Universidad Nacional de La Plata

Facultad de Informática

Master of Science Thesis

Fortran Refactoring for Legacy Systems

Author: Mariano Méndez

Supervisor: Dr. Fernando G. Tinetti

Co-Supervisor: Dra. Alejandra Garrido

"Tesis presentada para obtener el grado de Magister en Ingeniería de Software"

"Facultad de Informática - Universidad Nacional de La Plata" Marzo 2011

To Melina, who changed my life

Contents

Contents **i**

1 Introduction **3**

1.1 Background 4

1.1.1 Refactoring 5

1.1.2 Reverse Engineering and Reengineering 6

1.1.3 Legacy Systems 6

1.1.4 Refactoring applied to Legacy Systems 9

1.2 Initial Motivation 9

1.3 Contributions 10

2 Related Literature and Theoretical Focus **13**

2.1 Related Work 13

2.2 Restructuring 15

2.3 Refactoring Tools 17

2.3.1 Integrated Development Environment 18

2.3.2 Refactoring Tools 19

3 The Fortran Language **35**

3.1 A Complex Evolutionary Process 35

3.1.1 FORTRAN I 36

3.1.2 FORTRAN II 37

3.1.3 FORTRAN III 38

3.1.4 FORTRAN IV 38
3.1.5 FORTRAN 66 40
3.1.6 FORTRAN 77 41
3.1.7 Fortran 90 44
3.1.8 Fortran 95 47
3.1.9 Fortran 2003 48
3.1.10 Fortran 2008 49
3.1.11 Fortran Evolution 50

4 Fortran Refactoring 51
4.1 Different Viewpoints 51
4.2 A Classification of Fortran Refactorings 52
4.3 A Catalog of Fortran Refactoring 53
4.4 Refactorings to Improve Maintainability 53
4.4.1 Refactorings to Improve Presentation / Readability 54
4.4.2 Refactorings to Facilitate Design/Interface Change 59
4.4.3 Refactorings to Avoid Poor Fortran Coding Practices . . . 67
4.4.4 Refactorings to Remove Outdated and Obsolete Constructs 81
4.5 Performance Refactorings 95
4.5.1 Refactorings For Performance 95
4.6 Differences Between Fortran and Other Languages Refactorings . . 101

5 Photran:A Refactoring Tool for Fortran 103
5.1 The Architecture 105
5.2 Photran Core 105
5.3 The Program Representation 108
5.4 Refactoring Infrastructure 109

6 Refactoring Examples 111
6.1 Initial Steps 111
6.2 Transform CHARACTER* to CHARACTER(LEN =) 113
6.2.1 Inception 113

6.2.2 The Design 114
6.2.3 The Implementation 115
6.3 Standardize Input Output Formats 125
6.3.1 Inception 125
6.3.2 The Design 126
6.3.3 The Implementation 128
6.4 Replace Old Style Do Loops Refactoring 141
6.4.1 Inception 141
6.4.2 The Design 141
6.4.3 The Implementation 142
6.5 Remove Unreferenced Labels Refactoring 156
6.5.1 Inception 156
6.5.2 The Design 156
6.5.3 The Implementation 156

7 Case Study 163

7.1 A Unit of Measurement 163
7.2 Source Code Examples 164
7.2.1 Method 166
7.2.2 Example 1 166
7.2.3 Example 2 171
7.2.4 Future Applications 173

8 Conclusions 175

8.1 Results 175
8.2 Future Work 177

Bibliography 179

List of Figures 187

Acknowledgements

I would like to thank my advisors, Dr. Fernando G. Tinetti and Dra. Alejandra Garrido at the Universidad Nacional de La Plata, for their support, patience and guidance. I thank them for giving me this great opportunity and for the chance to do once more what I love most.

I thank Jeffrey Overbey *"El padre de la creatura"* at University of Illinois for his invaluable help with Photran and for his patience.

I would like to thank Dr. Ralph Johnson for investing some of his valuable time in giving me his viewpoint about Fortran Refactorings.

I want to thank the people at the Facultad de Informática de la Universidad de La Plata for not depriving me of the chance to embark on this project.

I thank Monica Lopez and Gustavo Cajaraville.

I thank Mariano Nastri for his support and patience.

I thank my grandfather Joseph for being an example to me. I would like thank Melina for her patience and support.

Finally, I am thankful to life for having granted me the skills and opportunities that made this possible.

Chapter 1

Introduction

Software Refactoring has become a mature discipline in the field of Software Engineering throughout many years [57]. It is deeply ingrained at different phases of software development process, such as modeling, programming, maintenance, etc. Although refactoring has rooted in the software maintenance process, the demand for automatic tools is by far larger than the supply. In fact the amount of this kind of tools offered is very small in comparison to the broad spectrum of programming languages .

Ten years ago a good refactoring tool needed a fast syntactical analyzer and an efficient search & replace engine [34]. These two features will not suffice on these days. Today a good refactoring tool needs also a well proven user interface, integration capabilities with existent IDEs, large or huge scale project support, user support channels, good documentation, content assist, refactoring assist, collaborative or team features, etc. The requirements for a refactoring tool are more complex than years ago.

Building a refactoring tool is not easy, because having good syntactical, lexical or analytical programming language engine is not enough. Moreover, the usability and learnability of the tool must be ensured.

Born as an object oriented programming concept, software refactoring has enlarged its frontiers to structured programming [34, 36] through years. Viewing

a program language like a statical entity is advisable. As software, programming languages are dynamical entities, they are constantly mutating and evolving, at different time scales [68].

At least two forces rule the construction of a refactoring tool. As for the special refactoring requirements that language programming entails, it could be stated that they are closely dependent on the evolution of the programming language through time. The other force involved in this process lies in the end-user requirements. Both of these forces are deeply connected. Therefore, a refactoring tool is the result of hard work among programmers on two sides, those who make the tool and those who use it.

As well as refactoring, programming languages evolve. In their evolution they walk through a wide range of evolutionary processes. Some of these processes impact on software. A great example to analyze is Fortran, a fifty-year-old programming language with a large number of software applications developed through years. Most of the Fortran software is legacy software. Legacy software is hard to maintain and understand because those who have to maintain the systems are not the same who created them. In this work we propose refactoring as a technique to understand, to comprehend, to upgrade, to modify and to add changes on legacy software.

This chapter is organized as follows: First, it presents a background where a description of refactoring foundation concepts and how it has evolved through time is shown. Second, legacy systems are described . Third, an introduction on why refactoring is a perfect tool to be applied in this kind of systems is displayed. Finally, as a conclusion for this introduction, the chapter offers the motivation and contributions of this research.

1.1 Background

Refactoring was born as a result of research into how to make reusable software. Dr Ralph Johnson [45, 60, 61, 44] at University of Illinois is a pioneer of this research field. He has been a major refactoring promoter. This concept has also

been broadly supported by Extreme Programming (XP) followers. Dr. Johnson research group has been studying refactoring "as a way of making reusable software" [43] while XP followers have a different viewpoint of refactoring. They recommend refactoring because it made code more understandable and allow a rapid development process and simple code structure while maintaining clean, scalable, and modular code. These two approaches, though different, seem to be connected. Simpler software tends to be more understandable and reusable as a consequence.

1.1.1 Refactoring

The refactoring concept has been introduced for the first time by W. Opdyke and Ralph Johnson [60]. In his thesis, W. Opdyke [59] introduces refactoring as behavior-preserving transformation, he describes refactoring as a process that improves the design of "already structured programs" making these programs reuse-prone. This approach extends the refactoring concept beyond object oriented programming. He has been the first to propose a wide catalog of many different refactorings. This catalog is divided into Basic Refactorings and Complex Refactorings (two or more basic refactorings). There are many definitions of refactoring. The most accepted one is found in Martin Fowler's Book [32], which defines refactoring as a technique for restructuring existing source code applying small transformations on the source code, modifying its internal structure and preserving the external behaviour of the software. Thus, we can think about refactoring as a process and as technique at the same time.

Trapped by its own essence (changeability, conformity, intangibility and complexity) software [16] has to deal with:

- Evolution: Like other human products, software has to change according to people's needs [47]
- Redesign: Unlike other human products, software can be modified even when its development process has been finished. If we take a closer look at

manufactured products, like a pen: once it has been produced, the pen can not be changed and it will remain the same until the end of its days.

- Quality: As a consequence of its evolution and redesign, the software quality will be undermined [28].

Born out of object oriented programming as a way to make OO programs reusable, this concept has gone beyond its borders, reaching other fields of software engineering like structured programming [34].

1.1.2 Reverse Engineering and Reengineering

As Chikosfky describes in "Reverse engineering and design recovery: A taxonomy", the field of the reverse engineering is centered on the process of analyzing a system in order to identify the system's components and how they are related and to create a representation of the system in another form or on a high level abstraction. On the other hand, Reengineering is the process that examines and alters a system to reconstitute it in a new form [21]. Following this classification, refactoring belongs to Reengineering. As a first step, in order to alter or apply change in a system, it is mandatory to have an understanding of it. Then, in the reengineering process, it is required to have some form of reverse engineering together with forward engineering [21].

1.1.3 Legacy Systems

The word legacy has its origins in the Old French term: *legacie* from Latin *Legatus*, it means "person delegated". In the Oxford compact dictionary the word legacy has been described as an adjective (of computer hardware or software) that has been superseded but is difficult to replace because of its wide use [26] (2010). There is not a formal definition of what a Legacy System is. However, we can find different approximations about what Legacy Systems are. Brodie and Stonebraker have defined a Legacy System as:

"*Any information system that significantly resists modification and evolution to meet new and constantly changing business requirements.*" [15].

K. Benneth proposed:

" *large software systems that we don't know how to cope with but that are vital to our organization.* "[13]

Nicolas Gold, summarizes the Legacy System concept as follows:

"Legacy Software is critical software that cannot be modified efficiently." [37]

These definitions have various things in common. One of them is the resistance to change, another one is how this kind of software has become critical to the Organization. An important concept that goes hand in hand with these definitions is the inherent complexity of legacy software.

There is an aspect where legacy software becomes a challenge, it is the maintenance stage. In this stage, the software that has been running in production for 20 or 30 years is hard to manage because software gradually deteriorates. During the maintenance a program may need different types of changes. Enhancements, corrections, adaptations and preventions to a system may be needed. All of these tasks require knowledge and comprehension about the system. It becomes another aspect where legacy becomes also a challenge. Furthermore, there are other factors to be considered regarding legacy software, such as :

- The programming language in which it has been implemented.
- The state-of-the-art software engineering techniques used when it has been created.
- The crucial task performed in the organization.
- The system size, generally medium or large.

- When, where, why, how ,and who implemented the software.

Thus, legacy systems are ruled by many different components and different perspectives. R. Center proposed a set of Re-engineering perspectives that can be used to deal with Legacy Software [19]. In his work, Center defines 5 perspectives:

1. *Engineering perspective* in which Legacy Systems are viewed as an engineering problem.

2. *Software perspective* related to two activities, program understanding and software evolution which make it possible to gather more comprehensible abstract representations.

3. *Managerial perspective* this viewpoint is responsible for planning, setting goals and determining organizational readiness.

4. *Evolutionary perspective* proposes a new view on software life-cycle, where a continuous evolution model is shown, breaking the old "develop then maintain" model.

5. *Maintenance perspective* where an analysis is conducted from a distinct viewpoint as regards the software developing process.

Two of these perspectives are closely related to refactoring tools: the maintenance perspective and the evolutionary perspective.

This thesis is based on a certain type of legacy software that came from scientific research. Scientists have become one of the most important legacy code producers for many reasons. One of these is long-lived field (about 50 years old) they have been working in. Another reason is the amount of code produced through years and the lack of a well-defined software development process. [75]

Currently, there are not automated tools to maintain,upgrade ,or modify legacy software. Languages like Fortran and Cobol have not such kind of automated tools. Chapter 2 reviews refactorings tools for other languages such C, C#, Java, Smalltalk, Haskell ,and Others.

1.1.4 Refactoring applied to Legacy Systems

Legacy system makes refactoring process and techniques one of the best options to be applied in the context of maintenance tasks, changes ,and understanding Legacy Software [66, 34, 67]. Automated refactoring tools provide programmers with not only an easy way to make changes. But also, with a broad understanding about the system, in order to reduce maintenance costs. Additionally, it serves to keep systems updated within programming language evolution and to extend system's functionality [59]. Finally, it contributes to decrease some complex aspects which are deep-rooted in software essence [16].

1.2 Initial Motivation

The motivation of this work comes from a Global Climate Model (GCM) Software which was in great need of being updated. This software was implemented by scientists in the '80s as a result of meteorological research [74]. Written in Fortran 77, this program has been used as an input to make climate predictions for the Southern Hemisphere. The execution to get a complete numerical data set takes several days. This software has been programmed using a sequential processing paradigm. In these days, where multicore processors are so widespread, the time that an execution takes to get a complete useful data set can be drastically reduced using this technology. As a first objective to reach this goal of reengineering we must be able to understand the source code. An essential Fortran code characteristic is that old source code versions became unreadable, not comprehensive and sometimes "ejects" the reader from the source code. In that way, we can not modify, update or improve unreadable source code. Then, as a first step to parallelize this code we must update it, turn it readable and easy to understand.

The GCM has a very complex internal structure. The program is divided into about 300 .f (Fortran 77) files [74]. These files generally implement only one Fortran subroutine. Less than 10% of the files are used for common blocks and constants. Approximately 25% of the lines in the source code are comments. The

total number of Fortran source code lines is 58000. A detailed work within the source code brings to light that [74]:

1 About 230 routines are called/used at run time. Most of the runtime is spent in routines located at deep levels 5 to 7 in the dynamic call graph from the main routine.

2 The routine with most of the runtime (the top routine from now on) requires more than 9% of the total program runtime and is called about 315000 times.

3 The top 10 routines (the 10 routines at the top of the flat profile) require about 50% of total runtime. Two of them are related to intrinsic Fortran functions.

Our first approach was using a scripting language and Find & Replace tools trying to upgrade the source code, this kind of code manipulation do not guarantee preservation of software behavior.

Then, our goal was to develop an automated tool to transform legacy software in more understandable, comprehensible and readable applying refactoring as main technique. At the same time a catalog of transformation to be applied in Fortran code is needed as a guide to programmers through this process.

1.3 Contributions

The major general contributions, independent of the previous example, are:

1. **A Classification of Fortran refactorings**: The way in which the refactorings were proposed is the result of how we think programmers need to use refactoring in their daily work. So we present the refactorings classified from the programmer's point of view.

2. **A Detailed catalog of Fortran refactorings**: Each refactoring proposed in this catalog has emerged from the Fortran programmer's needs. Our description rests on each refactoring motivation.

3. **A proposal of refactorings for parallelizing and performance improvements**: For some of these refactorings it has been proved that a much better performance existed [72]. A set of these transformations are closely related to those conducted by compilers to improve performance, like loop fusion or loop fission [27].

4. **A specification of some refactorings**: The implementation of a set of refactorings was explained in detailed and documented with the aim of providing a guide to be used in the initial steps in the refactoring built process.

5. **The use of refactorings on Fortran legacy systems**: In this work we have shown how to employ refactorings in the field of legacy systems. Furthermore, we have used refactoring applied to one of the most long-lived programming language such as Fortran.

6. **A metric definition**: We have presented a way to measure the source code transformation impact on source code readability as a metric called "FCRCS".

7. **A Contribution to Photran Project**: The refactorings implemented in this thesis will be all included in Photran 7.0 release.

8. **A public web site containing the catalog in different languages**: Aligned with the aims of this research, a public access web site was created to integrate and to promote Fortran refactorings and the eclipse-based-refactoring tool (Photran). This site was published in July 2010 [3].

The next chapter presents an overview of previous works and related areas to provide a starting point about refactoring in the field of Legacy Systems. Chapter 3 provides a detailed description of the evolutionary process of Fortran Language throughout its lifetime. Chapter 4 proposes a classification of Fortran Refactorings and it specifies in detail the complete catalog of Fortran Refactorings. Chapter 5 describes Photran refactoring tool and IDE core. Chapter 6 shows

thoroughly how some refactorings can be implemented in Photran. Chapter7 applies some refactorings into real life source code examples and introduces a metric to measure source code improvements. Chapter 8 summarizes the contributions of this research and describes its futures applications.

Chapter 2

Related Literature and Theoretical Focus

The first section of this chapter makes a review about refactoring applied to structured programming language. In section 2.1 related works are presented. Section 2.2 describes transformation tools and techniques. Section 2.3 make a thoroughly detail of refactoring tools up to date.

2.1 Related Work

The concept of code restructuring has existed for many years now, and some transformation tools have been built to apply transformation rules on a complete program in batch mode. An example of this kind of infrastructure is the D.M.S. tool, which allows for re-engineering and migration of programs in many different programming languages [12].

In the case of Fortran, the vast amount of existent lines of Fortran code and the investment made on them has encouraged the development of some tools to upgrade legacy Fortran code to new standards. Greenough and Worth have reported a number of software tools currently available that may apply transformations on Fortran programs [38]. There are at least two important reasons of why these tools have not been widely used. First, applying some transformation

rules in batch mode may help updating the code by replacing outdated constructs (e.g., replacing obsolete operators), but that does not necessarily imply that a developer will gain a better understanding of the structure of code, nor will it be able to clean it, modularize it or remove duplication. That is, legacy code will still be legacy even if it is written in Java but with poor development practices. Second, these transformation tools are not integrated with development environments.

The concept of refactoring as an interactive process performed by an expert programmer while carefully examining the code, in small and safe steps, was defined in Opdyke's thesis many years ago, in this work refactoring is presented in the context of Object Oriented Programming [59].

Since that time, Ralph Johnson's research group at the University of Illinois has promoted refactoring and the development of automated refactoring tools [60, 59, 61, 43, 35] , although it was not until the advent of agile methodologies that refactoring received widespread attention.

Garrido is the first author who has introduced refactoring concept to structured programming [34]. Her work is based on refactoring C programs [35, 36]. In her PhD. thesis Garrido presented an algorithm to handle C preprocessor directives. Specifically for Fortran, Vaishali De's master's thesis [25] enumerates a set of possible Fortran 90 refactorings. Later on, Overbey et al. [66] bring to light the need of refactoring tools integrated with IDEs for Fortran programs and in the High Performance world. In this work, Photran is introduced as an integrated development environment that provides the necessary infrastructure for implementing Fortran refactoring [5].

In a subsequent work [67], a study founded on the Fortran evolution enumerates outdated language constructs that a refactoring tool could help remove from Fortran code and proposes, more generally, a role that refactoring tools could play in language evolution. As an example, Photran was used to eliminate global variables.

Tinetti et al. [74] base their work improving Fortran legacy source for performance optimization on a weather climate model implemented about two decades

ago.

2.2 Restructuring

Restructuring can be defined as a subfield of Software Engineering that devoted its research to improving existent source code on the basis of applying source code transformation. The origins of Restructuring rest on " the modification of software to make it easier to understand and to change, or less susceptible to error when future changes are made" [6]. It is worth mentioning that Arnold's definition excludes restructuring for any other purpose, like the improvement of the source code with the aim of a better performance, the transformation of the code for parallelizing, and so foth.

In view of this, restructuring can be seen as a tool that can assist in solving the significant problems that arise during the maintenance stage within the life cycle of the software development process. Further reasons exist according to Arnold as to why source code must be borne in mind by software engineers:

- Reinforcing understandability of software by injecting software with known and easily decipherable structure, thus having other desirable side effects:
  - simpler documentation,
  - simpler testing tasks,
  - simpler auditing tasks,
  - substantial reduction of software's complexity .
- Reducing the necessary time for programmers to get acquainted with the system before implementing maintenance tasks.
- Making bugs easier to locate.
- Making it simple to introduce new functionalities

Software restructuring was born as a necessary tool to be implemented in the maintenance processes because of the essential features of software so as to

reduce development costs. It also can serve as a tool to introduce new software functionality.

Restructuring's main objective mainly consists in preserving or increasing software value. External software value can be increased by fully satisfying users' needs. For users, a good maintenance service means having software which is bugs free (or contains no visible bugs) together with a swift response in the case of a request for change. Systems being subjected to constant maintenance may grow increasingly difficult to change. If this hardening should affect the users' perception of the software's quality, the external software value will decrease.

The internal software value may be measured by the following criteria: 1) the maintenance cost saving resulting from some other software form, 2) the cost savings emerging from reusing parts of the software in other systems, and 3) the cost savings as a consequence of an expanded software lifetime.

Source code Restructuring reduces cost of maintenance, aligned to this it increases software re-usability and it extends system's life cycle. In this way internal software value is being increased too [6].

Another definition of software restructuring can be found in the literature like introduced by Chikofsky and Cross [20]: "Restructuring is the transformation from one representation to another at the same relative abstraction levels, while preserving subject system's external behavior (functionality and semantics)". A restructuring transformation is basically applied on the system's appearance, its aim is not to introduce new requirements.

Arnold introduces some software restructuring techniques [6] listed below:

1. Code Oriented Techniques:

   a **Programming Style**: the intent of this technique is to apply code transformation to make code more understandable:

   - *Pretty printing and code formatting*: Code is improved applying tabbing, spacing and one code per statement.
   - *Coding Style standardization*: The source code is modified to make it compliant to a specific code standard.

- *Restructuring with Preprocessor [46]*: Source code is replaced with preprocessor directives easier to understand.

c Control Flow: One of most complex restructuring techniques. Control flow transformations allow the implementation of a large number of tools.

- *Go-to removing*: In the following list may be found a set of techniques that aim to remove Goto statement.
  - *Early goto-less approach [14].*
  - *Giant case statement approach [7].*
  - *Boolean flag approach [76].*
  - *Duplication of coding approach [76].*
  - *Baker's graph-theoretic approach [11].*
  - *Refined case statement approach [50].*
- In the list below are listed a set of restructuring tools :
  - *RETROFIT (tm)[52].*
  - *SUPERSTRUCTURE (tm) [58].*
  - *RECORDER (tm) [18].*
  - *Cobol Structuring Facility (tm) [49].*
  - *Delta STRUCTURIZER (tm).*

2.3 Refactoring Tools

During the last 18 years the concept of refactoring has surpassed the borders of Object Oriented Programming. Its application have multiplied to the extent that one can find it being used in many areas of software engineering. Given its importance as a tool in the maintenance stage, a big number of automated tools for a wide range of programming languages has been developed. Furthermore, quite a few artifacts produced throughout the software development process have been benefited with refactoring tools.

2.3.1 Integrated Development Environment

Nowadays, most of the Integrated Development Environment, commonly called IDE, possesses automated options for refactoring. These features are available to programmers as menu options. These tools assist developers in the refactoring process and facilitate the use of good programming practices giving them the possibility to write high-quality source code on the spot.

Eclipse, Visual Studio .net , Net.Beans are a good example of this kind of refactoring resource. All of them provide the user with a refactoring engine with which refactorings techniques can be applied (Figure 2.1, Figure 2.2).

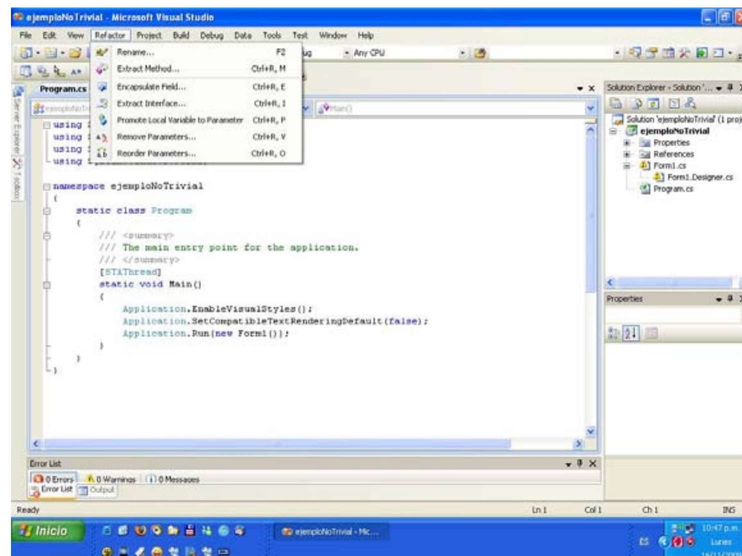

Figure 2.1: Microsoft Visual Studio

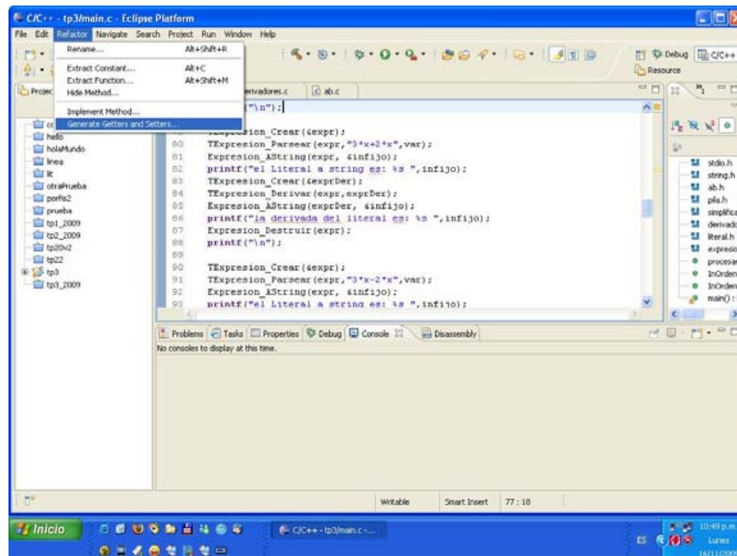

Figure 2.2: Eclipse

With the introduction of this technique in the software development process, a great deal of automated tools to be applied as IDE plug-ins have appeared and revolutionized the programming tasks.

