<BodyPlusInternals> <EndProgramStmt>
  |
```

This means that accessor methods that would otherwise be in a separate `ASTMainRangeNode` class will be placed in the `ASTMainProgramNode` class instead. Listing 4.6 shows that the accessors that were previously in `ASTMainRangeNode` have been in-lined to `ASTMainProgramNode`. Now there is no longer any need for a `ASTMainRangeNode` class.

**Listing 4.6** `ASTMainProgramNode` generated from # R1101

```
public class ASTMainProgramNode extends ScopingNode implements IProgramUnit
{
  public ASTProgramStmtNode getProgramStmt()

  public void setProgramStmt(ASTProgramStmtNode newValue)

  public IASTListNode<IBodyConstruct> getBody()

  public void setBody(IASTListNode<IBodyConstruct> newValue)

  public ASTContainsStmtNode getContainsStmt()

  public void setContainsStmt(ASTContainsStmtNode newValue)

  public IASTListNode<IInternalSubprogram> getInternalSubprograms()

  public void setInternalSubprograms(IASTListNode<IInternalSubprogram> newValue)

  public ASTEndProgramStmtNode getEndProgramStmt()

  public void setEndProgramStmt(ASTEndProgramStmtNode newValue)

  ...
}
```

### Annotation #6: `(customsuperclass=*)`

Specification # R1101 in the previous section also illustrates the use of the `(customsuperclass=ScopingNode)` annotation. This makes `ScopingNode` the parent of the `ASTMainProgramNode` class. Note that `ScopingNode` (or whichever custom super class is chosen) has to be a descendant of `ASTNode` because every node in the AST has to be of that type (either directly or as a descendant).

The `(customsuperclass=*)` annotation is a useful technique for delegating external methods that cannot be expressed through the grammar BNF file into a separate hand-coded class while still having the benefits of an auto-generated parser and AST.

### 4.3.2 Tokens

`Token`s 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(), getFileOffset(), getLength()`)

Most of the remaining fields are used internally for refactoring.

### 4.3.3 Fortran Program AST Example

The previous sections describe the conventions and additional annotations that are used to construct the AST nodes. While the conventions and annotations themselves are simple, the Fortran grammar is extremely complicated and contains hundreds of grammar specifications. Essentially, even the simplest Fortran program might contain a very complicated AST. For instance this simple Fortran program:

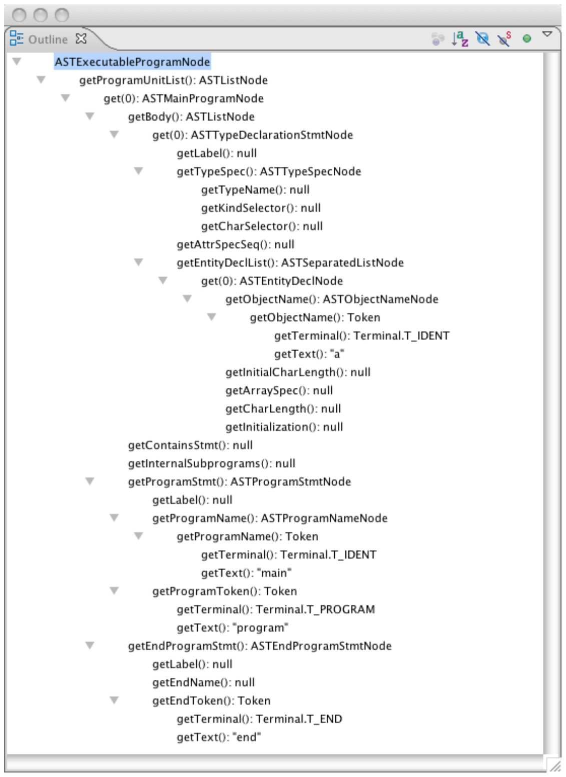
```
program main
    integer a
end
```

generates the AST shown in Figure 4.2. As an exercise, the reader is encouraged to derive the structure of the AST from the grammar specifications in the *fortran.bnf* file beginning with # R201, `<ExecutableProgram>`.

Moreover, modifying our previous program slightly to contain an array as shown below, causes the AST structure to become extremely complicated. The reader is encouraged to create a file containing this program and view the resulting AST in the Outline View in Photran. Notice how the `(5)` expression for specifying the dimension of the array requires building up an entire hierarchy of `ASTLevel5ExprNode`, `ASTLevel4ExprNode`, `ASTLevel3ExprNode`, . . .