2.3.2 Refactoring Tools

A detailed description of existing refactoring commercial tools for different programming languages will be listed as follows.

Smalltalk

- Smalltalk Refactoring Browser

  Smalltalk Refactoring Browser is probably the most famous refactoring tool. Built by the University of Illinois in the late nineties, Smalltalk Refactoring Browser has been a pioneer tool in this area [43]. It has been developed to be used by VisualWorks, VisualWorks/ENVY, and IBM Smalltalk. Some features of this tool are [71]:

  - Buffers: Users can edit many portions of code at the same time without opening other browsers.

- Drag & drop: The user can drag & drop methods on classes or protocols.
- Hierarchy: Users can easily switch between a hierarchy view and the normal category view without spawning a hierarchy view window, see Figure 2.3, 2.4 .
- Lint: The tool automatically searches among over 60 types of common Smalltalk bugs.
- Old methods: No more accidentally accepting changed methods. Every change the user makes on source code in a window, will be shown in red until the user updates his code or accepts the method.
- Refactorings: Perform some behavior preserving transformations such as abstracting references to an instance variable. The refactorings presented in Bill Opdyke's thesis are implemented.
- Undo support: The Refactoring Browser can undo/redo refactorings, method changes, and class changes.

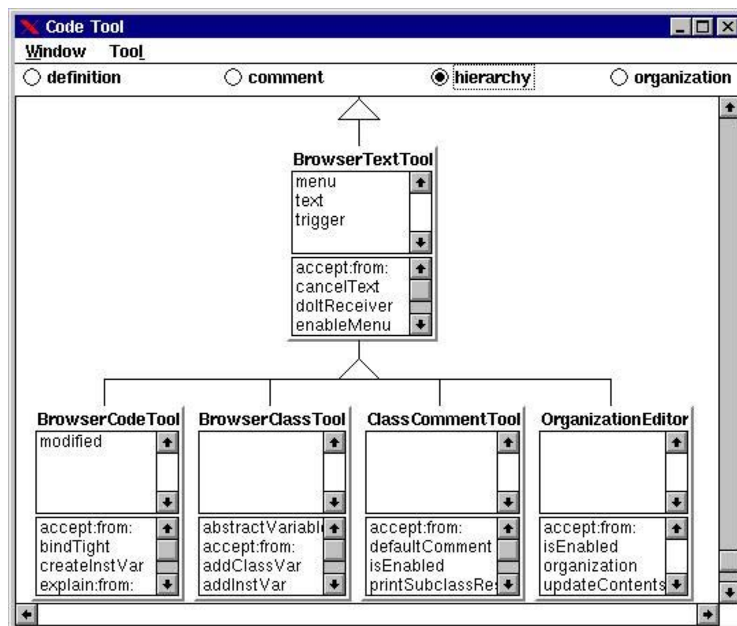

Figure 2.3: Refactoring Browser's hierarchy view.

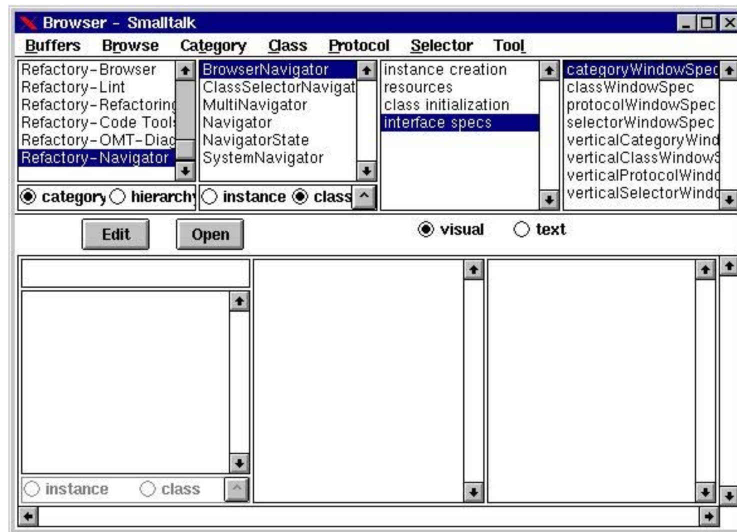

Figure 2.4: Refactoring Browser's normal view

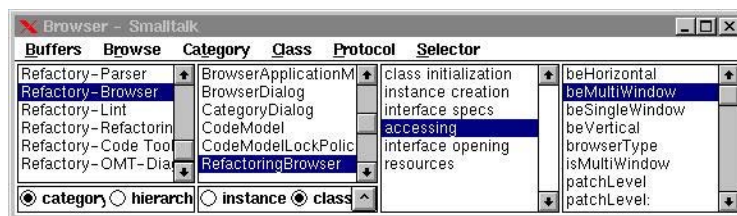

Figure 2.5: The Refactoring Browser's navigator

(http://st-www.cs.uiuc.edu/users/brant/Refactory/RefactoringBrowser.html).

Java

- InteliJ Idea

  This integrated development environment allows the user to apply some refactoring techniques such as: renaming, extracting methods, introducing local variables, and so forth. (http://www.jetbrains.com/idea/index.html)

- JFactor

It can be defined as a product family that provides tools to make use of refactoring techniques. Jfactor can be integrated with the integrated development environment of VisualAge. This product allows the user to utilize the following refactorings: extract method, rename method variables, introduce variable, inline temp, replace magic number with symbolic constant, inline method, rename method, safe delete method, pull up method, push down method, introduce foreign method, rename field, pull up field, push down field, encapsulate field, extract superclass, extract interface.
(http://old.instantiations.com/jfactor/default.htm)

- XRefactory

It is a software development tool especially designed for C and Java to facilitate exploiting refactoring techniques on behalf of programmers. One of its most salient characteristics is the fact that this product can serve as a plug-in for Emacs. Moreover, it supports other programming languages like C (Figure 2.6).

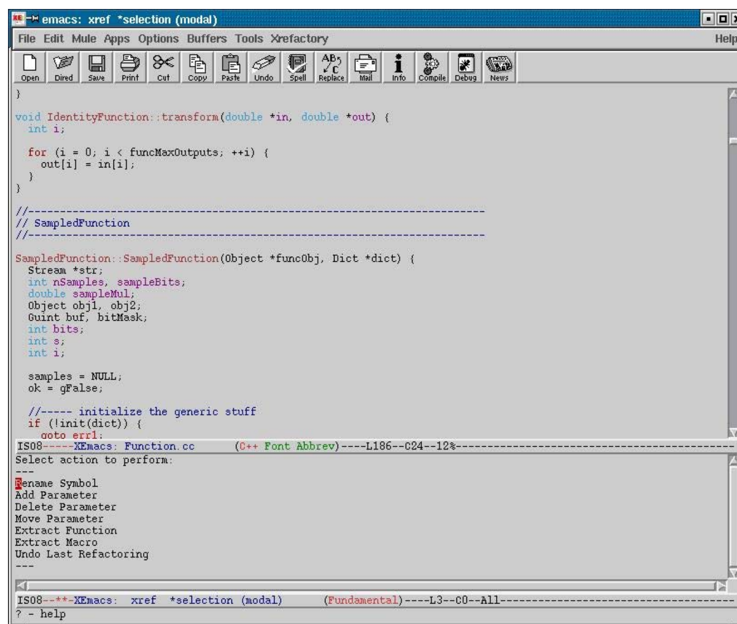

Figure 2.6: XRefactory screenshot

Another remarkable feature promoted by its creators is the one that has been meant to be used in large scale projects, for instance those which have millions of source code lines. This product allows users to apply: method (function) extraction; renaming of namespaces, classes, parameters, variables, fields (structure records) and methods (functions); insertion, deletion and moving of parameters; among others. Refactorings are safe with detection of possible conflicts. Other features are:

  - Detection of unused variables, methods and functions.
  - Functions for finding forgotten symbols.
  - Multiple projects support with project auto-detection.

  (http://www.xref.sk/xrefactory/main.html)

- RefactorIt

  It is a tool especially designed for java developers, that possesses the ability to apply 30 different refactoring techniques to source code. Additionally, it offers a graphic dependencies analyzer and more than 10 metrics for measuring software quality. It can be used as a stand-alone tool or can be integrated as an Eclipse plug-in, netBuilder or Jbuilder.

  (http://freshmeat.net/projects/refactorit/)

- JRefactory

  This tool was developed by Chris Seguin during 1999-2002, it reached 2.6.40 version. Since then Mike Atkinson has taken over, version 2.8 released in October 2003 has many major enhancements. This software is distributed freely and "as is". It is supported by a wide range of IDEs:

  - jEdit (4.1final and 4.2pre11).
  - Netbeans (3.6) - partially supported.
  - JBuilder X - partially supported.
  - Ant (1.5.4) - only pretty printing.

  - It also works as a stand-alone tool.

  For the next version support for other IDEs is expected such as: Eclipse, InteliJ and Emacs JDE.

  Features:

  With JRefactory, users can apply a set of available refactorings such as : Move class between packages (repackage), Rename class, Add abstract parent class, Add child class, Remove empty class, Extract interface, Push up field, Push down field, Rename Field, Push up method, Push up abstract method, Push down method, Move method, Extract method, Rename Parameter.

  Other features are available like UML Diagrams, Pretty Printing, Coding Standard Checking, Bug Finding, AST viewer and Metrics Gatherer. It comes as command-line application with or without GUI and as plug-in for Jedit, Jbuilder, NetBeans and Elixir.

  (http://jrefactory.sourceforge.net/)

- Transmogrify

  Transmogrify creators promote it as a refactoring tool. The user can apply the following:

  - Rename Symbol Extract Method
  - Replace Temp With Query
  - Inline Temp
  - Pull up field

  This product is a plug-in for Jbuilder and Forte4Java. Transmogrify parses all project's files and then it creates a window with the parsed source code,

then users can select the portion of code in which a refactoring will be applied. The tool will analyze a selection and make sure the refactoring chosen is valid. If it is not, an error message will be displayed. Otherwise, it will perform the requested refactoring, making a backup copy of all affected files.

(http://transmogrify.sourceforge.net/)

- JavaRefactor

  Made by Danny Dig in 2002, this tool is a Jedit plug-in. JavaRefactor allows users to apply refactorings base in a small catalog of Java refactorings. Users can rename class, field, method, and package; PushDown and PullUp of methods and fields in an inheritance hierarchy. Other refactorings should be added in a future version.

  The current version of this plugin does not allow source code to be synchronized once refactoring of a particular file has begun. In other words, once you have started refactoring the code in a buffer, future refactorings on that buffer will not take into account any independent changes made by editing until jEdit is restarted.

  (http://plugins.jedit.org/plugins/?JavaRefactor)

.NET

- C# Refactory

  This is a "refactoring, metrics and productivity add-in for Microsoft Visual Studio.NET". It allows users to apply refactoring from a small catalog:

  - Extract method
  - Decompose conditional
  - Extract variable
  - Introduce explaining variable

  - Extract superclass
  - Extract interface
  - Copy class
  - Push up members
  - Rename type
  - Rename member
  - Rename parameter
  - Rename local variable

  (http://www.xtreme-simplicity.net/CSharpRefactory.html)

- Refactor! - See Visual Basic refactoring tools.
- ModelMaker - See Delphi refactoring tools.
- Visual Assist X

  This is a plug-in for Visual Studio. It allows users to apply a reduced set of refactorings:

  - Rename, see Figure 2.7
  - Extract Method
  - Encapsulate Field
  - Change Signature
  - Move Implementation to Source File
  - Add Member
  - Add Similar Member
  - Document Method, see Figure 2.8
  - Create Declaration
  - Create Implementation

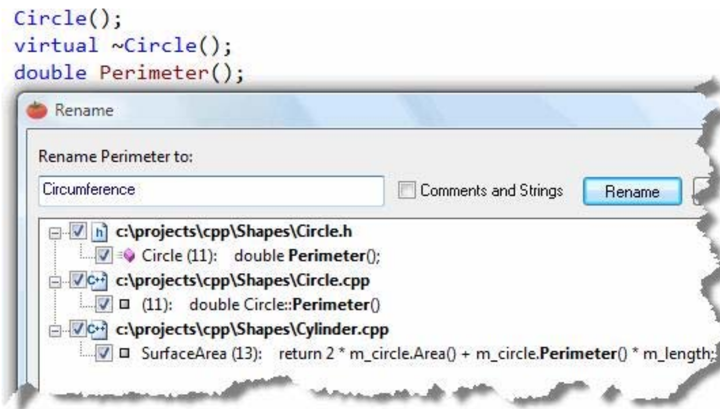

Figure 2.7: Visual Assist rename refactoring

```
//*************************************
// Method:    AmortizeLoan
// FullName:  Money::AmortizeLoan
// Access:    public
// Returns:   double
// Qualifier:
// Parameter: double fPrincipal
// Parameter: double fRate
// Parameter: int nMonths
//*************************************
double AmortizeLoan(double fPrincipal, double fRate, int nMonths);
```

Figure 2.8: Visual Assist document method refactoring

(http://www.wholetomato.com/)

- JustCode!

  Plug-in that provides also some refactoring features available for C#, Visual Basic.net and ASP.net. The user can apply these refactorings:

  – Rename:It allows users to quickly change the name of namespaces, types, methods, fields and practically every type of code written.

  – Organize and Add Missing Usings: It helps users to easily sort, remove unused and add missing using directives.

  – Move Type to Another File: Move a type to a new file having the same name as the type. That refactoring works with classes, enums, interfaces and structures.

- Introduce Field: It allows users to quickly create a new field from a selected constant expression and initialize it with the expression.
- Introduce Variable: It introduces a new variable from an existing one; uses an existing constant to introduce a new variable; uses an existing expression to introduce a new variable.
- Extract Method: It allows users to reorganize code for better reuse and readability by creating a new method based on a selected code fragment. Available for C#, VB.NET and JavaScript.
- Move/delete Parameter :Users can quickly change parameter's position or remove it from method's signature.
- Inline variable: This refactoring replaces all occurrences of the selected variable with its initializer.
- Rename File to Match Type Name: It allows users to quickly rename the opened file to match the selected type name.

(http://www.omnicore.com/en/justcodefeatures.htm)

C/C++

- Ref++
  It is a plug-in for Visual Studio .net that provides some refactoring features for C++.
- XRefactory - See Java.

Visual Basic

- Refactor!

  This tool is a plug-in for Visual Basic .Net that allows users to apply refactorings for this language. This product makes visual basic .net the only source code refactoring. Its creator claims that more than 30 refactorings can be applied. Distributed as a free tool, it has developed the following refactorings (Figure 2.3) :

- Reorder Parameters
- Extract Method, see Figure 2.9
- Extract Property
- Create Overload
- Surrounds With
- Encapsulate field
- Reverse Conditional
- Simplify expression
- Introduce Local
- Introduce constant
- Inline Temp
- Replace Temp with Query
- Split Temporary Variable
- Move initialization to declaration
- Split initialization from declaration
- Move declaration near reference

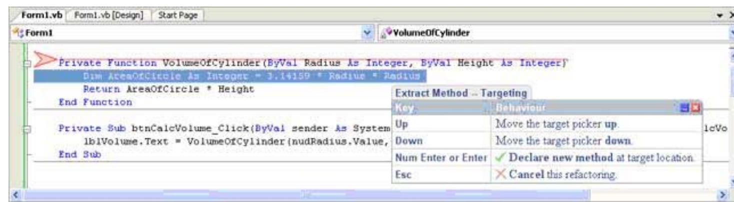

Figure 2.9: Refactor! extract method screenshot

(http://msdn.microsoft.com/es-es/vbasic/ms789083.aspx)

Delphi

- ModelMaker

This tool allows users to apply refactorings in Pascal, Delphi and C# source code. This tool is described as " a two-way class tree oriented productivity, refactoring and UML-style CASE tool". It allows programmers to:

- Copy/Move members to another class.
- Convert local variable or procedure to a field or method.
- Add/Remove a class to/from a module.
- Rearrange classes within modules.
- Create a Delegate from a method.
- Create an Event property or Event handler method from a Delegate.

Erlang

Erlang was designed by Ericsson, is a concurrent oriented programming language. Originally created to build distributed applications, fault tolerant and real time software. It is considered a functional language programming.

- Wrangler

  This is a refactoring tool built for Erlang, that allows users to apply refactoring to Erlang source code interactively. This tool is not for commercial use and it is a prototype.

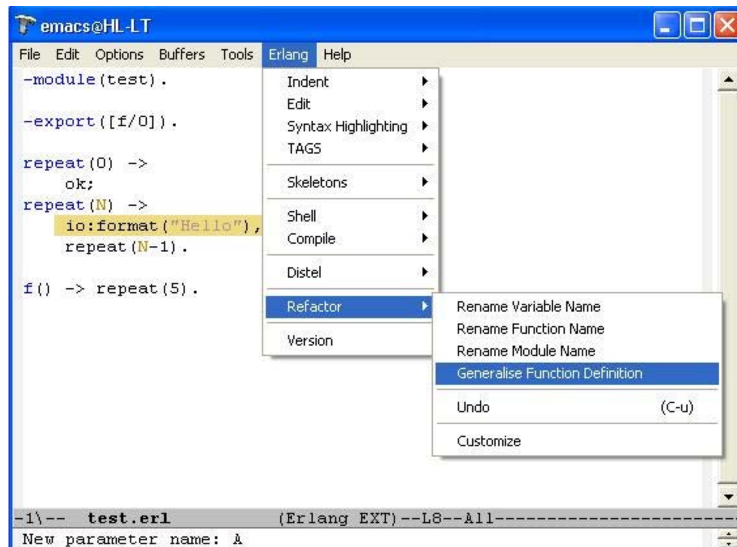

Figure 2.10: Wrangler, an Erlang refactoring tool

(http://www.cs.kent.ac.uk/projects/forse/wrangler/doc/overview-summary.html)

Haskell

- Hare [48]

  This is a prototype tool that allows refactoring Haskell programs. Also known as " The HasKell Refactorer" this tool allows users to apply refactorings, like :

  – Add or remove an argument

  – Algebraic or existential type

  – Concrete to Abstract Data Type

  – Constructor or constructor function

  – Delete/Add a definition

  – Introduce or remove a duplicate definition

  – Simple folding/unfolding

  – Generalise or specialise a definition

- Inside or Outside the ADT
- Layered data types
- Memoisation
- Monadification (variant 1)
- Widen or narrow definition scope
- Widen or narrow definition scope, with compensation (generalise/specialise)
- Renaming
- Add Constructor
- Convert Data Type to Newtype
- Remove Dead Code
- Merge Definitions
- Splitting a definition

Summary

Since Refactoring was born this technique has become recognized as vital in the programming process and an important number of tools has been developed. Nowadays, most of integrated development environments offer users refactorings options. Nevertheless, the refactorings found in these tools are, as a general rule, only the simple ones. As an example, there is no commercial tool capable to handle refactorings with C preprocesor directives.

IDE evolution makes refactoring tools evolve too. The requirements of this kind of tools have evolved. Code preview, refactoring undo, among other are some of the new requirements that a tool must have. Not only is the internal program representation involved but also the user interface is important. New tools may integrate new features like giving advice to programmers of which refactoring should be applied in a specific portion of code, keeping track of code modification, and so forth.

Although refactoring concept has been born in the heart of the object oriented programing, it has crossed over these borders. Refactoring tools can be found in object oriented programing, in structured programing and functional programming.

Making Automated Refactoring Tools is still a big challenge still in these days.

Chapter 3

The Fortran Language

Fortran is the most long lived programming language still at use today. This fact makes it a wonderful case to be studied in detail. This chapter shows Fortran's evolution and the changes in its features with the passing of time.

3.1 A Complex Evolutionary Process

The first publication of the Fortran language came out in November 1954 as a preliminary report "Specification for the IBM 704 Mathematical FORmula TRANslating System". Up to that time, most of the programming work was done in machine language. On November 10th, 1954 automated programming came into existence bringing out a new era in the evolution of programming languages : "The High Level Languages Era".

In 1954, during those days there were a set of factors that made, Programming Research Group (led by John Backus), embark on the development of Fortran. In those days the cost of hiring programmers was at least as great as the cost of computers. Besides most of the computers' time was spent on debugging [8].

Throughout its life Fortran has undergone a lot of changes just like Computer Science. Ever since 1954 we can trace back, at least, ten different versions/revisions of the language. One of most remarkable aspectz of this evolution is that almost every new version (or revision) of the language (except FORTRAN

I,II,II,IV) has maintained a backward compatibility with older versions. Even though some Fortran versions deleted some obsolete features in theory, in practice, compatibility remained: "Unlike Fortran 90, Fortran 95 was not a superset; it deleted a small number of so-called obsolescent features. This incompatibility is more theoretical than real however, as all existing Fortran 95 compilers include the deleted features as extensions" [23]. Thus, this language feature brings about a big number of Fortran programs being still used to this day without upgrading to more modern languages constructions.

3.1.1 FORTRAN I

Fortran language was finally described on October 15th, 1956 in the "IBM Programmer's Reference Manual, the Fortran Automatic Coding System for the IBM 740". This version is also known as FORTRAN I. In the initial release of the language, a set of 32 statements was provided, most of them for Input/Output. All the original statements are listed below [10, 9]:

- Control Statements:
  - PAUSE, STOP, ASSIGN, and CONTINUE statements.
  - GOTO, computed GOTO, assigned GOTO.
  - Arithmetic IF statement.
  - IF statements for checking exceptions such as: ACCUMULATOR OVERFLOW, QUOTIENT OVERFLOW, and DIVIDE CHECK.
  - IF for manipulating IBM 704's sense switches and sense lights: .
  - DO loops.
- Input-Output Statements:
  - Formated I/O : READ, READ INPUT TAPE, WRITE, FORMAT, WRITE OUTPUT TAPE, PRINT and PUNCH.
  - Unformated I/O: WRITE TAPE, READ TAPE, READ DRUM, WRITE DRUM, END FILE, REWIND, and BACKSPACE.

- Assignment statement.
- Specification statements:
    - FREQUENCY statement.
    - EQUIVALENCE statement.
    - DIMENSION statement.

3.1.2 FORTRAN II

After FORTRAN I had been launched, a stage for testing and debugging was needed. In that stage some faults and weaknesses in the system design came to light. In the fall of 1957 FORTRAN I creators began to think that they needed to correct these shortcomings. In September of 1957 this event was documented in an article called "Proposed Specifications for FORTRAN II for the 704". It introduced new features to the language such as [24]:

- User defined functions and subroutines.
- Reference parameter-passing mode.
- CALL and RETURN.
- SUBROUTINE, FUNCTION, and END.
- COMMON.
- New data types: DOUBLE PRECISION and COMPLEX.

```
      FUNCTION DELH1S(P2,D2,T,A )
C     ISOTHERMAL CHANGE IN ENTHALPY FROM ZERO DENSITY TO THE POINT P2,D2
C     ,T USING THE STROBRIDGE EQUATION OF STATE.
C     PRESSURE IN ATM, TEMPERATURE IN DEG K, DENSITY IN MOLES/LITER, AND
C     DELH1S IN JOULES/MOLE.
      DIMENSION A(17)
      P=P2
      D=D2
      EX=EXPF(-A(17)*D*D)
     0DELH1S=(P/D-A(1)*T+D*(A(3)+2.0*A(4)/T+3.0*A(5)/T**2 +5.0*A(6)/T**4
     1    )+A(8)*D*D/2.0 -(3.0*A(10)/T**2 +4.0*A(11)/T**3+5.0*A(12)/
     2    T**4)*EX/(2.0*A(17))-(D**2/(2.0*A(17))+1.0/(2.0*A(17)**2))
     3    *(3.0*A(13)/T**2+4.0*A(14)/T**3+5.0*A(15)/T**4)*EX
     4    +A(16)*D**5/5.0+(3.0*A(10)/T**2+4.0*A(11)/T**3+5.0*A(12)
     5     /T**4)/(2.0*A(17))+(3.*A(13)/T**2+4.0*A(14)/T**3+
     6    5.0*A(15)/T**4)/(2.0*A(17)**2))*101.3278
      RETURN
      END
```

Figure 3.1: IBM 7090 FORTRAN II code example extracted from [40].

In the spring of 1958 FORTRAN II was unveiled. This language was mostly designed by Nelson, Ziller and Backus [8]. One FORTRAN II source code example can be seen inFigure 3.1

3.1.3 FORTRAN III

While FORTRAN II was not still released (1958), people of IBM Programming Research Group were working in a new version of the language. This version included machine-dependent features that permitted users to include assembler code lines within Fortran source code. This intricacy made FORTRAN III code non portable, this may have been the reason that caused FORTRAN III not to be released as a product [8].

3.1.4 FORTRAN IV

One of the most important features of FORTRAN IV programming language was the suppression of machine-dependent characteristics like READ, PRINT, PUNCH, READ INPUT TAPE, and WRITE OUTPUT TAPE since they were replaced by READ and WRITE statements. Other FORTRAN IV characteristics were [1]:

- The introduction of the LOGICAL data type that could only have two values, .TRUE. or .FALSE., introducing the logical expressions evaluated in the if statement. Thus, instead of writing

  IF (W-Z) 10,20,10

  10 CONTINUE

  it was now possible to write

  IF (W .NE. Z) GO TO 20

- Subroutines and functions can be passed as arguments.

- The format's character strings could be enclosed in single quotes, instead of being with the count of characters in them for the former syntax of a Hollerith format specification.

An example of FORTRAN IV source code can be seen as follows, it was extract from [1]:

```
 EXTERNAL FUN
 X=ANUMIN(FUN,0.0 ,1.0)
 PRINT(6 ,11)  X
 STOP
 END
 REAL FUNCTION FUN(X)
 FUN=X+TAN(X)
 RETURN
 END
 REAL FUNCTION ANUMINT(FN,ALOW,AHIGH)
 EXTERNAL FN
 AINC=(AHIGH-ALOW)*0.001
 SUM=0.0005*(FN(ALOW)+FN(AHIGH))
 DO 7  I=1,999
7 SUM=SUM+FN(ALOW+FLOAT(I)*AINC)
 RETURN
 END
```

IBM FORTRAN IV compilers were developed for IBM/360 Mainframe,IBM 7090/7094 and others.

3.1.5 FORTRAN 66

An ANSI committee started developing, in may 1962, a new standard for the FORTRAN language. This committee was composed of industry and academic experts. This important achievement resulted in releasing two new standards. The first one defined FORTRAN, based on FORTRAN IV, became a milestone in the language history and it is also known as FORTRAN 66. A second standard was released too, it was called Basic FORTRAN, based on FORTRAN II where all machinery dependences were left out. The standard published in 1966 was the first High Level Language Standard in the world. FORTRAN 66 included [31, 2]:

- Control Statements:
  - MAIN PROGRAM, SUBROUTINE, FUNCTION, and BLOCK DATA program units.
  - DO loops.
  - Logical IF and arithmetic IF statements.
  - FORMAT statement.
  - CALL, RETURN, PAUSE, and STOP statements.
  - GOTO Statement.
  - Assigned GOTO
  - Computed GOTO statements.
- Input-Output Statements:
  - READ, WRITE, BACKSPACE, REWIND, and ENDFILE.
- Assignment statement.
- Specification statements:

- COMMON statement.
- DIMENSION statement.
- EQUIVALENCE statement.
- DATA statement for specifying initial values.

- Data Types:
  - INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL.

- Other features:
  - Intrinsic and EXTERNAL functions.
  - Assignments.
  - Hollerith constants in DATA statements and FORMAT statements, and acting as actual parameters to procedures.
  - Up to six characters, in length, for naming identifiers.
  - Comments.

3.1.6 FORTRAN 77

In 1969 ANSI saw the need to begin a revision of the FORTRAN 66 standard, Input/Output operations had to be improved and those new features that had been introduced by compilers vendors had to be included in the new standard. The revision caused the language to become the most widely used.

In 1978 the American National Standards Institute published a new standard (ANSI X3.9-1978) historically known as FORTRAN 77. In 1980 International Standard Organization adopted it as an International Standard (IS 1539:1980) which included new features that improved some shortcomings of FORTRAN 66 [29, 2]:

- Control Statements:

  - Block IF and END IF statements, with optional ELSE and ELSE IF clauses, to provide improved language support for structured programming.
  - DO loop extensions, including parameter expressions, negative increments, and zero trip counts.

- Input-Output Statements:
  - OPEN, CLOSE, and INQUIRE statements for improved I/O operation.
  - Direct-access file INPUT/OUTPUT.

- Specification statements:
  - IMPLICIT statement.
  - PARAMETER statement for specifying constants.
  - SAVE statement for persistent local variables.

- Data Types:
  - The required CHARACTER data type was introduced, with a wide range of facilities for character input, output and processing of character-based data.

- Other features:
  - Generic names for intrinsic functions.
  - A set of intrinsics (LGE, LGT, LLE, LLT) for lexical comparison of strings, based upon the ASCII collating sequence.

Since at that time deprecation concept was not allowed in ANSI standards, a set of old features was stripped out from the language standard, see Appendix A2 of the standard. The deleted features were:

- Hollerith constants and data:

  LINE=16HTODAY'S DATE IS:

- Reading into a H edit (Hollerith field) descriptor in a FORMAT specification.
- Over indexing of array bounds by subscripts.

  DIMENSION B(9,5)

  J= B(10,1)
- Transfer of control into the range of a DO loop.

This version of the Fortran standard is one of the most broadly used programming language for developing scientific applications. Historically, it became the most relevant standard of the programming language family.

The US Department of Defense made an extension of the FORTRAN 77 in 1978; this release was called MIL-STD-1753. Some new features were added to this extension such as: DO WHILE and END DO statements, INCLUDE statement, IMPLICIT NONE variant of the IMPLICIT statement and bit manipulation intrinsic functions. All of these features were subsequently included in Fortran 90 standard.

3.1.7 Fortran 90

At this point in the life of FORTRAN language, those decisions previously taken will determine the way in which the language will evolve. First, the language's name changed as can be seen in the standard introduction "Note that the name of this language, Fortran, differs from that in FORTRAN 77 in that only the first letter is capitalized. Both FORTRAN 77 and FORTRAN 66 used only capital letters in the official name of the language, but Fortran 90 does not continue this tradition ..." this minor detail of language specification will point out the impact to some rigid features carried from older versions of the language like fixed-format. Second, a set of new features were introduced as mentioned in the previous section, and due to the fact that Structured Programming was fully developed and Object Oriented Programming Paradigm was gradually growing in popularity, the language needed to be revised. New Fortran standard features were [30]:

- Array operations
- Improved facilities for numerical computation
- Parameterized intrinsic data types
- User-defined data types
- Facilities for modular data and procedure definitions
- Pointers

Most significant changes introduced in Fortran 90 regarding the code formating were:

- Free-form source input.
- Allow lowercase Fortran keywords.
- Long-named identifiers, up to 31 characters in length.
- New comment symbol'!" and the inline comments.

A thorough description of new features presented in Fortran 90 are listed below [30, 2]:

- Control Statements:
  - RECURSIVE procedures.
  - Structured loop constructs, with an END DO statement for loop termination, and EXIT and CYCLE statements for "breaking out" of normal DO loop iterations in an orderly way.
  - SELECT CASE statement construct.
- Data Types:
  - The capability to operate on arrays (or array sections) as a whole.
  - Dynamic memory allocation by means of the ALLOCATABLE attribute and the ALLOCATE and DEALLOCATE statements.
  - POINTER attribute, pointer assignment, and NULLIFY statement to facilitate the creation and manipulation of dynamic data structures.
  - New data type declaration syntax, to specify the data type and other attributes of variables.
- Other features:
  - Modules, to join related procedures and data together, making them available to other program units, including the capability to limit the accessibility to only specific parts of the module by adding the ONLY clause to the USE statement.
  - A widely improved argument-passing mechanism, allowing interfaces to be checked at compile time.
  - User-written interfaces for generic procedures.
  - Operator overloading.
  - Derived/abstract data types.

– Improved intrinsic procedures.

At this point of Fortran evolution the language possessed all of the structured programming language features like C, Pascal, etc. Furthermore, at this point the language was carrying old features since computer did not use bytes as concept (byte first appeared in July 1956, first Fortran's draft was published in 1954). These old-fashioned features were marked as obsolete instead of being deleted. The Appendix B.1 says : "The list of deleted features in this standard is empty.", so a program standard-compliant FORTRAN 77 was also standard-compliant to Fortran 90. The list of obsolete features detailed in Appendix B.1 is [30]:

1. Alternate return.
2. PAUSE statement.
3. ASSIGN statement.
4. Assigned GO TO statements.
5. Arithmetic IF statement.
6. Real and double precision DO control variables and DO loop control expressions.
7. Shared DO termination and termination on a statement other than END DO or CONTINUE use an END DO or a CONTINUE statement for each DO statement.
8. Branching to an END IF statement from outside its IF block branch to the statement following the END IF.
9. Assigned FORMAT specifiers.
10. cH edit descriptor.

3.1.8 Fortran 95

Fortran 95 revision introduced some minor changes to Fortran 90 standard. The most relevant feature that this standard introduced was the deletion of some obsolete language characteristics such as [41]:

- Real and double precision DO variables. The ability present in FORTRAN 77, and for consistency also in Fortran 90, for a DO variable to be of type real or double precision in addition to type integer, has been deleted.
- Branching to an END IF statement from outside its block. In FORTRAN 77, and for consistency also in Fortran 90, it was possible to branch to an END IF statement from outside the IF construct; this has been deleted.
- PAUSE statement. The PAUSE statement, present in FORTRAN 66, FORTRAN 77 and for consistency also in Fortran 90, has been deleted.
- ASSIGN and assigned GO TO statements and assigned format specifiers. The ASSIGN statement and the related assigned GO TO statement, present in FORTRAN 66, FORTRAN 77 and for consistency also in Fortran 90, have been deleted. Further, the ability to use an assigned integer as a format, present in FORTRAN 77 and Fortran 90, has been deleted.
- H edit descriptor. In FORTRAN 77, and for consistency also in Fortran 90, there was an alternative form of character string edit descriptor, which had been the only such form in FORTRAN 66; this has been deleted

Even when obsolete features were deleted, compatibility remained: "Unlike Fortran 90, Fortran 95 was not a superset; it deleted a small number of so-called obsolescent features. This incompatibility is more theoretical than real however, as all existing Fortran 95 compilers include the deleted features as extensions" [23].

Nevertheless, Fortran 95 standard introduced a set of new language features, most of them closely related to High Performance Fortran. Some of these new features are listed below [41]:

- FORALL statement.
- PURE and ELEMENTAL procedures
- Pointer initialization and structure default initialization
- initial association status for pointers
- implicit initialization of derived type objects

Minor features include

- new intrinsic function NULL
- new intrinsic function CPUTIME
- automatic deallocation of allocatable arrays
- SIGN can distinguish between +0 and -0
- comments in namelist input data
- references to pure functions in specification expressions
- changes to some intrinsic functions

3.1.9 Fortran 2003

In 2004 with object oriented programming deeply rooted in the software production, Fortran language needed to be updated to this new programming paradigm. Therefore, the Fortran 95 standard started to be revised, a major revision of it was required. Some of these features are listed below, extract from [42, 69]:

- Derived type enhancements: parameterized derived types, improved control of accessibility, improved structure constructors, and finalizers.
- Object oriented programming support: type extension and inheritance, polymorphisms, dynamic type allocation, and type-bound procedures.

- Data manipulation enhancements: allocatable components, deferred type parameters, VOLATILE attribute, explicit type specification in array constructors and allocate statements, pointer enhancements, extended initialization expressions, and enhanced intrinsic procedures.
- Input/output enhancements: asynchronous transfer, stream access, user specified transfer operations for derived types, user specified control of rounding during format conversions, named constants for preconnected units, the flush statement, regularization of keywords, and access to error messages.
- Procedure pointers.
- Support for the exceptions of the IEEE Floating Point Standard (IEEE 1989).
- Interoperability with the C programming language.
- Support for international usage: access to ISO 10646 4-byte characters and choice of decimal or comma in numeric formatted input/output.
- Enhanced integration with the host operating system: access to command line arguments, environment variables, and processor error messages.

At this point of Fortran evolution an important aspect of the standard implementation is the fact that in these days there is no Fortran compiler fully compliant with the 2003 Standard [22].

3.1.10 Fortran 2008

Fortran 2008 revision is another minor revision of the standard of Fortran language adding clarifications and corrections to Fortran 2003. In August 2010 the final revision was not published yet. Some of the most important features added in Fortran 2008 are [70]:

- CoArrays for parallel computing.

- Submodules provide additional structuring facilities for modules.
- DO CONCURRENT which allows loop iterations to be executed in any order or potentially concurrently.
- Contiguous attribute, for array occupies a contiguous memory block.

Some features initially introduced in the standard were subsequently left out such as BIT data type.

3.1.11 Fortran Evolution

As a successful programming language Fortran is characterized by a long lifetime and by having a huge production of legacy code due to its particular evolutionary process. Such process in which backward version compatibility is maintained and features deletion rarely occurs makes Fortran a very illustrative case to be studied.

Chapter 4

Fortran Refactoring

Even though refactoring concept was born within the pale of object oriented programming we think that this concept is a paramount tool to be applied on Fortran source code. Since it has been successfully used in C language [34], our objective is to build a reference catalog which will serve as a guide to Fortran programmers. In this chapter we will discuss and propose a detailed catalog of Fortran source transformation.

4.1 Different Viewpoints

There exist many different viewpoints when it comes to references about refactorings. One of the most widely used is the paradigm viewpoint. The paradigm viewpoint dichotomizes refactorings such as Object Oriented refactoring, Structured Programming refactoring or Functional Programming Refactoring. Such classification is not good enough if the programming language possesses some features in more than one paradigm. For example the latest version of C# has incorporated lambda notation (an ever-functional-feature); Fortran 2003 has made use of structured and object oriented features.

Another viewpoint to adapt in order to create a good classification may be found in the way that users or programmers need refactorings. This view is based on refactoring intent, that is to say, it depends on refactoring to Improve

Presentation / Readability. In this case, both, Object Oriented and Structured refactorings emerge. This classification is orthogonal with the aforementioned description, only one of them may be used at a time.

4.2 A Classification of Fortran Refactorings

Fortran is one of the most ancient programming language still being used. Fortran programs have a combination of

- Old-style Fortran language constructs, such as those designed in the early stages of the language, up to the '70s.
- Old-style software design methodology or no software development methodology at all. This lack of methodology has been partially mitigated by the strong relationship among scientific programs and mathematical methods implemented.

Fortran evolution has resulted in a wide range of equivalent syntactical constructions. From those equivalent constructions, the older ones (coming from old language version/s) have many disadvantages/drawbacks. Programmers do not need to be aware of all these variations and/or Fortran's dialects in an academic course about Fortran programming, but the scenario radically changes if a programmer is working on a twenty-year-old application that has been written by others in FORTRAN 77 [67, 74].

However, not all Fortran code is legacy code. Fortran has gained a leading role in the High Performance Computing world throughout the years. High Performance Fortran is an extension of Fortran 90 that supports parallel/vector computing [51]. Co-Array Fortran is an extension of Fortran 95 supported by Cray compilers [4]. Currently, old Fortran programs need to be made more efficient in multiprocessing systems with multi-core architectures [75]. Furthermore, multi-core processors are making single-threaded (or, directly, sequential) software obsolete, such as most of the legacy Fortran programs.

Other characteristics of old Fortran programs, such as using COMMON blocks for saving memory, give rise to numerous problems for identifying data as global or local to each subroutine. Automated and graphics tools for Fortran have not been used extensively, and refactoring is a good scenario to introduce and use tools such as Photran in daily software programming/maintenance work.

This section presents a catalog of refactorings for Fortran code. This list of refactorings does not intend to be exhaustive but we aim at providing a complete classification of refactorings according to their specific purpose. Classifying Fortran refactorings by purpose is not easy since a refactoring may belong to more than one category, and we need to decide where it provides the most benefit. However, we think it is worth the effort so developers can make a better decision at selecting the most advantageous refactoring for their needs. We have found two categories of Fortran refactorings: *Refactorings to Improve Maintainability* and *Refactorings to Improve Performance*. Each one of these classes may be divided into subclasses. This categorization is not the only possible one. Many classical refactorings have been intentionally omitted from this list since they are widely described in the literature [59, 32], although they fit into this categorization as well.

4.3 A Catalog of Fortran Refactoring

In the next sections a description of Fortran refactoring is presented as a catalog. We intend to provide an exhaustive definition of Fortran refactorings. Each refactoring is described by its name, intent, motivation, a list of pre or post condition and finally one example applied on source code.

4.4 Refactorings to Improve Maintainability

The refactorings in this category are intended to improve internal quality attributes of the code such as: readability, understandability, flexibility and extensibility (attributes that refactoring has been recognized to improve) and also refactorings

that allow upgrading the code to newer versions of Fortran, removing obsolete features.

4.4.1 Refactorings to Improve Presentation / Readability

Rename

Intent

Change the name of a variable, subprogram, etc.

Motivation

The name of the variable does not communicate its intentions.

Pre-Conditions

The new variable name must not be in conflict with other variable names in the scope.

Source Example

Code Before

```
print   *,GetSqr(n)
end Program

! Returns
the square of n
integer function GetSqr(x)
      integer::x
      GetSqr= x*x
end function GetSqr
```

Code After

```
print *,GetSquare(n)
end Program

! Returns
the square of n
integer function GetSquare(x)
      integer::x
      GetSquare=x*x
end function GetSquare
```

Extract Local Variable

Intent

Remove a subexpression from a larger expression and assign it to a local variable.

Motivation

An expression has grown in its size becoming too difficult to handle or too complex to understand.

Pre-Conditions

The new variable name must not be in conflict with the new name.

Source Example

Code Before

```
  program main
   if(ind.eq.2)then
        nlh=nl/2+1
        global_umax=0.
        do 50 k=nlh,nl
        do 50 lgns=1,lat2
        do 50 mg=1,lon
        global_umax=max(global_umax, _
          sqrt(ureal(mg,lgns,k)**2+ _
           vreal(mg,lgns,k)**2) )
   50   continue
   end if


end program
```

Code After

```
program main
    real :: velocity
    if(ind.eq.2)then
         nlh=nl/2+1
         global_umax=0.
         do 50 k=nlh,nl
         do 50 lgns=1,lat2
         do 50 mg=1,lon
         velocity = vreal(mg,lgns,k)
         global_umax=max(global_umax, _
           sqrt(ureal(mg,lgns,k)**2-
           +velocity**2) )
    50   continue
    end if
end program
```

Extract Local Procedure

Intent

Remove a sequence of statements from a procedure, place them into a new subroutine, and replace the original statements with a call to that subroutine.

Motivation

A portion of source code has become complicated, it has grown too long or simply warrants separation.

Pre-Conditions

New function name must not be in conflict with other function's names in the scope.

Source Example

Code Before

```
program main
   implicit none

   integer :: i,j,k
   integer :: w,z


   print *,'************'
   print *,'*  header  *'
   print *,'************'

   z=(2*k+3*i-5*j)
   print *,z









end program
```

Code After

```
program main
   implicit none

   integer :: i,j,k
   integer :: w,z
   call Print_Header()

   z=(2*k+3*i-5*j)
   print *,z

contains

   subroutine Print_Header()
      implicit none


      print *,'************'
      print *,'*  header  *'
      print *,'************'
   end subroutine
end program
```

Canonicalize Keyword Capitalization

Intent

Make all applicable keywords the same case throughout the selected Fortran program files.

Motivation

Most of Fortran code has been written through different versions of Fortran standards. In that way we can find Fortran code having been written in different capitalization with source code written heterogeneously.

Source Example

Code Before

```
program main

    integer :: i
    real :: j

    do i = 1,10
        j= i / 2
        print *,j
    end do

end program main
```

Code After

```
PROGRAM main

    INTEGER :: i
    REAL :: j

    DO i = 1,10
        j= i / 2
        PRINT *,j
    END DO

END PROGRAM main
```

Standarize Statements

Intent

Rewrite all variables declarations, so that there is only one variable declaration per line, and every variable declaration contains a double colon (::). This is intended to make the code more readable.

Motivation

Throughout time variable declarations have undergone different changes as Fortran standards came out, in that way we can find different manners to declare a variable in Fortran, for example *"integer i"* , *"integer::i"* or simply *"i"*.

Source Example

Code Before

```
program main
    implicit none

    integer :: i,j,k
    integer :: w,z




    call Print_Header()

    z=(2*k+3*i-5*j)
    print *,z
contains

    subroutine Print_Header()
        implicit none


        print *,'************'
        print *,'*  header  *'
        print *,'************'
    end subroutine

end program
```

Code After

```
program main
    implicit none

    integer:: i
    integer::j
    integer::k
    integer:: w
    integer::z
    call Print_Header()

    z=(2*k+3*i-5*j)
    print *,z
contains

    subroutine Print_Header()
        implicit none


        print *,'*************'
        print *,'*  header   *'
        print *,'*************'
    end subroutine

end program
```

4.4.2 Refactorings to Facilitate Design/Interface Change

Encapsulate Variable

Intent

Create getter and setter methods for the selected variable.

Motivation

One of the object oriented principles was not applied, encapsulation must be introduced.

Pre-Conditions

Setter and getter functions must not exist in the scope.

Source Example

Code Before

```
module module1

    integer,public :: temp
    integer :: i
    real ::j

end module module1
```

Code After

```
module module1
    integer :: temp
    private :: temp
    integer :: i
    real ::j
contains
    subroutine setTemp(value)
        implicit none
        integer, intent(in) :: value
        temp = value
    end subroutine

    integer function getTemp()
        implicit none
        getTemp = temp
    end function


end module module1
```

Make Private Entity Public

Intent

Switch a module variable or subprogram from Private to Public visibility.

Motivation

Sometimes the need of redistributing source code from one module to another makes necessary to turn into public some private variables for a certain time.

Source Example

Code Before

```
module module2
   implicit none
! integer1 and integer3
!cannot be made public w/o
!ONLY clause
! integer2 and integer4
!can be made public
   integer, private :: integer1
   integer, private :: integer2
   integer :: integer3, integer4
   private :: integer4, integer3
end module
```

Code After

```
module module2
   implicit none
! integer1 and integer3
!cannot be made public w/o
! ONLY clause
! integer2 and integer4
! can be made public
   integer, private :: integer2
   integer, public :: integer1
   integer :: integer3, integer4
   private :: integer4, integer3
end module
```

Change Subprogram Signature

Intent

Allow the user to add, remove, reorder, rename, or change the types of parameters of a function or subroutine, updating call sites accordingly.

Motivation

Sometimes parameters arrangements are not clear or may change for a given reason for example in order to be compliant with a certain standard. In these cases to reorder parameters becomes a need.

Source Example

Code Before

```
program basictest
    call simple(4,3,2)

    call simple(4,Gamma=2,Beta=3)
end program basictest

subroutine simple(Alpha, Beta, Gamma)
    integer, intent(in) :: Alpha
    integer, intent(out) :: Beta
    integer, intent(inout) :: Gamma
end subroutine
```

Code After

```
program basictest
    call simple(2,3,4)

    call simple(Gamma=2,Beta=3,Alpha=4)
end program basictest

subroutine simple( Gamma, Beta,Alpha)
    integer, intent(in) :: Alpha
    integer, intent(out) :: Beta
    integer, intent(inout) :: Gamma
end subroutine
```

Add Use of Named Entities To Module

Intent

It will allow a programmer to select entities in a module and add a USE ONLY statement in a target module (or alter the existing one).

Motivation

Modules can be used to pass data among program entities. This is done by declaring the commonly used data in the specification section of the module.