```
program main
    integer a(5) ! This declares an array of 5 integers
end
```

**Figure 4.2** AST for simple Fortran program as viewed through the Outline View



```
Outline
ASTExecutableProgramNode
  getProgramUnitList(): ASTListNode
    get(0): ASTMainProgramNode
      getBody(): ASTListNode
        get(0): ASTTypeDeclarationStmtNode
          getLabel(): null
          getTypeSpec(): ASTTypeSpecNode
            getTypeName(): null
            getKindSelector(): null
            getCharSelector(): null
          getAttrSpecSeq(): null
          getEntityDeclList(): ASTSeparatedListNode
            get(0): ASTEntityDeclNode
              getObjectName(): ASTObjectNameNode
                getObjectName(): Token
                  getTerminal(): Terminal.T_IDENT
                  getText(): "a"
              getInitialCharLength(): null
              getArraySpec(): null
              getCharLength(): null
              getInitialization(): null
      getContainsStmt(): null
      getInternalSubprograms(): null
      getProgramStmt(): ASTProgramStmtNode
        getLabel(): null
        getProgramName(): ASTProgramNameNode
          getProgramName(): Token
            getTerminal(): Terminal.T_IDENT
            getText(): "main"
        getProgramToken(): Token
          getTerminal(): Terminal.T_PROGRAM
          getText(): "program"
      getEndProgramStmt(): ASTEndProgramStmtNode
        getLabel(): null
        getEndName(): null
        getEndToken(): Token
          getTerminal(): Terminal.T_END
          getText(): "end"
```

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. If that is insufficient, then it is usually enough to construct a Visitor to visit *only* the nodes of interest and "collect" the information that is required.

## 4.4 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[2]
- ASTMainProgramNode
- ASTModuleNode
- ASTSubroutineSubprogramNode

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

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

```
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[3]. 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 5.4.1 for a comparison between `Token` and `TokenRef`.

### 4.4.1 Examples of Binding Analysis

#### 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:

[2] 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.

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

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

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

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

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

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

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

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

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

### 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

## 4.5 How to Get Acquainted with the Program Representation

*All these features work on* ***Fortran projects****. A new Fortran project can be created via File > New > Fortran > Fortran Project. These features do not work on individual Fortran source files.*

### 4.5.1 Visualizing ASTs

Photran can display ASTs in place of the ordinary Outline view. This behavior can be enabled from the Fortran workspace preferences:

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

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

### 4.5.2 Visually Resolving Bindings

This feature is disabled by default since it requires performing an update on the VPG database each time the file is saved – an operation that could be expensive for larger files. To enable this feature, right-click on the current Fortran project folder (**not** the individual Fortran source file) in the Fortran Projects View and select "Properties". In the dialog that pops-up, navigate to Fortran General > Analysis/Refactoring. Select the "Enable Fortran Declaration view" checkbox.

Then, in a Fortran editor, click on an identifier (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.

### 4.5.3 Visualizing Enclosing Scopes

Click on any token in the Fortran editor, and click Refactor > (Debugging) > Select Enclosing Scope. The entire range of source text for that token's enclosing `ScopingNode` will be highlighted.

### 4.5.4 Visualizing Definitions

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.

# Chapter 5

# Refactoring

## 5.1 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 1, 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 4) 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

Last significant update: 08/08/08

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

provided by the VPG.

## 5.2 Structure of a Fortran 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)[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. 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 5.1** Abstract methods of `FortranRefactoring` 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

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

`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 5.3.

The `FortranRefactoring` 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 `FortranRefactoring` before writing your own utility methods.

## 5.3 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 4.2.

### 5.3.1 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 5.2** Use of `setText` in `RenamingRefactoring` (see RenameRefactoring.java)

```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 5.4 shows an example of the `removeFromTree` method.

**Listing 5.3** AST manipulation methods in `IASTNode` (see Parser.java) that all AST nodes implement

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

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 4.3.1.

**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 `FortranRefactoring` 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 5.4.

### 5.3.2 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 5.4** 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();
}
```

## 5.4 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 `Token`s (`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 `TokenRef`s) *prior* to making any modifications to the AST. In general, ensure that all analysis (and storing of important information from `Token`s) should be done in the `doCheckInitialConditions` and `doCheckFinalConditions` methods of your refactoring before the `doCreateChange` method.