Source Example

Code Before

```
module mod1
end module

module mod2
    integer :: x, y
    integer :: z,w
end module

module mod3
    use mod1
    use mod2, only : x,y
end module

program myprogyo
    use mod3

    print *, x
end program
```

Code After

```
module mod1
end module

module mod2
    integer :: x, y
    integer :: z,w
end module

module mod3
    use mod1
    use mod2, only : x,y,w, z
end module

program myprogyo
    use mod3

    print *, x
end program
```

Add Only Clause To Use Statements

Intent

Create a list of the symbols that are being used from a module, and adds it to the Use statement.

Motivation

To increase readability, comprehensibility and maintainability to source code. To reduce coupling among modules.

Source Example

Code Before

```
module module4
    implicit none
        integer f

    contains
    subroutine help_common4
        common /mem/ a, b, c
        integer :: a, b, c
    end subroutine help_common4
end module module4

program test8
    use module4
    implicit none

    call help_common4
end program test8

subroutine asubroutine
    implicit none
    real blah
end subroutine
```

Code After

```
module module4
    implicit none
        integer f

    contains
    subroutine help_common4
        common /mem/ a, b, c
        integer :: a, b, c
    end subroutine help_common4
end module module4

program test8
    use module4, only: help_common4
    implicit none

    call help_common4
end program test8

subroutine asubroutine
    implicit none
    real blah
end subroutine
```

Move Entity Between Modules

Intent

Move a module variable or procedure from one module to another and adjust Use statements accordingly.

Motivation

A variable or a subroutine is declared in a module not accord with its intentions or its functionality.

Pre-Conditions

The variable or the subroutine must not be in conflict with those declared in the new module.

Source Example

Code Before

```
module module1
   integer :: a,b
   integer,parameter :: TWO=2
end module

module module2

   integer :: q=z,b=a
   integer :: z
   integer :: a


contains
```

Code After

```
module module1
   use module2,only :z
   integer :: q=z
   integer :: a,b
   integer,parameter :: TWO=2
end module

module module2

   integer :: b=a
   integer :: z
   integer :: a


contains
```

Safe-Delete Internal Subprograms

Intent

Removes from source code those subprograms no longer used.

Motivation

To reduce the source code complexity by removing all those subprograms never used.

Pre-Conditions

the subprogram must not have references to it.

Source Example

Code Before

```
program test
    y = 3
    j = 4

    stop
    contains

    subroutine dummy
    integer :: j
    end subroutine

end program
```

Code After

```
program test
    y = 3
    j = 4

    stop

end program
```

Change Subroutine to Function

Intent

To convert a subroutine into a function

Motivation

A subprogram conceived as a subroutine has been incorrectly designed, in its place a function is needed.

Source Example

Code Before

```
program test1

  implicit none
  integer :: i,j,sum,difference
  i = 4
  j=2
  call sum_diff(i,j,sum,diff)
  print*,"sum:",sum
  print*,"diff:",diff
end program test1

subroutine sum_diff(i,j,sum,diff)
 integer, intent(in)
:: i,j
 integer, intent(out) :: sum,diff
 sum = i+j
 diff= i-j
end subroutine sum_diff
```

Code After

```
program test1
  implicit none
  integer :: i,j,sum,difference
  i = 4
  j=2
sum = sum_diff(i,j,diff)
  print*,"sum:",sum
  print*,"diff:",diff
end program test1
function sum_diff(i,j,diff)result (sum)
  integer,intent(in)::i,j
  integer,intent(out)::diff
  integer :: sum
  sum = i+j
  diff= i-j
end function sum_diff
```

4.4.3 Refactorings to Avoid Poor Fortran Coding Practices

Remove Unreferenced Labels

Intent

Delete a label if it is never referenced.

Motivation

Old Fortran code uses labels very often. Labels make source code difficult to read ,as a consequence, to make a more readable code, unreferenced labels must be left out from the code.

Source Example

Code Before

```
program main
  integer :: i
      i=1
 100  if (i.lt.10) then
         i=1
 101     continue
 110  else
      end if
end program

900 subroutine OneSubroutine
           return
      end subroutine

integer function OneFunc()
994 OneFunc=1
996 return
end
```

Code After

```
 program main
  integer :: i
      i=1
      if (i.lt.10) then
         i=1

     else
      end if
end program

subroutine OneSubroutine
    return
end subroutine

integer function OneFunc()
OneFunc=1
return
end
```

Remove Real Type Iteration Index

Intent

Change non-integer Do parameters or control variables.

Motivation

This old Fortran feature can cause some unwanted side effects like different numbers of iteration each time the loop is executed, this kind of iteration index must be removed from source code.

Pre-Conditions

The new iteration index must not be in conflict with other variables in the scope.

Source Example

Code Before

```
do x = 1.0,2.0,1.0
    print *, INT(x)
end do
print *, x
```

Code After

```
integer::x
do x = 1,2,1
    print *, INT(x)
end do
print *, x
```

Remove Reserved Words As Variables

Intent

Rename variables named equal to Fortran reserved keywords.

Motivation

Fortran standards allow users to use a keyword name as a variable name. It can cause some unwanted side effects and difficulty in understanding the code.

Pre-Conditions

New variable names must no be in conflict with those that are defined in this scope.

Source Example

Code Before

```
   ip2=ip1
      twopi=-twopic
      if=1
      if(is.eq.2)twopi=twopic
200   if(ip2.ge.ip4)go to 480
      if=if+1
      ifcur=ifact(if)
      if(ifcur.ne.2)go to 120
      if(4*ip2.gt.ip4)go to 120
      if(ifact(if+1).ne.2)go to 120
      if=if+1
      ifcur=4
120   ip3=ip2*ifcur
      theta=twopi/float(ifcur)
```

Code After

```
      ip2=ip1
      twopi=-twopic
      new_name=1
      if(is.eq.2)twopi=twopic
200   if(ip2.ge.ip4)go to 480
      new_name=new_name+1
      ifcur=ifact(new_name)
      if(ifcur.ne.2)go to 120
      if(4*ip2.gt.ip4)go to 120
      if(ifact(new_name+1).ne.2)go to 120
      new_name=new_name+1
      ifcur=4
120   ip3=ip2*ifcur
      theta=twopi/float(ifcur)
```

Introduce Implicit None

Intent

Add Implicit None statements to a file and add explicit declarations for all variables that were previously declared implicitly.

Motivation

Fortran standards allow implicit variable declaration, this practice is not recommendable because it can cause undesired errors or side effects.

Source Example

See next page

Code Before

```
program main
    a=1
    b=2
    i=3
    j=4

contains
subroutine s
    implicit integer (a-c)
    ,complex(h) , real(w)

    c=1
    h=(4,5)
    w=3.0
end subroutine

end program
```

Code After

```
program main
    implicit none
    real :: a
    real :: b
    integer :: i
    integer :: j
    a=1
    b=2
    i=3
    j=4

contains
subroutine s
    implicit none
    integer :: c
    complex :: h
    real :: w

    c=1
    h=(4,5)
    w=3.0
end subroutine

end program
```

Introduce Intent In/Out

Intent

Introduce intent In or Out in each variable declaration within functions and subroutines.

Motivation

To get a clear understanding about a subroutine and/or a function is important to know which is the intent of parameters, if they are used as input parameter or as output parameter.

Source Example

Code Before

```
function Area_Circle(r)
implicit none
real :: Area_Circle
real :: r

! Declare local constant Pi
real:: Pi
parameter :: Pi = 3.14

Area_Circle = Pi * r * r


end function Area_Circle
```

Code After

```
function Area_Circle(r)
implicit none
real :: Area_Circle
real, intent(in) :: r

! Declare local constant Pi
real :: Pi
parameter :: Pi = 3.14

Area_Circle = Pi * r * r


end function Area_Circle
```

Remove Unused Local Variables

Intent

Remove declarations of local variables that are never used.

Motivation

Declared but unused variables may increase the code unreadability. In this case, it is advisable to remove them from source code.

Source Example

Code Before

```
program main
    implicit none

    integer :: i
    integer :: j
    integer :: k
    integer :: w
    integer :: y
    integer :: z

    k=(2*k+3*i-5*j)
    print *,z

end program
```

Code After

```
program main
    implicit none

    integer :: i
    integer :: j
    integer :: k
    integer :: z

    k=(2*k+3*i-5*j)
    print *,z

end program
```

Minimize Only List

Intent

Delete symbols that are not being used from the Only list in a Use statement.

Motivation

To reduce complexity, increase understandability and reduce intra-modular coupling.

Source Example

Code Before

```
module1
        integer :: i
contains
    subroutine helper()
        implicit none
        print(*,'blah')
    end subroutine
end module1

program test
    use module1, only : i,helper
    implicit none
    call helper
end program test
```

Code After

```
module1
        integer :: i
contains
    subroutine helper()
        implicit none
        print(*,'blah')
    end subroutine
end module1

program test
    use module1, only : helper
    implicit none
    call helper
end program test
```

Make Common Variable Names Consistent

Intent

Give variables the same names in all definitions of the Common block.

Motivation

Common blocks allow users to employ different variable names within common blocks. This phenomenon makes source code difficult to understand, read and maintain.

Pre-Conditions

New variables name must not be in conflict with other names in the scope.

Source Example

Code Before

```
   program main
    implicit none

    common /block/ a, b, c, &
         /mem/ r, f, t
    integer :: a
    real :: b
    double precision :: c
    integer :: r, f, t

    a = 5
    b = 4.6
    c = 2.345
    call helper

end program common1

subroutine helper
    implicit none
    common /block/ e, f, g
    integer :: e
    real :: f
    double precision :: g
end subroutine helper

end program
```

Code After

```
program main
    implicit none

    common /block/ a_common, b_common, c_common, &
         /mem/ r, f, t
    integer :: a_common
    real :: b_common
    double precision :: c_common
    integer :: r, f, t

    a_common = 5
    b_common = 4.6
    c_common = 2.345
    call helper

end program common1

subroutine helper
    implicit none
    common /block/ a_common, b_common, c_common
    integer :: a_common
    real :: b_common
    double precision :: c_common
end subroutine helper

end program
```

Add Identifier to END statement

Intent

add the identifier that belongs to the End statements (End Function , End Subroutine).

Motivation

Nested statements may cause code to grow in complexity. To avoid this situation, each statement allowing end statement must be identified. In order to do this, Fortran permits to add the belonging identifier such “END FUNCTION identifier-name” at the end of some statements.

Source Example

Code Before

```
module testmodule
    integer :: xfromtestmodule
10 end

function testfunction(A)
    integer, intent(in) :: A
    testfunction = 4;
20 end ! A comment after

program fortrantest

    print *, "Main_program!"

contains
integer function testfunction(A)
    integer, intent(in) :: A
    testfunction = 4;
30 end function

subroutine do_stuff
   print *,"Hi!"
40 end subroutine
end

subroutine do_stuff
    print *,"Hi!"
50 end
```

Code After

```
module testmodule
    integer :: xfromtestmodule
10 end module testmodule

function testfunction(A)
    integer, intent(in) :: A
    testfunction = 4;
20 end function testfunction ! A comment after

program fortrantest

    print *, "Main_program!"

contains
integer function testfunction(A)
    integer, intent(in) :: A
    testfunction = 4;
30 end function testfunction

subroutine do_stuff
    print *,"Hi!"
40  end subroutine do_stuff
end program fortrantest

subroutine do_stuff
    print *,"Hi!"
50 end subroutine do_stuff
```

Delete Unused Common Block Variable

Intent

Remove unused variables declared in a Common Block.

Motivation

The use of common blocks increases the program complexity, it is convenient to delete those common block variables not used by any statement in the program. Variables declared in common blocks are easily forgotten.

Source Example

See next page.

Code Before

```
   program main
    implicit none

    common /block/ a, b, c, _
    /mem/ r, f, t
    integer :: a
    real :: b
    double precision :: c
    integer :: r, f, t

    a = 5
    c = 2.345
    call helper

end program common1

subroutine helper
    implicit none
    common /block/ e, f, g
    integer :: e
    real :: f
    double precision :: g

    e=6
    g=1.25
end subroutine helper

end program
```

Code After

```
   program main
    implicit none

    common /block/ a, c,
    /mem/ r, f, t
    integer :: a
    double precision :: c
    integer :: r, f, t



    a = 5
    c = 2.345
    call helper

end program common1

subroutine helper
    implicit none
    common /block/ e, g
    integer :: e
    double precision :: g

    e=6
    g=1.25
end subroutine helper

end program
```

Add Dimension Statement

Intent

Add the Dimension statement to declare an array.

Motivation

Old Fortran arrays declaration may be done without dimension clause. To upgrade code with a more updated standard feature, a dimension clause should be introduced in the source code.

Source Example

Code Before

```
    real A(10,20), x(50)
.
```

Code After

```
    real A, x
    dimension x(50)
    dimension A(10,20)
```

Remove Format Statement Labels

Intent

Replace the format code in the read/write statement directly, instead of specifying the format code in a separate format statement.

Motivation

Format statement has been used along different versions of Fortran to allow formatted Input/Output. There is a more structured construction to reach the same objective.

Pre-Conditions

New format parameters must not be in conflict with others in the same scope. No duplicate labels should be in the code.

Source Example

Code Before

```
program test1
    implicit none

    integer :: X
    integer :: Y

    read(1,100,REC=13,ERR=30) X, Y
100 format (I10,F10.3)



end program test1
```

Code After

```
program test1
    implicit none
    character(LEN=9), parameter ::FMT100="I10,F10.3"
    integer :: X
    integer :: Y

    read (1,FMT100,REC=13,ERR=30) X, Y




end program test1
```

4.4.4 Refactorings to Remove Outdated and Obsolete Constructs

Replace Obsolete Operators

Intent

Replace all uses of old-style comparison operators (such as .LT. and .EQ.) with their newer equivalents (symbols such as ¡ and ==).

Motivation

Old style operators are in the appendix B since Fortran 90 standard. A practice of good Fortran programming is to remove such old-fashion operators.

Source Example

Code Before

```
program main
 implicit none

 integer :: i,j,k

 i = 1
 j = 2
 k = 3

if( i.lt.j .and. k.ne.1 .or. k.gt.k) then
   print *,":-)"
end if

end program main
```

Code After

```
program main
     implicit none

     integer :: i,j,k

     i = 1
     j = 2
     k = 3

     if ( i<j .and. k/=1 .or. k>k) then
         print *,":-)"
     end if

end program main
```

Change Fixed Form To Free Form

Intent

Change Fortran fixed format files to Fortran free format files.

Motivation

Since Fortran 90 Standard the programming language allows free-form. The fixed-form sometimes turns the source code unreadable, incomprehensible and hard to maintain.

Source Example

Code Before

```
|---5-7--1----------2-------
      program main

C siple Do Loop
      do 110 i = 1,10
      110 j=i
C siple Do Loop2
      do 120 i = 1,10
      USAV(I,K)=UCLIN(I,K)+
     &VSAV(I,K)-VCLIN(I,K)
      UCLIN(I,K)=UP(I,K)
      VCLIN(I,K)=VP(I,K)
  120 continue
      i=1
      if (i.lt.10) then
      i=1
      else
      i=1+1
      end if


      end program main
```

Code After

```
program main

! siple Do Loop
do 110 i = 1,10
    110 j=i
! siple Do Loop2
do 120 i = 1,10
    USAV(I,K)=UCLIN(I,K)+VSAV(I,K)-VCLIN(I,K)

    UCLIN(I,K)=UP(I,K)
    VCLIN(I,K)=VP(I,K)
120 continue
    i=1
    if (i.lt.10) then
        i=1
    else
        i=1+1
    end if


end program main
```

Transform Character* to Character(Len =) Declaration

Intent

Replace Character*Len with the equivalent Character(Len =) for string declaration.

Motivation

This kind of string declaration is in the appendix B since Fortran 90 Standard, it was replaced by Character(len=) declaration.

Source Example

Code Before

```
program main

implicit none

character*12 cint(12,12)
character*1 cnmlp(lon,lat,2),rainp(lon,lat,2)

character*10 UnOld,lolo* 5 = 'helios'
character*3 CONST,GREEK
character   CATLOG*10,NAME*20
character *10 uno,dos
character hname*20
Character str*10
character hname*20, name*50, lname*50,
 _expdesc*50, hist*65
!     no change
character (len=10)s , str2*36
character (len=10)s1 , str1='lola'
character(len=10) :: UnNewString10

integer i

end program main
```

Code After

```
program main

    implicit none

    character(len=12)::cint(12,12)
    character(len=1)::cnmlp(lon,lat,2)
    character(len=1)::rainp(lon,lat,2)

    character(len=10)::UnOld
    character(len=5)::lolo='helios'
    character(len=3)::CONST
    character(len=3)::GREEK
    character(len=10)::CATLOG
    character(len=20)::NAME
    character(len=10)::uno
    character(len=10)::dos
    character(len=20)::hname
    character(len=10)::str
    character(len=20)::hname
    character(len=50)::name
    character(len=50)::lname
    character(len=50)::expdesc
    character(len=65)::hist
!     no change
    character (len=10)s , str2*36
    character (len=10)s1 , str1='lola'
    character(len=10) :: UnNewString10
    integer i

end program main
```

Remove Computed Go To Statement

Intent

Replace a computed Go To statement with an equivalent Select-Case construct containing Go To or if possible remove the Go Tos statement entirely.

Motivation

This is one of the most ancient Fortran feature released in 1956, this construction must be removed from Fortran source code, this is a not structured construction. It allows spaghetti code production.

Source Example

Code Before

```
go to (12,24,36), index
```

Code After

```
select case (index)
  case ( 1 )
      go to  12
  case ( 2)
      go to 24
  case ( 3 )
      goto 36
end select
```

Remove Arithmetic If Statement

Intent

Replace an old arithmetic If statement, being analogous to removing computed Go To.

Motivation

This is one of the most ancient Fortran feature released in 1956, this construction must be removed from Fortran source code, this is a not structured construction. It allows spaghetti code production.

Source Example

Code Before

```
 program iftest
    integer :: x = -2

    if (x) 10,20,30

10    print *,"x is negative!"
    goto 40
20    print *,"x is zero!"
    goto 40
30    print *,"x is positive!"

40    print *,"end transmission."

end program iftest
```

Code After

```
 program iftest
    integer :: x = -2

    if(x< 0) then
        goto 10
    else if(x == 0) then
        goto 20
    else
        goto 30
    end if

10    print *,"x is negative!"
    goto 40
20    print *,"x is zero!"
    goto 40
30    print *,"x is positive!"

40    print *,"end transmission."

end program iftest
```

Remove Assigned Go Tos

Intent

Remove assigned Go To statements.

Motivation

This is another ancient Fortran feature released in 1956, this construction must be removed from Fortran source code, this is a not structured construction. It allows spaghetti code production.

Source Example

Code Before	Code After
100 . . .	100 . . .
assign 100 **TO** H	. . .
. . .	. . .
GO TO H . . .	**GO TO** 100 . . .

Replace Old Styles DO loops

Intent

Replace old styles Do Loop Continue with the equivalent Do Loop with End Do statement.

Motivation

A DO loop can currently be terminated on a CONTINUE statement, this causes all sorts of confusion, loops must be written in an actualized way.

Source Example

Code Before

```
program main

  ! siple Do Loop
  do 110 i = 1,10
110 j=i

! siple Do Loop2
    do 120 i = 1,10
      USAV(I,K)=UCLIN(I,K)
      VSAV(I,K)=VCLIN(I,K)
      UCLIN(I,K)=UP(I,K)
      VCLIN(I,K)=VP(I,K)
120 continue


end program main
```

Code After

```
  program main

! siple Do Loop
do i = 1,10
    110 j=i
END DO

! siple Do Loop2
do i = 1,10
    USAV(I,K)=UCLIN(I,K)
    VSAV(I,K)=VCLIN(I,K)
    UCLIN(I,K)=UP(I,K)
    VCLIN(I,K)=VP(I,K)
    120 continue
END DO

end program main
```

Replace Shared Do Loop Termination

Intent

Replace all shared Do Loop termination construct with the equivalent Do Loop with End Do statement.

Motivation

A number of DO loops can currently be terminated on the same (possibly executable) statement, this causes all sorts of confusion, loops must be written in an actualized manner.

Source Example

Code Before

```
   program main

    ! Shared Do Loop Termination
    do 100 j=1,10
    do 100 w=1,10
100 i=j+1


end program main
```

Code After

```
  program main

    ! Shared Do Loop Termination
    do j=1,10
        do w=1,10
            100 i=j+1
        end do
    end do

end program main
```

Transform To While Sentence

Intent

Remove simulated While made by If and Go To statement.

Motivation

There is a WHILE statement simulated with a non structured construction, to avoid the use of not structured construction it must be replaced with a WHILE statement.

Source Example

Code Before

```
      . . .
   integer :: n
   n = 1
   10 if (n .lt. 100) then
        n = 2*n
        write (*,*) n
        goto 10
     end if
   . . .
```

Code After

```
. . .
. . .
. . .
. . .
n = 1
do while(n .lt. 100 )
   n=2 * n
   write (*,*)n
end do
```

Move Common Block to Module

Intent

Remove all declarations of a particular Common block, moving its variable declarations into a module and introducing Use statements as necessary.

Motivation

The use of common blocks make the source code hard to understand and read since common blocks can have different names among modules.

Source Example

See next page.

Code Before

```
module module1
    implicit none

    common /block/ a, b
    integer :: a,b
contains

integer function add()
    implicit none
    common /block/ e, f
    integer :: e,f
    add=e+f
end function add

integer function mult()
    implicit none
    common /block/ e, f
    integer :: e,f
    mult=e*f
end function mult
end module module1
```

Code After

```
module module1
    type mytype
        integer :: a
        integer :: b
    end type
contains

integer function add(a)
    type(mytype):: a
    add=a%e+a%f

end function add

integer function mult(a)
    type(mytype):: a
    mult=a%e+a%f

end function mult


end module module1
```

Move Saved Variables To Common Block

Intent

Create a Common block for all saved variables of a subprogram. Declarations of these variables in the subprogram are transformed such that they are no longer "saved". The generated common block is declared both in the main PROGRAM and in the affected subprogram[63].

Motivation

To eliminate the static behavior from certain variables.

Source Example

See new page.

Code Before

```
PROGRAM MyMain
    USE MySeparateFileMod
    COMMON /MyTestFun_common1/ aVar
    REAL :: a_xxx1
    REAL :: comVar
    REAL :: aVar
    COMMON /CB1/ comVar
    print *, test
    print *, internalModVar
    comVar = 5.5
    CONTAINS

    REAL FUNCTION MyTestFun()
        REAL :: com
        COMMON /MyTestFun_common2/ com
        REAL :: q = 3.3, w, e = 5.5
        REAL, DIMENSION(5) :: r, t
        REAL, SAVE :: u = 1.1
        REAL, SAVE :: o
        REAL, POINTER :: p
        POINTER o
        REAL :: b, c, d
        REAL, POINTER :: a
        POINTER c, d
        SAVE a, r, p, b, c
        DIMENSION b(10)
        c = 1.2
        MyTestFun = 3.3
    END FUNCTION MyTestFun

    REAL FUNCTION MyTestFun2(aVar)
        REAL, DIMENSION (10:10) :: aVar
        CHARACTER (LEN=30) :: char
        REAL :: bVar(100:100)
        DOUBLE PRECISION :: cVar(10)
        REAL, PARAMETER :: b = 1.1
        REAL c
        POINTER c
        SAVE
        MyTestFun2 = 0.0
    END FUNCTION MyTestFun2
END PROGRAM MyMain
SUBROUTINE MySub
    REAL :: test
    COMMON /CB1/ comVar
    test = 1.1
    comVar = comVar + comVar
END SUBROUTINE MySub
```

Code After

```
PROGRAM MyMain
    USE MySeparateFileMod
    REAL, POINTER :: a_xxx2
    REAL, DIMENSION(10) :: b_xxx1
    REAL, POINTER :: c_xxx1
    REAL :: e_xxx1 = 5.5
    REAL, POINTER :: o_xxx1, p_xxx1
    REAL :: q_xxx1 = 3.3
    REAL, DIMENSION(5) :: r_xxx1
    REAL :: u_xxx1 = 1.1
    COMMON /MyTestFun_common3/ a_xxx2, b_xxx1, c_xxx1 _
, e_xxx1, o_xxx1, p_xxx1, q_xxx1, r_xxx1, u_xxx1
    COMMON /MyTestFun_common1/ aVar
    REAL :: a_xxx1
    REAL :: comVar
    REAL :: aVar
    COMMON /CB1/ comVar
    print *, test
    print *, internalModVar
    comVar = 5.5
    CONTAINS
    REAL FUNCTION MyTestFun()
        COMMON /MyTestFun_common3/ a_xxx2, b_xxx1, c_xxx1 _
, e_xxx1 , o_xxx1, p_xxx1, q_xxx1, r_xxx1, u_xxx1
        REAL :: com
        COMMON /MyTestFun_common2/ com
        REAL :: q_xxx1, w, e_xxx1
        REAL, DIMENSION(5) :: r_xxx1, t
        REAL :: u_xxx1, o_xxx1
        REAL, POINTER :: p_xxx1
        POINTER o_xxx1
        REAL :: b_xxx1, c_xxx1, d
        REAL, POINTER :: a_xxx2
        POINTER c_xxx1, d
        DIMENSION b_xxx1(10)
        c_xxx1 = 1.2
        MyTestFun = 3.3
    END FUNCTION MyTestFun

    REAL FUNCTION MyTestFun2(aVar)
        REAL, DIMENSION (10:10) :: aVar
        CHARACTER (LEN=30) :: char
        REAL :: bVar(100:100),c
        DOUBLE PRECISION :: cVar(10)
        REAL, PARAMETER :: b = 1.1
        POINTER c
        SAVE
        MyTestFun2 = 0.0
    END FUNCTION MyTestFun2
END PROGRAM MyMain
SUBROUTINE MySub
    REAL :: test
    COMMON /CB1/ comVar
    test = 1.1
    comVar = comVar + comVar
END SUBROUTINE MySub
```

Data To Parameter

Intent

Change a Data declaration to Parameter declaration making more clear which variables are constant and which ones are not.

Motivation

Since the use of DATA statement is chained to variables assignments, sometimes there are variables used as constants. In a way to improve the memory allocation and memory access these identifiers should be declared as constants.

Source Example

Code Before

```
program dataToParameter

    implicit none

    real :: x, y, z
    integer :: a, b, c !A comment

    !Those values are assigned
    data x,y,z/1.,2.,3./

    !About to assign more values

    data a/10/,b/15/,c/20/

    x = 5.4
    b = 6

end program dataToParameter
```

Code After

```
program dataToParameter
    implicit none
    real :: x, y, z
    integer :: a, b, c !A comment

    data x/1./ !Those values are assigned
    parameter ( z = 3. ) !Those values are assi
    parameter ( y = 2. ) !Those values are assi

    data b/15/

    parameter ( c = 20 )
    parameter ( a = 10 )
    !About to change some assigned values
    x = 5.4
    b = 6

end program dataToParameter
```

4.5 Performance Refactorings

This category currently has some examples of how refactoring can be used to improve performance while preserving not only the behavior of the program but also the readability and maintainability of the code. This is one of the factors that sets refactoring apart from optimization.

4.5.1 Refactorings For Performance

Change To Vector Form

Intent

rewrite a Do Loop into an equivalent Fortran vectorial notation, which allows the compiler to make better optimizations [75].

Motivation

To eliminate do loop from source code and to utilize vector notation allowed in Fortran. Sometimes it can cause a performance improvement because of compiler optimization.

Source Example

Code Before	Code After
`do 10 i = 1,100`	`. . .`
`x(1) = x(1) + y(1)`	`x(1:100)= x(1:100)+(1:100)`
`10 continue`	`. . .`

Interchange Loops

Intent

swap inner and outer loops of the selected nested do-loop, in the case that doing so allows to optimize memory access pattern and allows to take advantage of data prefetching techniques.

Motivation

When the loop variables index into an array, such a transformation can improve locality of reference, depending on the array's layout.

Source Example

```
Code Before
------------------------------

program main

 integer :: max
  parameter (max = 10)

  integer::clouds(max,max)
  integer::ocean(max,max)
  integer :: i,j

  do i = 1,10
   do j= 1,10
       clouds(i,j) = ocean(i+2,j-1)
   end do
  end do

end program main
```

```
Code After
------------------------------

program main

 integer :: max
 parameter (max = 10)

 integer::clouds(max,max)
 integer::ocean(max,max)
 integer :: i,j

 do j= 1,10
     do i = 1,10
       clouds(i,j) = ocean(i+2,j-1)
     end do
 end do

end program main
```

Loop Reversal

Intent

Take an incrementing or decrementing loop, swap the lower and upper bounds, and negate the step.

Motivation

This optimization may help to eliminate dependencies enabling other kinds of optimizations. Together with the fact that certain architectures use looping constructs at Assembly language level that count in a single direction only (e.g. decrement-jump-if-not-zero (DJNZ)).

Source Example

Code Before

```
program test
   implicit none
   integer :: i

   do i = 1 ,10 ,2
        print *,i
   end do
 end program test
```

Code After

```
program test
   implicit none
   integer :: i

   do i = 10,1,-2
        print *,i
   end do

 end program test
```

Loop Unrolling

Intent

Take the selected do-loop and either completely or partially unroll it. This will also optionally include a conditional statement to make sure the loop stays in bounds.

Motivation

Duplicate the body of the loop multiple times, in order to decrease the number of times the loop condition is tested and the number of jumps, which may degrade performance by impairing the instruction pipeline.

Source Example

Code Before

```
program test
  implicit none
  integer :: i

  do i = 1 ,10 ,2
     print *,i
  end do

end program test
```

Code After

```
program test
  implicit none
  integer :: i

  do i = 1 ,10 ,8
     print *,i
     print *,i+2
     if (i+4>10) then exit
     print *,i+4
     print *,i+6
  end do
end program test
```

Loop Tiling

Intent

This refactoring takes a double nested do-loop, and creates a nested do-loop with four levels of depth. Instead of iterating through a two dimensional array (for example) by going through each row, it will loop over smaller tile blocks. [63]

Motivation

Loop tiling reorganizes a loop to iterate over blocks of data size, it can produce a gaining on performance.

Source Example

```
Code Before                          Code After
------------------------------       ------------------------------

  implicit none                      program test

    integer :: i                     implicit none
    integer :: j                     integer :: i1, j1
    integer :: n=1, m=20, p=10
                                     integer :: i
                                     integer :: j
    do i=n,10                        integer :: n=1, m=20, p=10
        do j=n,m                     do i1=floor(real(n-20)/3)*3+20,8,3
            print *, i                 do j1=floor(real(n-20)/3)*3+20,floor(real(m-20)/3)*3+20,3
        end do                             do i=max(n,i1),min(10,i1+2)
    end do                                     do j=max(n,j1),min(m,j1+2)
                                                   print *, i
                                               end do
                                           end do
                                       end do
                                     end do

end program test                     end program test
```

Loop Fusion

Intent

Take two do-loops, normalize their bounds, and finally put the loop bodies in a single do-loop.

Motivation

Two or more adjacent loops would iterate the same number of times (whether or not that number is known at compile time), their bodies can be joined as long as they make no reference to each other's data.

Source Example

Code Before

```
program test

   implicit none
   integer :: i,j

   do i = 1 ,10,2
        print *,i
   end do

   do j = 21,25,1
        print *,j
   end do
end program test
```

Code After

```
program test

   implicit none
   integer :: i,j

   do i = 0 ,4,1
        print *,(i*2+1)
        print *,(i*1+21)
   end do


 end program test
```

4.6 Differences Between Fortran and Other Languages Refactorings

As a long lived language Fortran has compiled a vast amount of intricate language constructions. These constructions have been compiled in a language that evolved throughout 50 years of existence. We consider Fortran the best example of a successful language who resists the push of time. Therefore, the language evolution through years has brought about Fortran specific code transformation to be performed with the intention to make Fortran programs compliant with the standard evolution. Fortran is the first case studies of a set of other programming languages with a long trajectory like COBOL or Lisp.

Refactoring tools are of paramount importance to help programming languages to evolve and to help programmers to keep their programs up-to-date. And in the specific case of Fortran refactoring tools will play a role of evolution facilitator.

As a conclusion, the study of these kinds of transformation open the door to examine the modern programming languages evolution course such as Java, c#, Ruby, etc.

Chapter 5

Photran:A Refactoring Tool for Fortran

Photran is an advanced, multiplatform integrated development environment (IDE) for Fortran based on Eclipse. Photran has a number of powerful features. As an IDE, it integrates editing, source navigation, compilation, and debugging into a single tool. It uses *make* for compilation, which allows it to work with virtually any existing Fortran compiler; so-called *error parsers* are provided which interpret the error messages from popular compilers, associating error markers with the appropriate lines of code. *Language-based searching* allows a Fortran programmer to quickly find a subprogram or module with a particular name, or to find all of the references to a particular variable or subprogram. From the beginning, Photran was designed to support refactoring, and much of its development effort has focused on providing a robust refactoring infrastructure. Version 6.0 (released June, 2010) contains 16 refactorings, and many more are under development. The development version of Photran provides name binding, control flow, and basic data flow information to support precondition checking, see Figure 5.1.

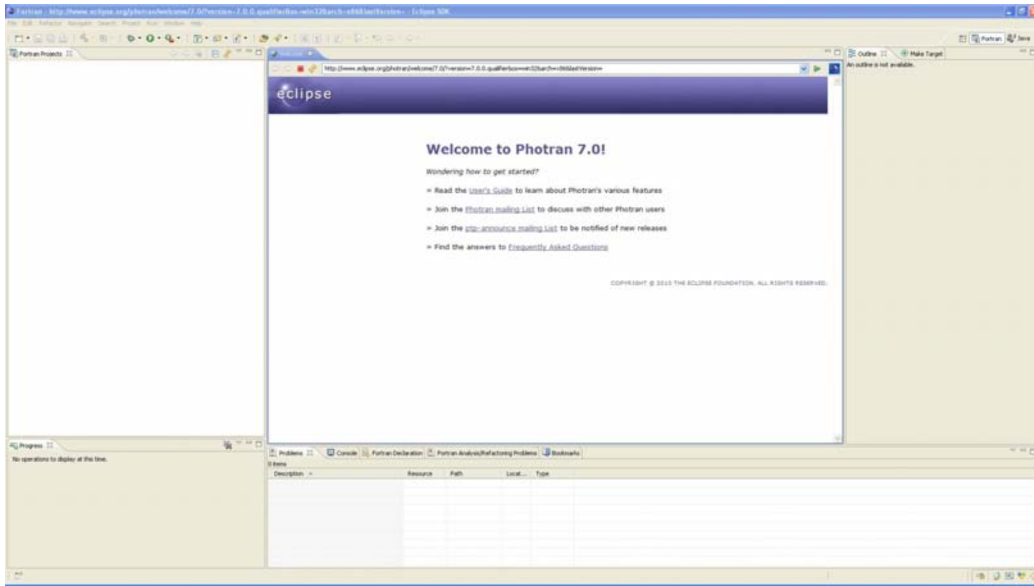

Figure 5.1: Photran, Fortran View

5.1 The Architecture

Photran is based on Eclipse C Development Tool. CDT integrates a set of tools to compile programs. One of these tools is make, which controls the generation of executables, another one is called gdb and it integrates interactive debugging. In 2006 Overbey and Rasmusen provided a patch to CDT when an extension point was added. This extension point allowed make-based programming languages, other than C, to be used at the CDT core. Once a new language has been plugged at this new extension point, the new language is allowed to use CDT capabilities [62].

Photran was born as a research project at the University of Illinois, basically within the Research Group of Dr. Ralph Johnson.

5.2 Photran Core

In the list below we can find a detailed description of the Photran core , see Figure 5.2.

The main packages making up Photran include (description mainly extracted

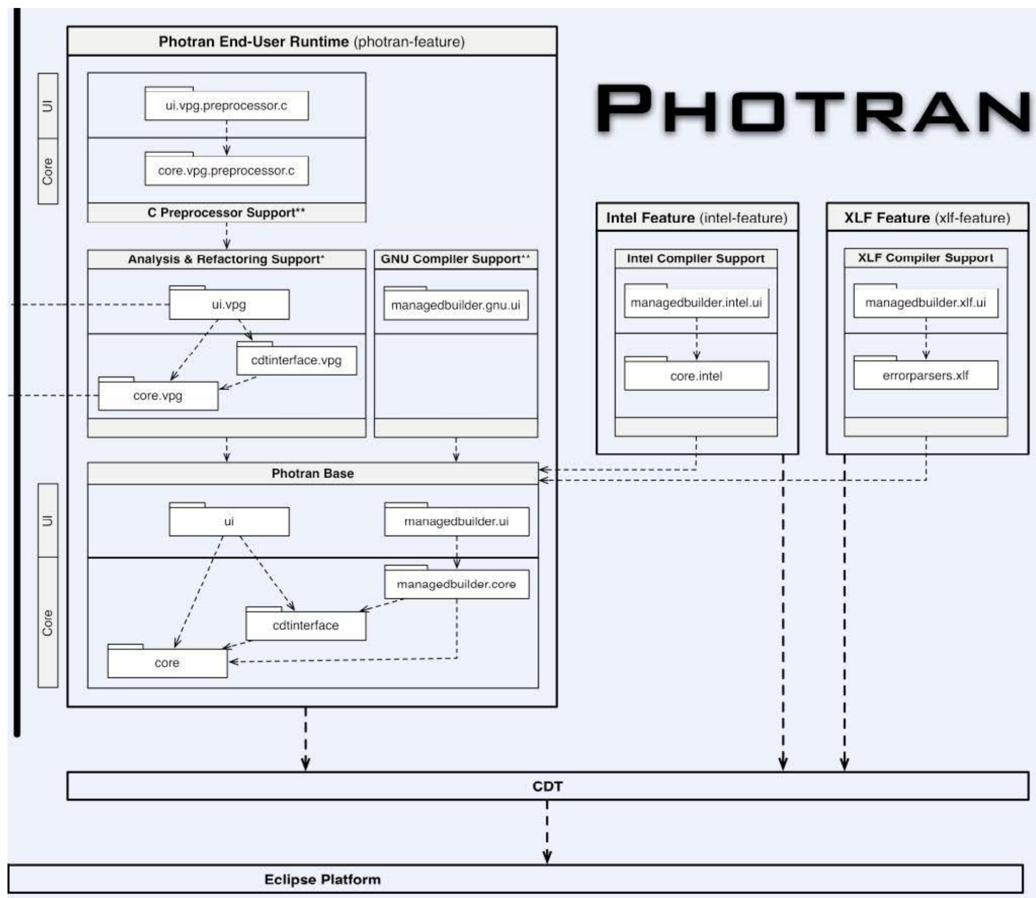

Figure 5.2: Photran Architecture [64]

from[64, 65]):

- `org.eclipse.photran.cdtinterface`
  - The FortranLanguage class, which adds Fortran to the list of languages recognized by CDT
  - Fortran model elements and icons for the Outline and Fortran Projects views
  - An extension point for contributing Fortran model builders
  - The Fortran perspective, Fortran Projects view, and other CDT-based parts of the user interface
  - New Project wizards and Fortran project templates

- `org.eclipse.photran.core`

  As described in Photran Developer Guide, this package contains

  - Workspace preferences for Fortran projects
  - Error parsers for Fortran compilers
  - Utility classes

- `org.eclipse.photran.core.vpg`

  This is probably the most complex package of Photran because the whole refactoring infrastructure lies within. Inside of it the Parser, the VPG (Virtual Program Graph), the AST (Abstract Syntax Tree) nodes and the refactorings can be found.

  - Fortran parser and the AST
  - Fortran preprocessor (to handle INCLUDE lines)
  - Parser-based model builder
  - Photran's VPG
  - Utility classes (e.g., SemanticError, LineCol)
  - Project property pages
  - Name binding analysis (equivalent to symbol tables)
  - Refactoring/program transformation engine
  - Refactorings

- `org.eclipse.photran.ui`

  In this package UI components not provided by CDT infrastructure were built.

  - Fortran Editors
    - Fixed Format Editor
    - Free Format Editor
  - Preference pages

- `org.eclipse.photran.ui.vpg` In this package UI components closely related to the refactoring infrastructure are deployed, such as: specific refactoring UI, input - output dialogs, etc.

- `org.eclipse.photran.core.vpg.preprocessor.c` In this package the required classes for refactoring Fortran with C preprocessed directives are found.

5.3 The Program Representation

Photran contains two very important structures. The first one is the AST, which maintains the entire representation of a Fortran program. The AST structure is filled with AST nodes, a set of classes that represent each possible element of the programming language, Figure 5.3 is an example of it. The second important structure is the VPG which facilitates the handling of the AST and the embedded analysis information, acting as a facade between the AST and the programmer [65]. Thus, VPG allows refactoring programmers to acquire or release ASTs; it also sets off scope and binding analysis; while allowing the user to obtain variable definitions, and so forth.

Figure 5.3: An example of Photran AST

5.4 Refactoring Infrastructure

Photran divides refactorings into two categories: *editor-based refactorings*, which require the user to select part of a Fortran program in a text editor in order to initiate the refactoring, and *resource refactorings* which apply to entire files.

To create a new refactoring, the developer must decide whether it will be an editor-based refactoring or a resource refactoring. Photran provides different superclasses for each. The developer then creates a concrete subclass and adds a line of XML to a configuration file to make Photran aware of the new refactoring. The concrete subclass must define methods which first provide the name of the refactoring. This becomes its label in the Refactor menu. It is also used to describe the refactoring in the Edit > Undo menu item and in other user interface elements.

Second, check initial preconditions. These are usually simple checks which verify that the user selected the correct construct in the editor, that the file is

not read-only, and so foth.

Third, it is necessary to acquire user input. For example, a refactoring to add a parameter to a subprogram must ask the user to supply the new parameter's name and type. Then check final preconditions. These validate user input and perform any additional checks necessary to ensure that the transformation can be performed, the resulting code will compile, and it will retain the behavior of the original program.

And finally, perform the transformation. Once all preconditions have been checked, this method determines what files will be changed, and how. Thanks to the XML configuration file and Java's reflective facilities, much of the user interface for a refactoring comes "for free".

Then Photran automatically adds the refactoring to the appropriate parts of the Eclipse user interface, and it provides a wizard-style dialog box which allows the user to interact with the refactoring. This dialog includes a *diff*-like preview, which allows the user to see what changes the refactoring will make before committing it.

Chapter 6

Refactoring Examples

In this chapter a thorough specification about how to build Fortran refactorings in Photran will be presented. Four Fortran specific refactorings have been selected from the proposed catalog with the intention of describing and implementing them in this thesis.

6.1 Initial Steps

To introduce a new refactoring in the Photran menu, it is necessary to edit the plugin.xml file. As we said in the previous chapter, there are two types of refactorings in Photran. The first type is called **editor refactoring**, this kind of refactoring allows the users to work with a code selection or with the editor selected file (Fortran constructions like loops or a certain set of statements). The second one is called **resource refactoring**, this refactoring type allows the users to work with an entire set of files, named *resources* in Photran (programs or modules).

Photran refactoring engine provides two classes corresponding to the two refactoring types (editor or resource). To make a new refactoring, a class must be created so as to extend the FortranEditorRefactoring or FortranResourceRefactoring (see Figure 6.1).

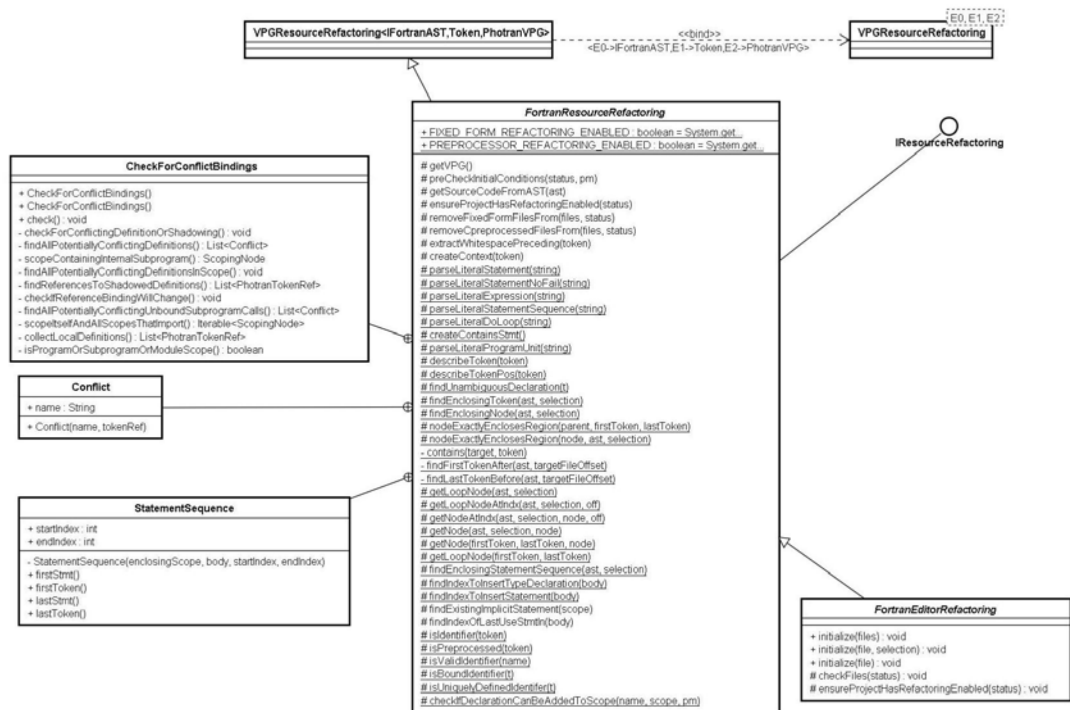

Figure 6.1: Photran refactoring class diagram

Once we have decided which subclass to implement in order to make the refactoring, we need to make Photran aware of the new refactoring by editing the plugin.xml [65].

There are some methods that must be implemented and overridden in order to obtain a refactoring. Each one of these methods have a precise intent within the refactoring structure. Those methods are :

- public String getName()
- protected void doCheckInitialConditions(RefactoringStatus status, IProgressMonitor pm) throws PreconditionFailure
- protected void doCreateChange(IProgressMonitor pm) throws CoreException, OperationCanceledException
- protected void doCheckFinalConditions(RefactoringStatus status, IProgressMonitor pm) throws PreconditionFailure

6.2 Transform Character* to Character(Len =)

6.2.1 Inception

In order to introduce a more updated way to declare string variables, we propose a new refactoring. This refactoring changes the multiple ways to write a string declaration by replacing all declaration types by character(len=). By doing this, all declarations will result in a more updated, readable and understandable code. The following code shows different ways to declare character strings:

```
character*10 NewString
character NewString*10
character NewString(10)
character (len=10) NewString
character (len=10) :: NewString
```

The following string declarations are a real life example:

```
character*1   cnmlp(lon,lat,2),rainp(lon,lat,2)
character*10  UnOld,String* 5 = 'helios'
character*3   const,greek
character     catalog*10,name*20
character *10 name,phone
character hname*20
character str*10
character hname*20, name*50, lname*50,expdesc*50, hist*65
character (len=10)s , str2*36
character (len=10)s1 , str1='lola'
character(len=10) :: aNewString10
```

6.2.2 The Design

At this point, we need to define how the refactoring will work. Basically, this refactoring rewrites all character declarations into CHARACTER(LEN=) form. To perform this change, the refactoring must collect all character declarations within a certain scope in a file. Then it must transform them into a CHARACTER(LEN=) declaration.