### 5.4.1 `Token` or `TokenRef`?

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

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

For most refactorings, using either `Token`s or `TokenRef`s 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 `TokenRef`s would minimize the number of ASTs that must be in memory.

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

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

## 5.5 Examples

The "Rename", "Introduce Implicit None" and "Move COMMON To Module" refactorings found in the `org.eclipse.photran.internal.core.refactoring` 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 C. 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.

## 5.6 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 `FortranRefactoring`. 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 6

# Photran Editors

## 6.1 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.

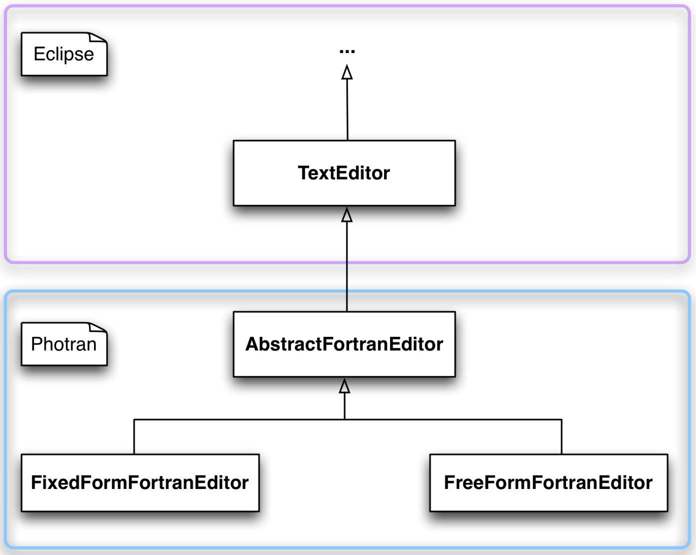
Fortran 90/95 adopted the free-form format that most programmers today are accustomed to. Nonetheless, because Fortran 77 is still considered a subset of Fortran 90/95, it is possible to write programs following the fixed-form format. As such, the free-form editor maintains some visual cues on column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

The UML class diagram in Figure 6.1 shows the hierarchy of the editors in Photran.

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

Last significant update: 08/08/08

**Figure 6.1** Photran editor class hierarchy



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 1.

## 6.2 Contributed `SourceViewerConfiguration`

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

Photran 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 `FortranVPGSourceViewerConfigurationFactory` and the default (but less feature-full) `FortranModelReconcilingSourceViewerConfiguration`.

## 6.3 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 4.

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 6.1** 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

# Getting the Photran 4.0 Sources from CVS

*BEFORE YOU BEGIN: Make sure you are running **Eclipse 3.4** and a **Java 5** or later JVM. We recommend the Eclipse for RCP/Plug-in Developers Package.*

If you already have CDT 5.0 installed and do not need to edit the CDT source code, Part I can be skipped.

**Part I. Check out the CDT 5.0 sources from CVS**

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. In the CVS Repositories view
   - Expand ":pserver:anonymous@dev.eclipse.org:/cvsroot/tools"
   - Then expand "HEAD"

Last significant update: 08/08/08

5. Right-click on “org.eclipse.cdt”
6. Select “Configure Branches and Versions. . . ”
7. Under “Browse files for tags”, expand “all”, then expand “org.eclipse.cdt”, then click on the .project file
8. Under “New tags found in the selected files”, click on the Deselect All button, then check cdt_5_0 in the list above it
9. Click Add Checked Tags
10. Click OK
11. Now, in the CVS Repositories view
    - Expand “:pserver:anonymous@dev.eclipse.org:/cvsroot/tools”
    - Then expand “Branches”
    - Then expand “cdt_5_0”
    - Then expand “org.eclipse.cdt cdt_5_0_0”
    - Then expand “all”
12. 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.ui.tests). 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.)
13. 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**

14. In Eclipse, switch to the CVS Repository Exploring perspective.
15. Right-click the CVS Repositories view; choose New, Repository Location

16. 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 |

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

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

    - org.eclipse.photran-dev-docs (if you intend to contribute to the documentation)
    - org.eclipse.photran.cdtinterface
    - org.eclipse.photran.core
    - org.eclipse.photran.core.intel
    - org.eclipse.photran.core.vpg
    - org.eclipse.photran.core.vpg.tests
    - org.eclipse.photran.core.vpg.tests.failing
    - org.eclipse.photran.errorparsers.xlf
    - 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.vpg

    (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).