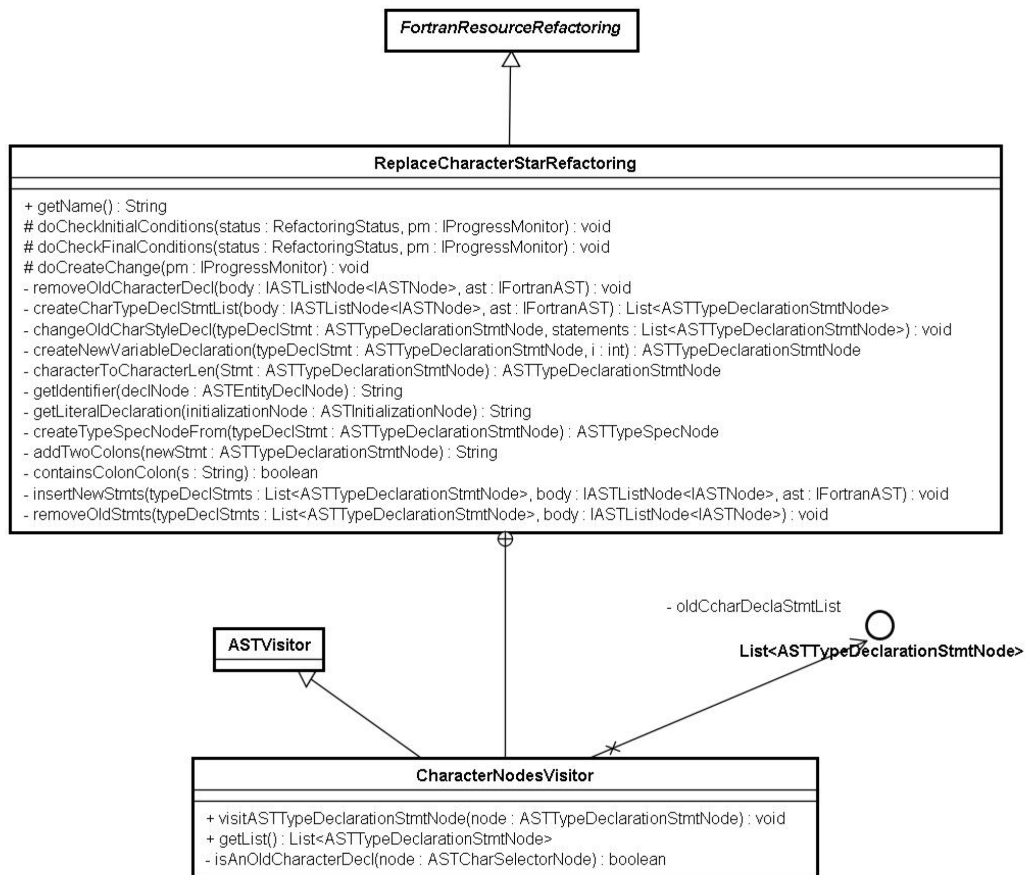

Figure 6.2: Photran replace character star refactoring class diagram

6.2.3 The Implementation

A new subclass of FortranResourceRefactoring was added in the `org.eclipse.photran.internal.core.refactoring`. Moreover, the responsibility to maintain node references is delegated to a visitor pattern [33] which is implemented within the refactoring class as static final class (see Figure 6.2):

The Implementation steps:

1. Edit the plugin.xml:

```
<group><!-- Refactorings that reformat code -->
  <resourseRefactoring
     class="org.eclipse.photran.internal.core.refactoring.ReplaceCharacterStarRefactoring"
  />
  <resourceRefactoring
```

```
    class="org.eclipse.photran.internal.core.refactoring.RepObsOpersRefactoring"
  />
</group>
```

2. The class must expose its name to the eclipse environment, to do this the getname() method is defined. To expose this name, this overriding is needed:

```
@Override
public String getName() {
  return Messages.ReplaceCharacterToCraracterLenRefactoring_Name;
}
```

3. In order to check the initial preconditions required to apply the refactoring transformation, the doCheckInitialConditions method is overridden. Inside this method, the work of checking the initial conditions required by the refactoring will be performed. In this case, only one precondition is required, character * declarations must be present in the source code. The transformation will be made only on those character declarations not complying with the required format.

   The doCheckInitialConditions() method checks the refactoring engine availability by the ensureProjectHasRefactoringEnabled() method. As a second step, the removal of the fixed format files and the C-preprocessed files is performed by the removeFixedFormFilesFrom() and removeCpreprocessed-FilesFrom() methods in order to extract from the selected files those which are not available for refactoring yet, such as fixed format files and preprocessed files. A visitor is used in this stage to collect all characters * declaration and to check if there is a string declaration to be transformed. In the case that no declaration is found, a message will be shown.

```
@Override
  protected void doCheckInitialConditions(RefactoringStatus status,
          IProgressMonitor pm) throws PreconditionFailure {

      ensureProjectHasRefactoringEnabled(status);
      removeFixedFormFilesFrom(this.selectedFiles, status);
      removeCpreprocessedFilesFrom(this.selectedFiles, status);
```

```
        //iterateThroughAllTypeDeclarationStmtNodes
        CharacterNodesVisitor characterVisitor = new CharacterNodesVisitor();
        IFile file = this.fileInEditor;
        IFortranAST ast = vpg.acquirePermanentAST(file);
        ast.accept(characterVisitor);

        // if there is not any character * a message is shown
        if (characterVisitor.getList().size()<1)
            fail(Messages.ReplCharToCharLenRef_CharacterStarDeclNotSelected);
    }
```

Note: Messages.ReplCharToCharLenRef stands for Messages.ReplaceCharacterToCharacterLenRefactoring

The CharacterNodesVisitor is responsible for counting the old string declarations. We traverse the AST structure counting the character * declaration forms. This class is listed below:

```
private static final class CharacterNodesVisitor extends ASTVisitor {
        private List<ASTTypeDeclarationStmtNode> oldCcharDeclaStmtList=
                new LinkedList<ASTTypeDeclarationStmtNode>();

        @Override
        public void visitASTTypeDeclarationStmtNode (ASTTypeDeclarationStmtNode node){
            ASTTypeSpecNode specTypeNode=node.getTypeSpec();

            if (isCharacterDeclaration(ASTTypeSpecNode specTypeNode)){

                ASTCharSelectorNode charSelectorNode = specTypeNode.getCharSelector();
                if (charSelectorNode!=null) {
                   if (isAnOldCharacterDecl(charSelectorNode)){
                       // put the node in the list is a Character *
                       oldCharDeclaStmtList.add(node);
                    }
                }
                else oldCharDeclaStmtList.add(node);
            }
        }

        public List<ASTTypeDeclarationStmtNode> getList() {
            return this.oldcharDeclaStmtList;
        }

        private boolean isAnOldCharacterDecl(ASTCharSelectorNode node) {
            return  ! node.isAssumedLength()
                    && ! node.isColon()
                    && ! (node.getConstIntLength()==null)
                    && (node.getLengthExpr()== null)
                    && (node.getKindExpr()==null)
                    && (node.getKindExpr2()==null);
        }
```

```
        private boolean isCharacterDeclaration(ASTTypeSpecNode specTypeNode) {
            return (specTypeNode!= null) && specTypeNode.isCharacter();
        }

    }
```

4. Since no user input is needed this is the last step to perform the transformation. To refactor the source code we need to iterate the different scopes in the file, by using the method doCreateChange().

```
@SuppressWarnings("unchecked")
@Override
protected void doCreateChange(IProgressMonitor pm) throws CoreException,
     OperationCanceledException {

  IFile file = this.fileInEditor;
  IFortranAST ast = vpg.acquirePermanentAST(file);
  List<ScopingNode> scopes = ast.getRoot().getAllContainedScopes();
  for (ScopingNode scope : scopes)
      if (!(scope instanceof ASTExecutableProgramNode) &&
          !(scope instanceof ASTDerivedTypeDefNode))
          removeOldCharacterDecl((IASTListNode<IASTNode>)scope.getBody(), ast);

  this.addChangeFromModifiedAST(this.fileInEditor, pm);
  vpg.releaseAST(this.fileInEditor);

 }
```

For each scope in the AST, the visitor must gather all the AST nodes representing a character * declaration in order to check whether the node should be rewritten.

```
private void removeOldCharacterDecl(IASTListNode<IASTNode> body , IFortranAST ast) {
  // Removes all character declaration from a scope
  // creating first a list of new Character Declarations

  List<ASTTypeDeclarationStmtNode> typeCharDeclStmts =
      createCharTypeDeclStmtList(body, ast);

  insertNewStmts(typeCharDeclStmts, body, ast);
  removeOldStmts(typeCharDeclStmts, body);

 }

 private List<ASTTypeDeclarationStmtNode> createCharTypeDeclStmtList(IASTListNode<IASTNode> body
{
        List<ASTTypeDeclarationStmtNode> statements = new
            LinkedList<ASTTypeDeclarationStmtNode>();
```

```
        CharacterNodesVisitor charVisitor= new CharacterNodesVisitor();
        ast.accept(charVisitor);
        for ( IASTNode node : body ){
            if (matches(node,charVisitor.getList()))
                changeOldCharStyleDecl((ASTTypeDeclarationStmtNode)node, statements);
        }
        return statements;
    }

    private boolean matches (IASTNode node, List<ASTTypeDeclarationStmtNode> list){
        return ( node instanceof ASTTypeDeclarationStmtNode )
              && (list.contains(node) )
    }
```

5. Helper Methods:

   To perform the Transform Character * to Character (Len) refactoring some helper methods were implemented. This transformation can be divided into two parts: the node collection and the node rewriting. To perform the rewriting stage, some methods whose responsibility consists of rewriting new nodes, have been defined. The method createNewVariableDeclaration() is in charge of rewriting the strings declaration. The code is listed below:

```
@SuppressWarnings("unchecked")
    private ASTTypeDeclarationStmtNode createNewVariableDeclaration
         (ASTTypeDeclarationStmtNode typeDeclStmt, int size) {

        IASTListNode<ASTEntityDeclNode> variables =
            typeDeclStmt.getEntityDeclList();

        ASTTypeDeclarationStmtNode newStmt =
            (ASTTypeDeclarationStmtNode)typeDeclStmt.clone();

        if (size>0) newStmt.setTypeSpec(createTypeSpecNodeFrom(typeDeclStmt));

        IASTListNode<ASTEntityDeclNode> newVariable =
            (IASTListNode<ASTEntityDeclNode>)variables.clone();

        List<ASTEntityDeclNode> listOfVariablesToRemove =
            new LinkedList<ASTEntityDeclNode>();

        for (int j=0; j<variables.size(); j++)
            if (j != i) listOfVariablesToRemove.add(newVariable.get(j));

        newVariable.removeAll(listOfVariablesToRemove);
        newStmt.setEntityDeclList(newVariable);

        // Insert "::" if the original statement does not contain that already

        String source = addTwoColons(newStmt);
        newStmt = (ASTTypeDeclarationStmtNode)parseLiteralStatement(source);
```

```
        // replace old Style Character
        newStmt=characterToCharacterLen(newStmt);
        return newStmt;

    }
```

Another complex helper method to perform this refactoring is the characterToCharacterLen() method, which serves the functions of rewriting the string node declarations.

```
private ASTTypeDeclarationStmtNode characterToCharacterLen(
        ASTTypeDeclarationStmtNode Stmt) {

    String length=""; //$NON-NLS-1$
    String literalIniDec=""; //$NON-NLS-1$
    ASTTypeSpecNode type=Stmt.getTypeSpec();
    ASTEntityDeclNode declNode=Stmt.getEntityDeclList().get(0);

    if (isCharacterStarString(Stmt)){
      if (hasCharacterLength(declNode))
          length=type.getCharSelector().getConstIntLength().getText();
      else {
            // declarations is something like: character *10 First, Second*5 and we
            //are working on ---> Second*5
            length=declNode.getCharLength().getConstIntLength().getText();
            // remove length
            declNode.getCharLength().removeFromTree();
            // remove character selector
            type.getCharSelector().removeFromTree();
      }
    }
    else {
        if (isCharacterStringStar(declNode)) {
            length=declNode.getCharLength().getConstIntLength().getText();
            declNode.getCharLength().removeFromTree();
        }
        else {
            String strType=type.getCharacterToken().getText();
            // is :character*
            if (isCharacterStrr(strType))
                length=strType.substring(strType.indexOf("*")+1);
            else
                length="1";
        }
    }
    String source1= "character(len=" + length + ")" +"::";
    String source2 = getIdentifier(Stmt.getEntityDeclList().get(0));
    String commentsBefore=Stmt.findFirstToken().getWhiteBefore();
    String commentsAfter=Stmt.findLastToken().getWhiteBefore();
```

```
        literalIniDec  =
           getLiteralDeclaration(Stmt.getEntityDeclList().get(0).getInitialization());

        String literalStmt=
                 commentsBefore + source1 + source2 + literalIniDec +commentsAfter;

        Stmt=(ASTTypeDeclarationStmtNode)parseLiteralStatement(literalStmt);

        return Stmt;
    }

    private boolean isCharacterStarString( ASTTypeDeclarationStmtNode Stmt){
           return (Stmt.getTypeSpec().getCharSelector()!= null)
    }

    private boolean hasCharacterLength( ASTEntityDeclNode node ){
      return (node.getCharLength() ==null)
    }

    private boolean isCharacterStringStar(ASTEntityDeclNode node){
     return (node.getCharLength() !=null);
    }

    private boolean isCharacterStar(String){
       return (strType.contains("*"));
    }
```

Some images of the refactoring process can be appreciated at Figures 6.3, 6.4 and 6.5.

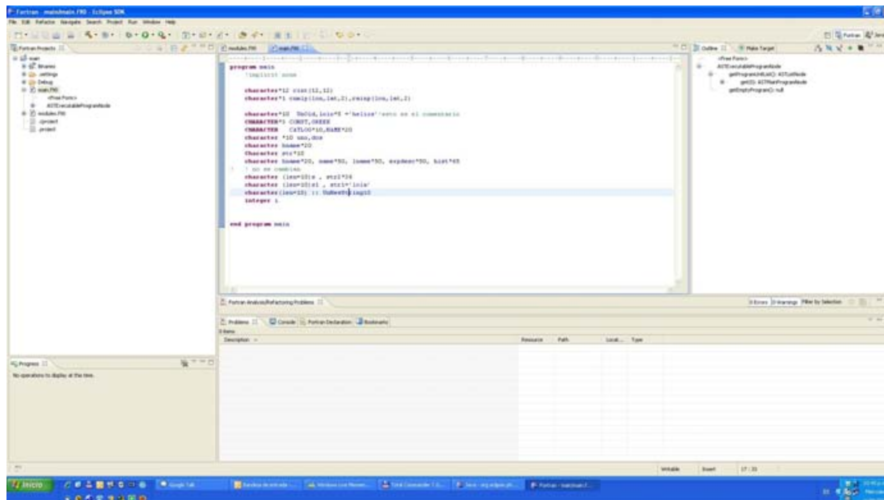

Figure 6.3: Code before applying Transform CHARACTER* to CHARACTER(LEN =) refactoring.

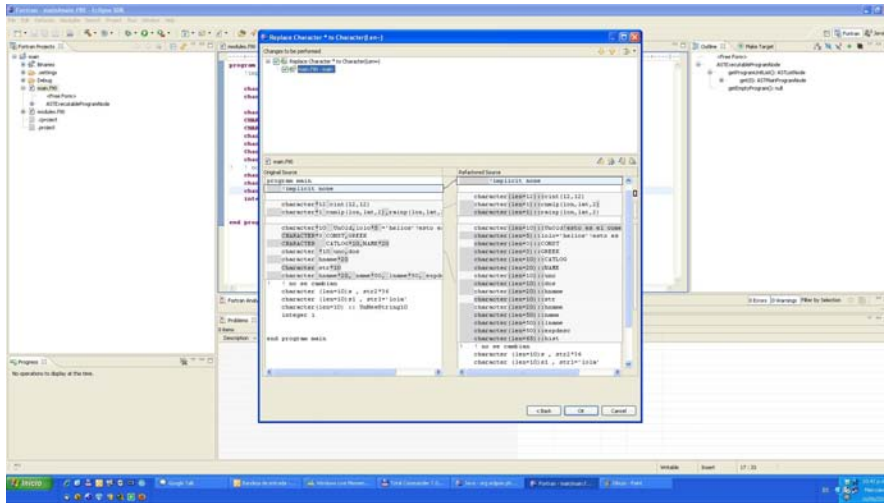

Figure 6.4: Transform CHARACTER* to CHARACTER(LEN =) Diff-view.

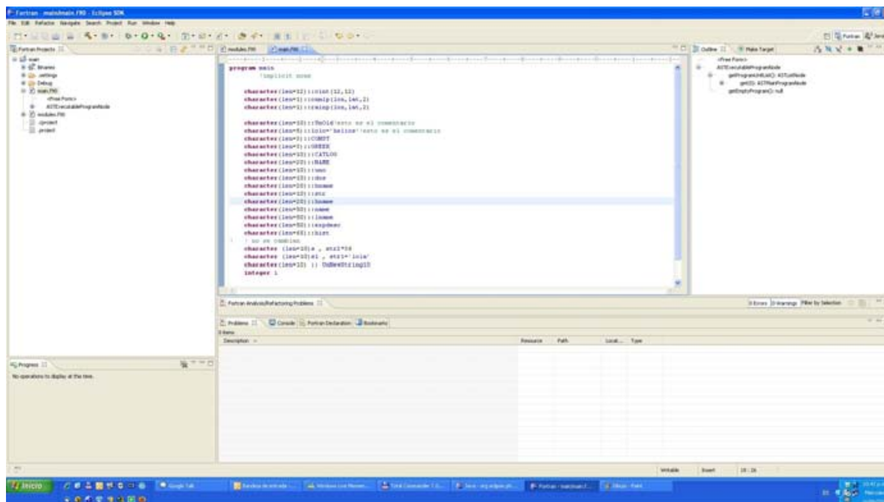

Figure 6.5: Transform CHARACTER* to CHARACTER(LEN =) after the refactoring was applied.

6.3 Standardize Input Output Formats

6.3.1 Inception

The FORMAT statement have gone hand in hand with Fortran ever since language first version was published in 1956 (FORTRAN I). This instruction is used to get format inputs and outputs. As a very old instruction, the syntax of Format statement uses labels. As a consequence of this situation, the Fortran source code that uses input/output operations is plagued with FORMAT statements and its labels. As an example, a real life source code is listed below:

```
      IF(M.EQ.2) PRINT 8002,J,ITT
      IF(M.EQ.1) PRINT 8001,J,ITT
      IF(M.EQ.2) PRINT 8002,J,ITT
      IF(M.EQ.3) PRINT 8003,J,ITT
      IF(M.EQ.4) PRINT 8004,J,ITT
8001 FORMAT(20H TEMPERATURE FOR J =,I4,12H AT TIMESTEP,I7)
8002 FORMAT(20H SALINITY     FOR J =,I4,12H AT TIMESTEP,I7)
8003 FORMAT(20H TRAER 1     FOR J =,I4,12H AT TIMESTEP,I7)
8004 FORMAT(20H TRAER 2     FOR J =,I4,12H AT TIMESTEP,I7)
      PRINT 8011,J,ITT
8011 FORMAT(20H W VELOITY   FOR J =,I4,12H AT TIMESTEP,I7)
      PRINT 8021, J,ITT
8021 FORMAT(20H U VELOITY   FOR J =,I4,12H AT TIMESTEP,I7)
      SL = 1.0
      !ALL MATRIX(U,IMT,ISTRT,ISTOP,0,KM,S!L)
      PRINT 8022, J,ITT
8022 FORMAT(20H V VELOITY   FOR J =,I4,12H AT TIMESTEP,I7)
      PRINT 912,ITT,DATE,MON,EKTOT,MS!AN,
912  FORMAT(' ITT=',I12,' DDMM=',1p,1e10.3,I3,I3,' QN=',0pF6.1)
      IF(MOD(ITT,1460).EQ.0)PRINT 913,EKTOT,DTABS(1),DTABS(2)
913  FORMAT(15X,'EN=',1PE15.8,' DT=',1PE14.7,' DS=',1PE14.7)
      print 917,(k,(zdzz(k)/100),alevel(k),(tlevel(k,m),m=1,4),k=1,km)
917  format(/,'  k   depth area',8x,'temp',11x,'sal',9x,'rms v',7x,'rms w',/)
      do 1918 k = 1,km
      print 918,tmin(k),itmin(k),jtmin(k),k
918  format(' min temperature =',f7.2,'    at point (',3i3,')')
1918 continue
      PRINT 9100
      PRINT 9101,ENGINT(1),ENGEXT(1),TTDTOT(1,1),TTDTOT(1,2)
      PRINT 9102,ENGINT(2),ENGEXT(2),TTDTOT(2,1),TTDTOT(2,2)
      PRINT 9103,ENGINT(3),ENGEXT(3),TTDTOT(3,1),TTDTOT(3,2)
      PRINT 9104,ENGINT(4),ENGEXT(4),TTDTOT(4,1),TTDTOT(4,2)
      PRINT 9105,ENGINT(5),ENGEXT(5),TTDTOT(5,1),TTDTOT(5,2)
      PRINT 9106,ENGINT(6),ENGEXT(6),TTDTOT(6,1),TTDTOT(6,2)
      PRINT 9109,PLI!IN,PLI!EX,TVAR(1),TVAR(2)
      PRINT 9107,ENGINT(7),ENGEXT(7)
      PRINT 9108,ENGINT(8),ENGEXT(8)
```

```
9100 FORMAT( 1X,50HWORK BY: INTERNAL MODE EXTERNAL MODE,10X,50H TEMPERATURE SALINITY)
9101 FORMAT( 1X,20HTIME RATE OF HANGE,2(1PE15.6),10X,20HTIME RATE OF HANGE ,2(1PE15.6))
9102 FORMAT( 1X,20HHORIZONTAL ADVETION,2(1PE15.6)10X,20HHORIZONTAL ADVETION,2(1PE15.6))
9103 FORMAT( 1X,20HVERTIAL ADVETION,2(1PE15.6),10X,20HVERTIAL ADVETION ,2(1PE15.6))
9104 FORMAT( 1X,20HHORIZONTAL FRITION ,2(1PE15.6),10X,20HHORIZONTAL DIFFUSION,2(1PE15.6))
9105 FORMAT( 1X,20HVERTITION,2(1PE15.6),10X,20HSURFAE FLUX,2(1PE15.6))
```

This instruction makes code difficult to follow, read and understand. Of course there is another way to introduce the strings into the code to format the input and output operations. The programmer is only expected to declare a string parameter with the value of the desired format.

6.3.2 The Design

In this refactoring a visitor must traverse the entire AST structure searching for FORMAT, PRINT, WRITE and READ statements. Once all statements were collected for each FORMAT instruction, a string parameter must be declared and assigned with the corresponding format string. Subsequently, each input output statement referring to this FORMAT instruction must be modified to point to the new string parameter.

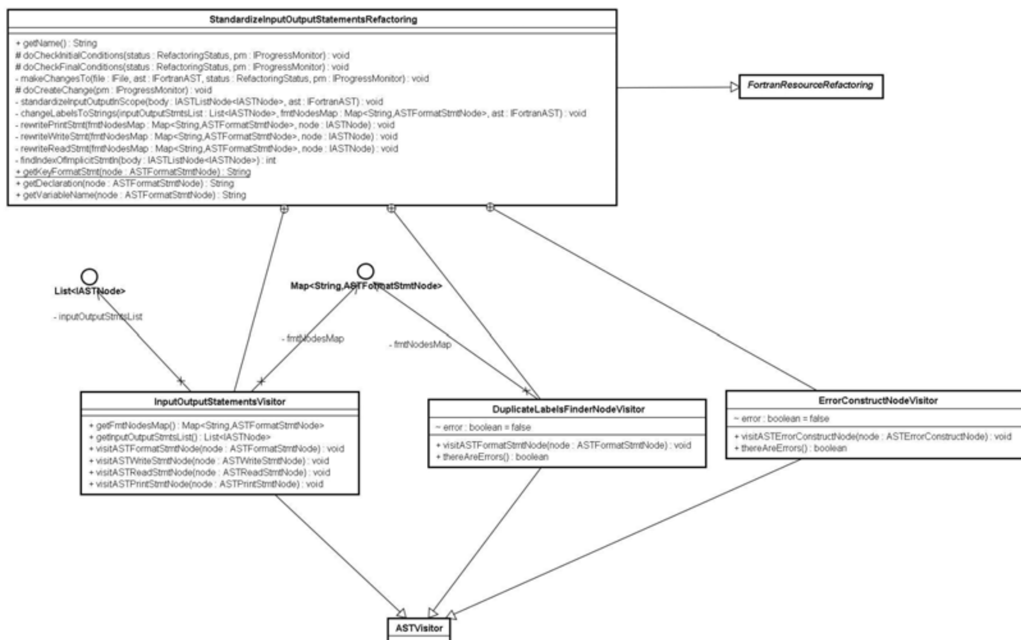

Figure 6.6: Photran Standardize I/O Refactoring Class Diagram

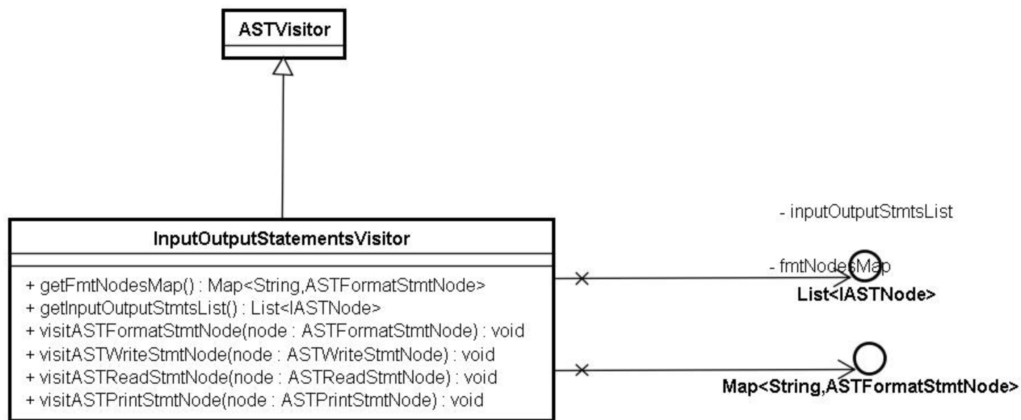

Figure 6.7: InputOutputStatement Visitor Class Diagram

6.3.3 The Implementation

A new subclass of FortranResourseRefactoring was added in the `org.eclipse.photran.internal.core.refactoring`. In this case, the responsibility for maintaining the references lists is delegated to a visitor, because this transformation may be applied to more than one file, so the class can not hold all the references (see Figure 6.6 and Figure 6.7)

1. Edit the plugin.xml :

```
<group><!-- Refactorings that change local variable declarations -->
<resourceRefactoring
 class="org.eclipse.photran.internal.core.refactoring.IntroImplicitNoneRefactoring" />
<resourceRefactoring
 class="org.eclipse.photran.internal.core.refactoring.DataToParameterRefactoring" />
<resourceRefactoring
 class="org.eclipse.photran.internal.core.refactoring.RemoveUnusedVariablesRefactoring"/>
<resourceRefactoring
 class="org.eclipse.photran.internal.core.refactoring.StandardizeStatementsRefactoring" />
</group>
```

   This step was performed to integrate the new refactoring into photran User Interface.

2. Expose the name :

```
@Override
public String getName() {
  return Messages.StandardizeInputOutputFormatsRefactoring_Name;
```

```
}
```

3. Check the initial preconditions:

   In this case two preconditions are checked. First of all, no errors must be found in the source code, in order to achieve this precondition the ErrorConstructNodeVisitor class traverses each AST in the selected files. A further precondition is checked, no labels with duplicate values are allowed to run this refactoring, the DuplicateLabelsFinderNodeVisitor looks for labels defined more that once. It is important to note that the refactoring engine works with source code without being compiled ,moreover , no source code correctness is guaranteed before applying the refactoring.

```
@Override
protected void doCheckInitialConditions(RefactoringStatus status, IProgressMonitor pm)
    throws PreconditionFailure {

    ensureProjectHasRefactoringEnabled(status);
    // Exclude from the list to refactor the fixedForm files
    removeFixedFormFilesFrom(this.selectedFiles, status);
    // Exclude from the refactoring list those files with C preprocessed
    removeCpreprocessedFilesFrom(this.selectedFiles, status);
    // Check for errorStmtNodes

    try
    {
        for (IFile file : selectedFiles){

            ErrorConstructNodeVisitor errorFinder =
                  new ErrorConstructNodeVisitor();

            DuplicateLabelsFinderNodeVisitor duplicateLabelFinder=
                  new DuplicateLabelsFinderNodeVisitor();

            IFortranAST ast = vpg.acquirePermanentAST(file);

            if (ast == null)
                 status.addError("One of the selected files (" + file.getName()
                     + ") cannot be parsed.");

            ast.accept(errorFinder);

            if (errorFinder.thereAreErrors())
                    fail(Messages.StandInOuttFmtRefactoring_ErrorStmtNodesFound);

            ast.accept(duplicateLabelFinder);
```

```
            if (duplicateLabelFinder.thereAreErrors())
        fail(Messages.StandInOuttFmtRefactoring_ErrorFmtStmrLabeledWithSameValue);

            vpg.releaseAST(file);
        }
    }
    finally{
        vpg.releaseAllASTs();
    }
}
```

NOTE : Messages.StandInOuttFmtRefactoring stands for Messages.StandardizeInputOutputFormatsRefactoring

The error finder visitor checks for the existence of ErrorConstructionNodes, its code follows:

```
private static final class ErrorConstructNodeVisitor extends ASTVisitor{
    boolean error= false;

    @Override public void visitASTErrorConstructNode(ASTErrorConstructNode node){
        error=true;
        traverseChildren(node);
    }

    public boolean thereAreErrors() {
        return error;
    }
}
```

The DuplicateLablesFinderVisitor checks for distinct labels with the same name to prevent having source code with grammatical mistakes like two distinct labels with the same name. All this intricate work is required because Photran is allowed to apply a refactoring on source code not yet compiled:

```
private static final class DuplicateLabelsFinderNodeVisitor extends ASTVisitor{
  boolean error= false;
  // it contains all format Stmt

  private Map<String,ASTFormatStmtNode> fmtNodesMap=
     new HashMap<String,ASTFormatStmtNode>();

  @Override
  public void visitASTFormatStmtNode (ASTFormatStmtNode node){
      // Visit All FormatStmtNodes and put then in a list
      if (isLabeled(node)) {
          String Key =getKeyFormatStmt(node);
```

```
            if (!this.fmtNodesMap.containsKey(Key)) fmtNodesMap.put(Key,node);
            else error=true;
        }
    }

    public boolean thereAreErrors() {
        return error;
    }

    private boolean isLabeled(ASTFormatStmtNode node){
        return (node.getLabel()!=null);
    }

}
```

4. To perform the transformation on the source code we need to traverse the entire AST of each file, because it is a resource refactoring, we use the doCheckFinalConditions() Method.

```
@Override
protected void doCheckFinalConditions(RefactoringStatus status, IProgressMonitor pm)
   throws PreconditionFailure {

     try
     {
         for (IFile file : selectedFiles){
             IFortranAST ast = vpg.acquirePermanentAST(file);
             if (ast == null)
                 status.addError("One of the selected files (" + file.getName()
                     + ") cannot be parsed.");
             makeChangesTo(file, ast, status, pm);
             vpg.releaseAST(file);
         }
     }
     finally {
         vpg.releaseAllASTs();
     }
}
```

For each AST, all its scopes are analyzed.

```
private void makeChangesTo(IFile file, IFortranAST ast, RefactoringStatus status,
    IProgressMonitor pm) throws Error {
    try
    {
        if (ast == null) return;
        List<ScopingNode> scopes = ast.getRoot().getAllContainedScopes();
        for (ScopingNode scope : scopes)
            if (!(scope instanceof ASTExecutableProgramNode)
                && !(scope instanceof ASTDerivedTypeDefNode))
```

```
            standardizeInputOutputInScope((IASTListNode<IASTNode>)scope.getBody(), ast);
        addChangeFromModifiedAST(file, pm);

    }
    catch (Exception e) {
        throw new Error(e);
    }
}
```

For each scope, a visitor must gather four types of statements: FORMAT, PRINT, WRITE, READ. For this purpose, we need to use the InputOutputStatementsVisitor class that extends an ASTVisitor.

```
@SuppressWarnings("restriction")
private void standardizeInputOutputInScope(IASTListNode<IASTNode> body, IFortranAST ast) {

    InputOutputStatementsVisitor visitor = new InputOutputStatementsVisitor();

    body.accept(visitor);

    // get the list of new declarations

    IASTListNode<IBodyConstruct> newDeclarations =
        constructNewDeclarations(visitor.fmtNodesMap);

    // change format references with the string
    if (haveDeclarations(newDeclarations)){
        changeLabelsToStrings(visitor.inputOutputStmtsList, visitor.fmtNodesMap, ast);
        // add string declarations
        body.addAll(findIndexOfImplicitStmtIn(body), newDeclarations);
        // remove all formats statements
        Reindenter.reindent(newDeclarations, ast);
        for ( ASTFormatStmtNode item : visitor.fmtNodesMap.values() ){
            body.remove(item);
        }
    }

}

private boolean haveDeclarations(IASTListNode<IBodyConstruct> declarations){
     return (declarations!=null);
}
```

A complex helper class needed to perform this refactoring is the InputOutputStatementsVisitor class which is in charge of collecting the AST nodes required for this refactoring. Basically, it maintains a Map which has strings representing labels as keys and values which are ASTFormatStmtNode ob-

jects. It is this set of nodes which will be refactored. For each entry like `8011 FORMAT(20H W VELOITY  FOR J =,I4,12H AT TIMESTEP,I7)` an entry inside the map can be found.

It also possesses a linked list containing the Input or Output statements as IASTNodes (see Figure 6.13).

```
private static final class InputOutputStatementsVisitor extends ASTVisitor {
 // it contains all format Stmt
 private Map<String ,ASTFormatStmtNode> fmtNodesMap=
            new HashMap<String ,ASTFormatStmtNode >();

 // it contains all InputOutput
 private List<IASTNode> inputOutputStmtsList= new LinkedList<IASTNode>();

 public Map<String , ASTFormatStmtNode> getFmtNodesMap(){
     return fmtNodesMap;
 }

 public List<IASTNode> getInputOutputStmtsList(){
     return inputOutputStmtsList;
 }

 @Override
 public void visitASTFormatStmtNode (ASTFormatStmtNode node) {
     // Visit All FormatStmtNodes and put then in a list
     if (isLabeled(node)){
         String Key =getKeyFormatStmt(node);
         if (!this.fmtNodesMap.containsKey(Key)) fmtNodesMap.put(Key,node);
     }
 }

 @Override
 public void visitASTWriteStmtNode( ASTWriteStmtNode node){
   // visit All WriteStmtnodes add them to InputOutputStmtsList
   List<ASTIoControlSpecListNode> ioControlSpec = node.getIoControlSpecList();

   for( ASTIoControlSpecListNode specNode : ioControlSpec){

      ASTFormatIdentifierNode fmtIdentifierNode=
                              specNode.getFormatIdentifier();

      if (IsValidNode(fmtIdentifierNode)){
          ASTLblRefNode fmtLabel= fmtIdentifierNode.getFormatLbl();
          if (isValidNode(fmtLabel)){
              String Key = fmtLabel.getLabel().toString();
             if (!Key.equals(null)) inputOutputStmtsList.add(node);
          }
      }
   }
```

```
    }


    @Override
    public void visitASTReadStmtNode( ASTReadStmtNode node)
    {
        // visit All ReadStmtnodes add them to InputOutputStmtsList
        ASTRdCtlSpecNode rdCtlSpec = node.getRdCtlSpec();

        IASTListNode<ASTRdIoCtlSpecListNode> rdIoCtlSpecList=
                rdCtlSpec.getRdIoCtlSpecList();

        for( ASTRdIoCtlSpecListNode specNode : rdIoCtlSpecList){
            ASTFormatIdentifierNode fmtIdentifierNode=specNode.getFormatIdentifier();

            if (isValidNode(fmtIdentifierNode)){
                ASTLblRefNode fmtLabel= fmtIdentifierNode.getFormatLbl();
                if (isValidNode(fmtLabel)){
                    String Key = fmtLabel.getLabel().toString();
                    if (!Key.equals(null)) inputOutputStmtsList.add(node);
                }
            }
        }
    }


    @Override
    public void visitASTPrintStmtNode( ASTPrintStmtNode node){

        // visit All WriteStmtnodes add them to InputOutputStmtsList

        ASTFormatIdentifierNode fmtIdentifierNode= node.getFormatIdentifier();
        ASTLblRefNode fmtLabel= fmtIdentifierNode.getFormatLbl();

        if (isValidNode(fmtLabel)){
            String Key = fmtLabel.getLabel().toString();
            if (!Key.equals(null)) inputOutputStmtsList.add(node);
        }
    }

    private boolean IsValidNode(IASTNode node) {
        return (node != null);
    }

    private boolean isLabeled(ASTFormatStmtNode node){
        return (node.getLabel()!=null);
    }
   }
```

5. Helper Methods :

To perform the Standardize Input Output refactoring some helper methods were implemented. As this is a complex refactoring, the transformation can be split into two parts. The first stage is the node collection. The second part is node rewriting. To perform this stage, four methods whose responsibilities lie on the rewriting of new nodes have been defined. As a consequence, constructNewDeclarations() is used to create the new string parameter declaration in order to hold the format strings. The methods rewritePrintStmt(), rewriteWriteStmt() and rewriteReadStmt() are in charge of rearranging the input or output statements. The code that follows shows these methods:

```
    private IASTListNode<IBodyConstruct>
constructNewDeclarations(Map<String,ASTFormatStmtNode> fmtNodesMap ) {
        int i=1;
        StringBuilder newStmts = new StringBuilder();

        // changes all the format statements with a string declaration
        for(ASTFormatStmtNode item : fmtNodesMap.values()){
            newStmts.append(getDeclaration(item) + EOL);
        }
        return parseLiteralStatementSequence(newStmts.toString());
    }


private void rewritePrintStmt(Map<String, ASTFormatStmtNode> fmtNodesMap, IASTNode node)
{
    ASTFormatIdentifierNode fmtId=((ASTPrintStmtNode)node).getFormatIdentifier();

    if (isValidFormatIdentifier(fmtId)){

        String label = fmtId.getFormatLbl().toString().trim();

        if (fmtNodesMap.containsKey(label)){
            String variableOutputList="";
            String colon=",";
            //build new print statement
            ASTOutputItemListNode outputItemList=
                    ((ASTPrintStmtNode)node).getOutputItemList();

            if (isValidOutputItemList(outputItemList)){
                variableOutputList=outputItemList.toString();
            }
            else colon=""; //$NON-NLS-1$
            String VariableName=getVariableName(fmtNodesMap.get(label));
            String newPrintSttm =
               "Print " + VariableName + colon + variableOutputList+ EOL;
```

```
                node.replaceWith(newPrintSttm);
            }
        }
    }

    private boolean IsValidFormatIdentier(ASTFormatIdentifierNode fmtId){
        return (fmtId!=null);
    }

    private boolean IsValidOutputItemList(ASTOutputItemListNode node){
        return (node!=null);
    }

    private void rewriteWriteStmt(Map<String, ASTFormatStmtNode> fmtNodesMap, IASTNode node){
        IASTListNode<ASTIoControlSpecListNode> IoControlSpecList =
            ((ASTWriteStmtNode)node).getIoControlSpecList();

        if (isValidIoControlSpecList(IoControlSpecList)) {
            for( ASTIoControlSpecListNode IoControlSpecListNode : IoControlSpecList) {

                ASTFormatIdentifierNode fmtIdentifierNode=
                    IoControlSpecListNode.getFormatIdentifier();

                ASTUnitIdentifierNode unitIdentifier=
                    IoControlSpecListNode.getUnitIdentifier();

                if (isValidFormatIdentifier(fmtIdentifierNode)){
                    ASTLblRefNode lblRef=fmtIdentifierNode.getFormatLbl();
                    String label = lblRef.toString().trim();
                    if (fmtNodesMap.containsKey(label)){
                        //build new write statement
                        String variableOutputList=
                            ((ASTWriteStmtNode)node).getOutputItemList().toString();

                        String unitIdentStr=null;
                        if (unitIdentifier!=null) unitIdentStr=unitIdentifier.toString();

                        String newWriteStmt =
                                "write (" + unitIdentStr + ","
                                + getVariableName(fmtNodesMap.get(label))
                                + ")" + variableOutputList+ EOL;

                        node.replaceWith(newWriteStmt);
                    }
                }
            }
        }
    }

    private boolean isValidIoControlSpecList(IASTListNode<IASTNode> IoControlList){
        return (IoControList!=null);
    }
```

```
private boolean isValidFormatIdentifier(IASTNode node){
    return (node!=null)
}

 private void rewriteReadStmt(Map<String, ASTFormatStmtNode> fmtNodesMap, IASTNode node) {

    ASTRdCtlSpecNode rdCtlSpec = ((ASTReadStmtNode)node).getRdCtlSpec();

    IASTListNode<ASTRdIoCtlSpecListNode> rdIoCtlSpecList=
                                          rdCtlSpec.getRdIoCtlSpecList();

    String unitAndFmt="";
    String otherSpec="";
    String inputItemList= ((ASTReadStmtNode)node).getInputItemList().toString();

    if (hasIoControlCpecList(rdIoCtlSpecList)) {
        for( ASTRdIoCtlSpecListNode specNode : rdIoCtlSpecList) {
            ASTFormatIdentifierNode fmtIdentifierNode=specNode.getFormatIdentifier();
            if (hasFormatIdentifier((fmtIdentifierNode)){
                ASTUnitIdentifierNode unitIdentifier=specNode.getUnitIdentifier();
                if (unitIdentifier!=null) unitAndFmt =unitIdentifier.toString();
                ASTLblRefNode lblRef=fmtIdentifierNode.getFormatLbl();
                String label = lblRef.toString().trim();
                if (label!=null){
                    if (fmtNodesMap.containsKey(label))
                      unitAndFmt=
                unitAndFmt.concat(",").concat(getVariableName(fmtNodesMap.get(label)));
                }
            }
            else otherSpec=otherSpec.concat(specNode.toString());
        }
    }
    String newWriteStmt = "read (" + unitAndFmt+ otherSpec + ")" + inputItemList+ EOL;
    node.replaceWith(newWriteStmt);
 }
```

Some images can be appreciated at Figures 6.8 , 6.9 and 6.10

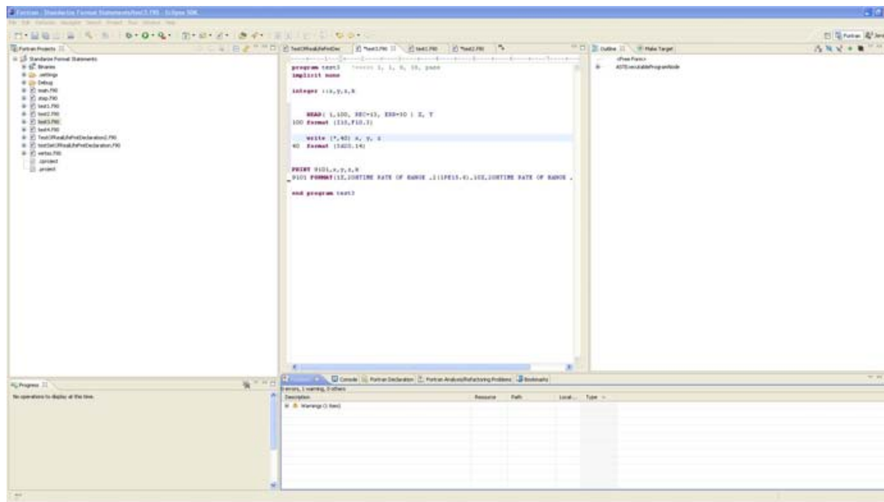

Figure 6.8: Fortran source code before applying the refactoring

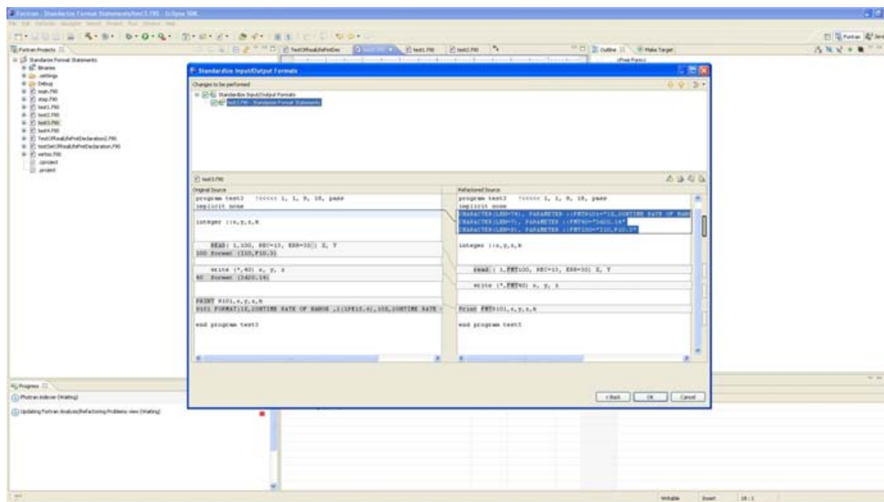

Figure 6.9: The Diff-view of Standardize IO Refactoring

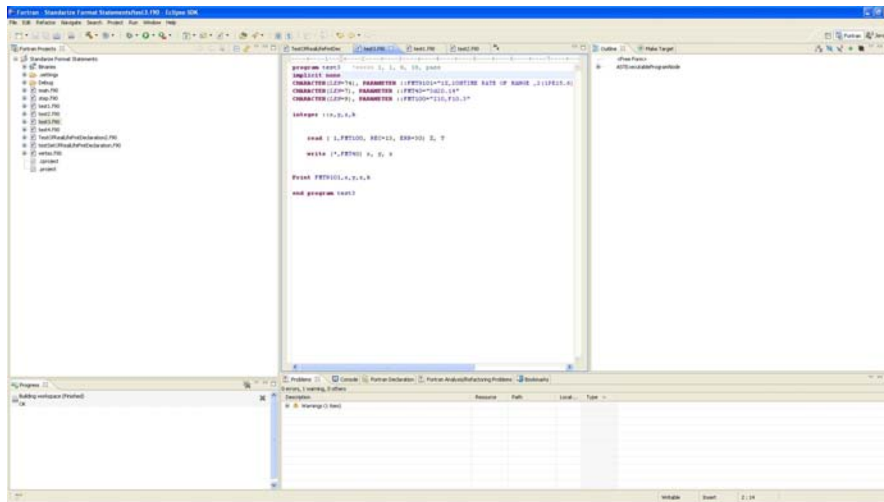

Figure 6.10: Fortran source code after applying the refactoring

6.4 Replace Old Style Do Loops Refactoring

6.4.1 Inception

A further refactoring we implemented is called Replace Old-Style Do-Loops [67, 75]. There are many different ways to write a do-loop in Fortran depending on what version of Fortran is being used. “Old-style” do-loops contain a numeric statement label in the loop header; the statement with that label constitutes the end of the loop (see Figure 6.11). On the other hand, “new-style” do-loops consist of matched DO/END DO pairs, which are generally preferred (see Figure 6.12).

```
    ....                   ....
    DO 100 I=1,30          DO 100 I=1,30
    V(I)=0             100 V(I)=0
100 CONTINUE               ....
    ....                   ....
```

Figure 6.11: Old-Style Fortran Do Loops

```
....                    ....
DO I=1,30               DO I=1,30
   V(I)=0                  100  V(I)=0
   100 CONTINUE         END DO
END DO                  ....
```

Figure 6.12: New-Style Fortran Do Loops

6.4.2 The Design

Replace Old-Style Do-Loops was implemented as an editor refactoring in Photran as follows:

Preconditions:

- The source code must have at least one do-statement.

- The terminating statement label for each old-style do-loop must be unique.
- The terminating statement must be at the same level of the nesting as the do-statement. For example, the terminating statement cannot be inside an if-construct in the loop. Regarding the complexity of this refactoring it was designed as an Editor refactoring allowing the user to refactor the entire selected file although it works as a resource refactoring, multiple files transformation can be hard to handle, a future version of it could handle multiple files can be refactored.

Transformation:
This refactoring transforms all old-style do-loops in the selected file into new-style do-loops. An END DO statement is inserted immediately following the terminating statement for each old-style do-loop. The statement label is removed from the loop header, and the loop body is re-indented.

One of the most complex aspects to handle is the one in connection with the modifying of the LoopReplacer class in order to make it capable of recognizing old style do loop in a more sophisticated AST node structure. LoopReplacer class is in charge of rewriting it as a “proper” structure. “ Due to a deficiency in the parser, DO-constructs are not recognized as a single construct; DO and END DO statements are recognized as ordinary statements alongside the statements comprising their body. ” [64].