## Part III. Running the test cases

19. In Package Explorer view, select the `org.eclipse.photran.core.vpg.tests` project.
20. Right-click on that project and select Run As > Run Configurations.... A dialog will appear.
21. In that dialog, create a new **JUnit Plug-in Test** launch configuration. Call it “Photran-Tests”.
22. For the configuration that you have just created, switch to the “Environment” tab and create a new variable called “TESTING” with a value of 1.
23. Select “Run” to run the tests. To run the tests again, just launch the “Photran-Tests” configuration from the Eclipse Run menu.

    ***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.*

## Part IV. Deploying Photran Feature

24. If you are interested in creating a *deployable feature* for Photran, you also need to check out these **additional** four projects from CVS:
    - org.eclipse.photran-feature
    - org.eclipse.photran.intel-feature
    - org.eclipse.photran.vpg-feature
    - org.eclipse.photran.xlf-feature
25. In Eclipse, select File > Export...
26. In the dialog that pops-up, select Plug-in Development > Deployable features.
27. Click next.
28. In the list, select
    - org.eclipse.photran_feature (4.0.4)
    - org.eclipse.photran.intel (4.0.4)

- org.eclipse.photran.vpg_feature (4.0.4)
- org.eclipse.photran.xlf_feature (4.0.4)

29. Specify a destination folder to export those features. Click Finish.
30. The Photran features are ready for deployment.

# Appendix B

# Creating an Error Parser

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:

Last significant update: 08/08/08

```
<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 C

# Simple Fortran Refactoring Example

## C.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://subversion.cs.uiuc.edu/pub/svn/FruitFly/edu.uiuc.nchen.obfuscator/trunk/`

## C.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.

Last significant update: 08/08/08

`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.

## C.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 C.1 shows how the constructor needs to be setup.

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.

**Listing C.1** `ObfuscateAction` for our simple refactoring example

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

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

    ...
}
```

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 C.2 shows how to add a dummy input page.

**Listing C.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.");

  }
    });
}
```

## C.4 Creating the Actual Refactoring

Section 5.2 gives a good overview of the **four** methods that a Fortran refactoring needs to implement. And Section 5.4 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 5.4, 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 D

# Creating Tests for the Rename Refactoring

JUnit tests for the Rename refactoring are located in the `org.eclipse.photran.refactoring.tests` plug-in project. 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`.

Last significant update: 08/08/08

Because our strategy for testing requires the *exact* line and column position, using tabs instead of spaces for indenting could interfere with the positioning. Therefore, for testing purposes, test cases (the Fortran test files) should be indented with **spaces only**. However, when an actual Fortran programmer invokes a refactoring through the Eclipse UI, the indentation with tabs or spaces is not a problem because the Eclipse editor is smart enough to provide the correct position (based on expanding tabs into spaces internally).

```java
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`. A good testing strategy ensures that a program behaves correctly: it should do **only** what it is supposed to do and nothing more. In our case, it should rename only the identifiers that are affected and ensure that the other identifiers are left untouched.

`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:

**Listing D.1** Partial representation of the boolean matrix in `Rename3`

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

/* vvv can be renamed to >>>   myProgram, aRenamed3, bRenamed3, contained,
/* myProgram */ new boolean[] { false,   true,      true,      true,   ...
/* aRenamed3 */ new boolean[] { true,    false,     false,     false,  ...
/* bRenamed3 */ new boolean[] { true,    false,     false,     false,  ...
/* contained */ new boolean[] { true,    false,     false,     false,  ...
/* external  */ new boolean[] { false,   false,     false,     false,  ...
/* moduleA   */ new boolean[] { false,   false,     false,     false,  ...
/* aSub1of3  */ new boolean[] { true,    false,     false,     false,  ...
/* aSub2of3  */ new boolean[] { true,    false,     false,     false,  ...
/* aSub3of3  */ new boolean[] { true,    true,      true,      true,   ...
/* moduleB   */ new boolean[] { false,   false,     false,     false,  ...
/* bSub1of3  */ new boolean[] { true,    true,      true,      true,   ...
/* bSub2of3  */ new boolean[] { true,    false,     false,     false,  ...
/* bSub3of3  */ new boolean[] { true,    true,      true,      true,   ...
/* moduleC   */ new boolean[] { false,   false,     false,     false,  ...
/* cSub      */ new boolean[] { true,    true,      true,      true,   ...
};
```