6.4.3 The Implementation

A new subclass of FortranEditorRefactoring was added in the `org.eclipse.photran tran.internal.core.refactoring`. This class maintains two lists. The first one will retain a reference to each AST DO node statement in order to keep a reference to each Do Loop statement in the source code. The second list will keep a reference to each label in the Fortran source code:

```
public class ReplaceOldStyleDoLoopRefactoring extends FortranEditorRefactoring{

    private List<ASTProperLoopConstructNode> loopList=
                    new LinkedList<ASTProperLoopConstructNode>();

    private List<IActionStmt> lblList= new LinkedList<IActionStmt>();

}
```

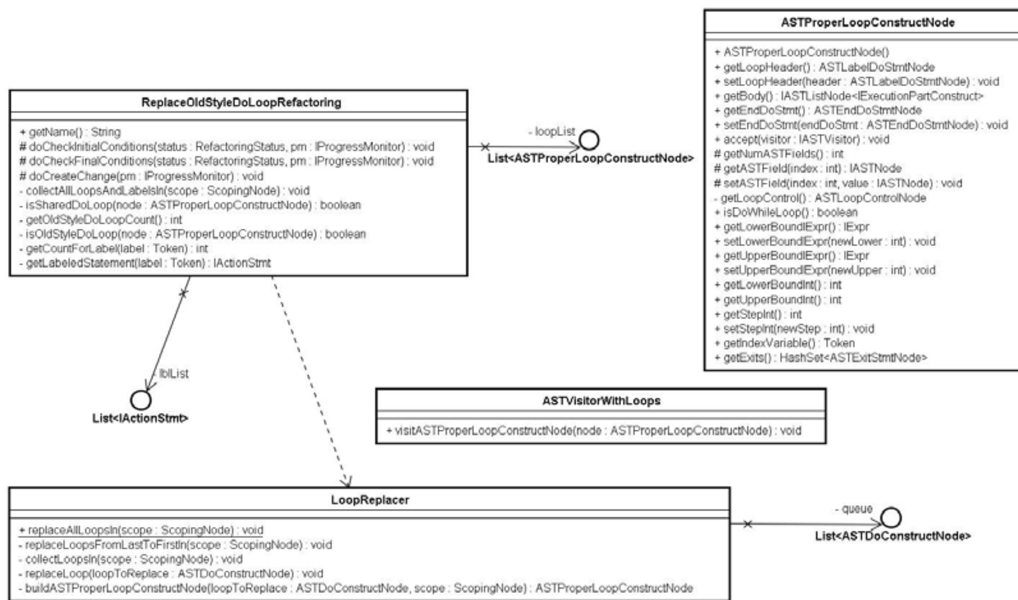

Figure 6.13: Photran Replace Old Style Do Loops Refactoring Class Diagram

1. As a first step the plugin.xml was edited:

```
<group><!-- Refactorings that eliminate old language constructs -->
  <editorRefactoring
 class="org.eclipse.photran.internal.core.refactoring.ReplaceOldStyleDoLoopRefactoring"
    />
```

   This step was performed to integrate the new refactoring into Photran User Interface.

2. Expose the name :

```
@Override
public String getName(){
        return Messages.ReplaceOldStyleDoLoopRefactoring_Name;
}
```

3. To check the initial preconditions a set of steps are performed by this method, the pseudo-code description is:

   Change AST to represent DO-loops as ASTProperLoopConstructNodes

   Collect All Loops and all labels

   Must have at least one OldStyle Do-Loop

for each oldStyle DoLoop

- there must be exactly one statement with the given "Label"
- it must be at the same level of the nesting as ASTDoStmt
- then the grandfather of the labeledStmt
- (must be the loopBody) == father of LoopHeader

```
 @Override
protected void doCheckInitialConditions(RefactoringStatus status, IProgressMonitor pm)
    throws PreconditionFailure {

 ensureProjectHasRefactoringEnabled(status);

 // Change AST to represent DO-loops as ASTProperLoopConstructNodes

 LoopReplacer.replaceAllLoopsIn(this.astOfFileInEditor.getRoot());

 collectAllLoopsAndLabelsIn(this.astOfFileInEditor.getRoot());

 //must have at last one OldStyle Do-Loop

 if (getOldStyleDoLoopCount()==0)
     fail(Messages.RepOldDoLoopRef_ThereMustBeAtLeastOneOldStyleDoLoop);

 // for each oldStyle DoLoop

 for(ASTProperLoopConstructNode node : loopList){
     if (isOldStyleDoLoop(node)){
         verifyLabel(node);
         if (!isSharedDoLoop(node)) {
             checkNodeLevel(node)
         }
     }
 }
}

private void checkNodeLevel(ASTProperLoopConstructNode node){
   // it must be at the same level of the nesting as ASTDoStmt
   // then the grandpa of the labeledStmt
   // (must be the loopBody) == father of LoopHeader
  IActionStmt labeledStmt=
            getLabeledStatement(loopHeader.getLblRef().getLabel());
  IASTNode loopBody=labeledStmt.getParent();
  if (loopBody.getParent()!=loopHeader.getParent())
      fail(Messages.bind(
            Messages.RepOldDoLoopRef_EndOfLoopError, labelName)
      );
}
```

```
private void verifyLabel(ASTProperLoopConstructNode node){

    //there must be exactly one statement with the given "Label"
    ASTLabelDoStmtNode loopHeader=node.getLoopHeader();
    int labelCount=getCountForLabel(loopHeader.getLblRef().getLabel());
    String labelName=loopHeader.getLblRef().getLabel().getText();

     if (labelCount>1)
         fail( Messages.bind(Messages.RepOldDoLoopRef_AmbiguousLabel,labelName ));
     else if (labelCount<1)
          fail(Messages.bind(
                      Messages.RepOldDoLoopRef_MissingLabel,labelName)
              );

}
```

Note: Messages.RepOldDoLoopRef stands for Messages.ReplaceOldStyleDoLoopRefactoring

By this moment, all the do loop constructions and labels nodes should have been gathered and transformed to an ASTPropeLoopConstructionNode. Once all nodes have been put together, the initial conditions required for the transformation are checked. If one of them fails, the entire process stops.

4. Perform the transformation:

   To refactor the source code in this case we need to iterate the entire do loop construction list so as to find each do loop node that is an old style do loop. In the case we have found an old style do loop, an END DO statement is added at the end of the construction. To finish it, the label referenced in the loop header is removed. The steps are described as follows:

   For Each Old Style DoLoop in the list

   - If the node is an Old Style Do loop
     - Add an END DO Statement
     - Remove from the Loop Header the label Reference
     - Re-indent the node

   The transformation can be seen at figure 6.14

```
    ....                  ....
    DO 100 I=1,30         DO I=1,30
    V(I)=0                   V(I)=0
100 CONTINUE                 100 CONTINUE
    ....                  END DO
    ....                  ....
```

Figure 6.14: AST Node Rewriting

```
@Override
protected void doCreateChange(IProgressMonitor pm)
            throws CoreException, OperationCanceledException{

        // For Each Old Style DoLoop in the list

        for(ASTProperLoopConstructNode node : loopList){
            if (isOldStyleDoLoop(node)){
                ASTEndDoStmtNode newNode =
                        (ASTEndDoStmtNode) parseLiteralStatement ("END_DO" + EOL);

                // Add and END DO Statement
                node.setEndDoStmt(newNode);

                // Remove from the Loop Header the label Reference
                node.getLoopHeader().setLblRef(null);

                // Re-indent the node
                Reindenter.reindent( node,
                                     this.astOfFileInEditor,
                                     Strategy.REINDENT_EACH_LINE);
            }
        }
        this.addChangeFromModifiedAST(this.fileInEditor, pm);
        vpg.releaseAST(this.fileInEditor);

    }
```

5. Helper Methods :

As expected, to perform the Replace Old Style Do Loops refactoring some helper methods have been implemented. For this refactoring the most complex work is carried out in the recognition of the old style constructions. In order to get this work done, this class, the LoopReplacer, becomes respon-

sible for getting all do loop nodes (see Figure 6.15) and translate them into an improved node structure (see Figure 6.16).

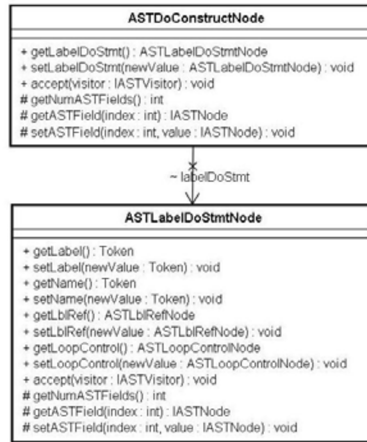

Figure 6.15: ASTDoConstructNode Class Diagram

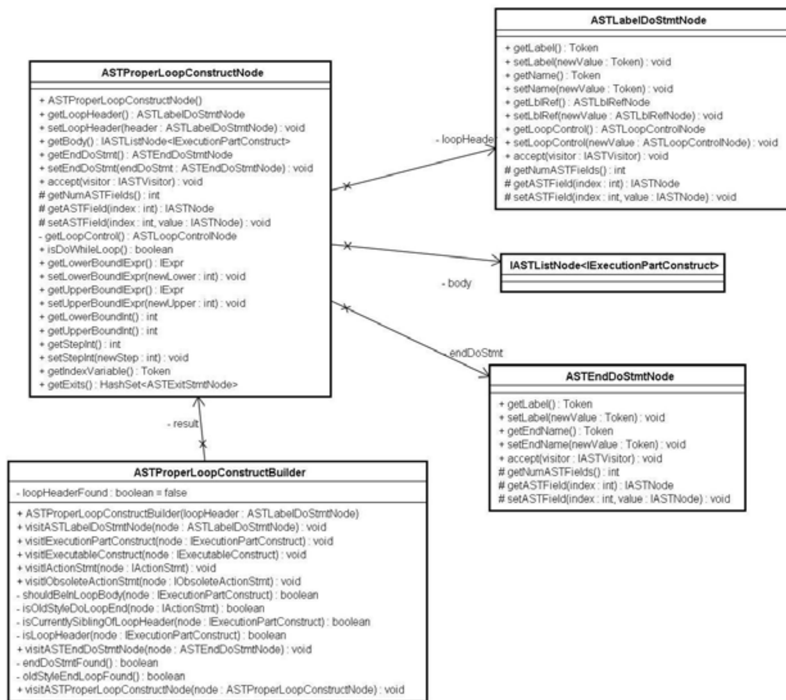

Figure 6.16: ASTProperLoopConstructNode Class Diagram

This new AST loop node has been divided into tree components: the loop header, loop body and the loop end. In order to recognize all types of do loop construction, the LoopReplacer class has been entirely modified.

```
public class ASTProperLoopConstructNode extends ASTNode implements IExecutableConstruct{
    private ASTLabelDoStmtNode loopHeader;
    private IASTListNode<IExecutionPartConstruct> body;
    private ASTEndDoStmtNode endDoStmt;
}
```

The LoopReplacer Class:

All the vital work for this refactoring is based on the work of this class. Initially, it recognized only one kind of do loop constructions: the ones that we can call modern do loops finishing with the END DO statement. The new class must be able to recognize old style do loops and shared do loops. To perform this work the LoopReplacer class must traverse the entire AST structure searching for ASTDoConstructNodes inserted in it by the Parser. This job is done by the collectLoopsIn() method. After all loops nodes are collected the replacement is performed by visiting each node and translating the old node structure into the new one. This class was originally written by Jeff Overbey.

```
public class LoopReplacer
{
  public static void replaceAllLoopsIn(ScopingNode scope) {
      new LoopReplacer().replaceLoopsFromLastToFirstIn(scope);
  }

  /** A list of all the loops in scope, from last to first */
  private List<ASTDoConstructNode> queue = new LinkedList<ASTDoConstructNode>();

  private void replaceLoopsFromLastToFirstIn(ScopingNode scope) {
      collectLoopsIn(scope);
      while (!queue.isEmpty())
          replaceLoop(queue.remove(0));
  }

  private void collectLoopsIn(ScopingNode scope){
      scope.accept(new ASTVisitor(){
          @Override public void visitASTDoConstructNode(ASTDoConstructNode node){
              // Collect all DoLoopsStmt
```

```
                queue.add(0, node);
            }
            } );
    }
    private void replaceLoop(ASTDoConstructNode loopToReplace) {

        // Save ancestor nodes, since parent pointers will
        // be changed when we manipulate the AST in
        // #buildASTProperLoopConstructNode below.

        IASTNode oldParent = loopToReplace.getParent();

        ScopingNode scope =
                loopToReplace.findNearestAncestor(ScopingNode.class);

        // Now manipulate the AST
        ASTProperLoopConstructNode newLoop =
                buildASTProperLoopConstructNode(loopToReplace, scope);

        loopToReplace.replaceWith(newLoop);
        newLoop.setParent(oldParent);
    }

    private ASTProperLoopConstructNode buildASTProperLoopConstructNode
        (ASTDoConstructNode loopToReplace, ScopingNode scope){

        ASTLabelDoStmtNode lastLoopHeader = loopToReplace.getLabelDoStmt();

        // First, remove siblings of lastLoopHeader
        // that should actually be in the loop body

        ASTProperLoopConstructBuilder nodeBuilder =
              new ASTProperLoopConstructBuilder(lastLoopHeader);

        scope.accept(nodeBuilder);

        // We needed to keep the loop header in the AST
        // so that that ASTProperLoopConstructBuilder could find
        // Now that it's finished, we can move the loop header
        // into the ASTProperLoopConstructNode

        lastLoopHeader.removeFromTree();
        nodeBuilder.result.setLoopHeader(lastLoopHeader);
        return nodeBuilder.result;
    }

    private class ASTProperLoopConstructBuilder extends ASTVisitorWithLoops {

        private final ASTProperLoopConstructNode result =
                    new ASTProperLoopConstructNode();

        private final ASTLabelDoStmtNode loopHeader;
```

```
 private final IASTNode doConstructNode;
 private final IASTNode listEnclosingDoConstructNode;
 private boolean loopHeaderFound = false;
 private IASTNode oldStyleEndLoopRef=null;

 // First, save ancestor nodes, since parent pointers will be changed when we
 // manipulate the AST in the #visit methods below

 public ASTProperLoopConstructBuilder(ASTLabelDoStmtNode loopHeader){
     this.loopHeader = loopHeader;
     this.doConstructNode = loopHeader.getParent();
     this.listEnclosingDoConstructNode = doConstructNode.getParent();
     this.oldStyleEndLoopRef=null;
 }

 // Start accumulating body statements when we find the loop header
 @Override public void visitASTLabelDoStmtNode(ASTLabelDoStmtNode node){
    if (node == loopHeader)loopHeaderFound = true;
    traverseChildren(node);
}

// Accumulate all statements between the loop header and the END DO stmt
@Override public void visitIExecutionPartConstruct(IExecutionPartConstruct node){
    if (shouldBeInLoopBody(node)){
        node.removeFromTree();
        this.result.getBody().add(node);
    }
}

 @Override public void visitIExecutableConstruct(IExecutableConstruct node){
     visitIExecutionPartConstruct(node);
 }

@Override public void visitIActionStmt(IActionStmt node) {
    //  Obtain a reference to the end of the old Style Loop Node
    visitIExecutionPartConstruct(node);
    if (isOldStyleDoLoopEnd(node)){
        this.result.setEndDoStmt(null);
        this.oldStyleEndLoopRef=node;
    }
    //traverseChildren(node);
}

@Override public void visitIObsoleteActionStmt(IObsoleteActionStmt node) {
    visitIExecutionPartConstruct(node);
}

private boolean shouldBeInLoopBody(IExecutionPartConstruct node) {
    return loopHeaderFound
        && !endDoStmtFound()
        && !oldStyleEndLoopFound()
        && !isLoopHeader(node)
```

```
        && isCurrentlySiblingOfLoopHeader(node);
}

private boolean isOldStyleDoLoopEnd( IActionStmt node){
    if ( (node.getLabel()!=null) && (this.loopHeader.getLblRef()!=null) ) {
        return loopHeaderFound
            && !endDoStmtFound()
            && !(node.getParent() == this.listEnclosingDoConstructNode)
            && ( this.loopHeader.getLblRef().getLabel().getText() ==
                   node.getLabel().getText() ) ;
    }
    return false;
}


private boolean isCurrentlySiblingOfLoopHeader(IExecutionPartConstruct node) {
    return node.getParent() == listEnclosingDoConstructNode;
}

// Don't accumulate either the ASTLabelDoStmtNode or
// the ASTDoConstructNode in the body; these are the header

private boolean isLoopHeader(IExecutionPartConstruct node) {
    return node == loopHeader || node == doConstructNode;
}

// Stop accumulating body statements as soon as we find an END DO stmt
@Override public void visitASTEndDoStmtNode(ASTEndDoStmtNode node) {
  if (loopHeaderFound && !endDoStmtFound()
      && (node.getParent() == listEnclosingDoConstructNode)
      && !oldStyleEndLoopFound() ) {
        node.removeFromTree();
        this.result.setEndDoStmt(node);
    }

    traverseChildren(node);
}

private boolean endDoStmtFound() {
    return this.result.getEndDoStmt() != null;
}

private boolean oldStyleEndLoopFound() {
    return this.oldStyleEndLoopRef != null;
}


@Override public void visitASTProperLoopConstructNode(ASTProperLoopConstructNode node){
    // Do not traverse child statements of nested loops
    // Except if you are working with a Shared Do Loop Termination
    // you need to know where the ending Loop is
```

```
            if (node.getLoopHeader().getLblRef()==null) return;
            if (this.loopHeader.getLblRef()==null) return;

            String nodeLabel=node.getLoopHeader().getLblRef().getLabel().getText();
            String headerLabel= this.loopHeader.getLblRef().getLabel().getText();

            if ( !endDoStmtFound() && !oldStyleEndLoopFound()
                    && nodeLabel.equals(headerLabel)   ) {
                visitIExecutionPartConstruct(node);
                this.oldStyleEndLoopRef=node.getLoopHeader().getLblRef();
            }
        }
    }
}
```

Figures 6.17, 6.18 and 6.19 show the refactoring process.

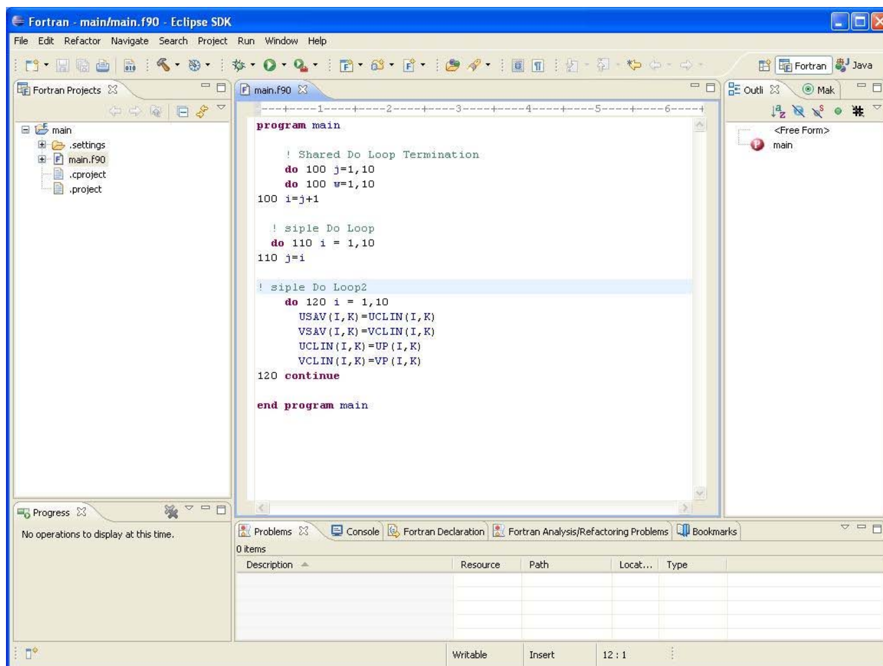

Figure 6.17: Old style do loop source code

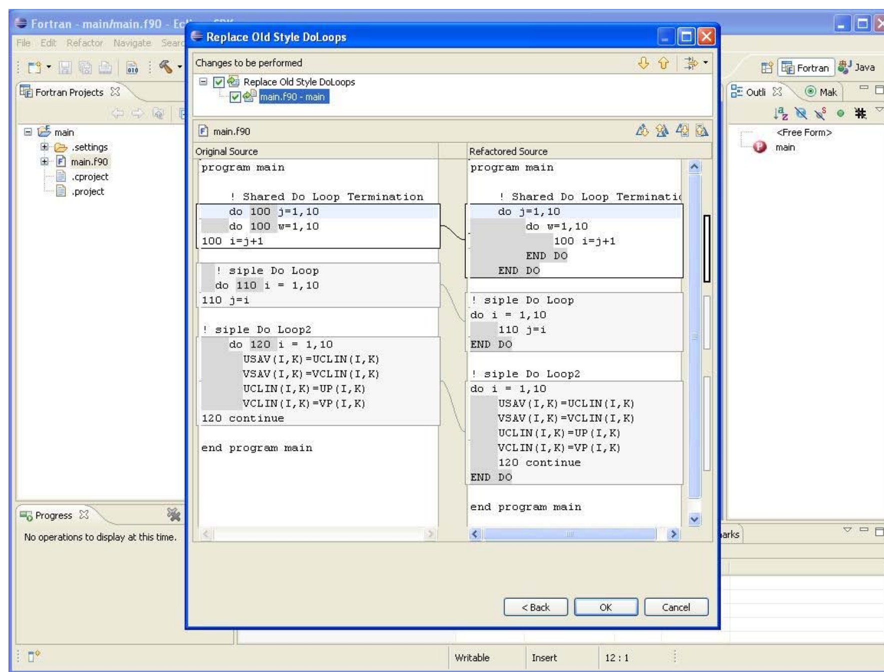

Figure 6.18: Replace Old Style Do Loop Diff-view

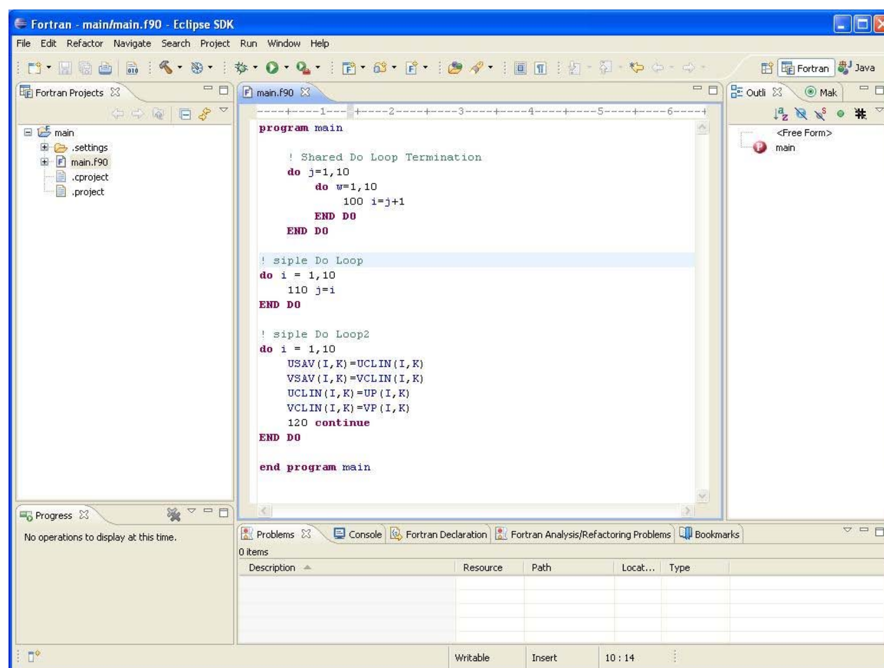

Figure 6.19: The source code refactored

6.5 Remove Unreferenced Labels Refactoring

6.5.1 Inception

The motivation of this refactoring is to remove from source code those labels no longer referenced by any statement. This is possible because the label has never been referenced or because a previous refactoring was applied and the label is no longer referenced.

6.5.2 The Design

To remove unreferenced labels, we must first recognize all the labels in the source code. To do this the refactoring class must have a list of labels gathered when visiting the AST. Additionally, we need to find out how many references there are in the code for each label. Finally, we will remove from the source code each label with the reference count equal to 0 (See Figure 6.20).

6.5.3 The Implementation

A new subclass of FortranEditorRefactoring was added to the project with a Map in which a reference count will be maintained for each labeled statement. The map is needed because the AST does not provide such information. This refactoring was created as an editor refactoring in order to refactor only the editor selected file. An improvement can be made by transforming it into a resource refactoring.

```
public class RemoveUnreferencedLabelsRefactoring extends FortranEditorRefactoring {
    private Map<String,Integer> labelMap;
}
```

The implementation steps:

1. The plugin.xml was edited, this step has been performed to integrate the new refactoring into photran User Interface.:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
   <!--=================-->
   <!-- Refactorings -->
```

```
<!--=================-->

<!-- NOTE: When adding refactorings, please update
     http://wiki.eclipse.org/PTP/photran/refactorings -->
<extension
      point="org.eclipse.rephraserengine.ui.refactoring.refactorings">
   <resourceFilter
         class="org.eclipse.photran.internal.ui.vpg.PhotranResourceFilter" />
   <!-- Define the Refactor menu -->
   <group><!-- Refactorings that reformat code -->
      <editorRefactoring
class="org.eclipse.photran.internal.core.refactoring.RemoveUnreferencedLabelsRefactoring"
         />
   </group>
</extension>
```

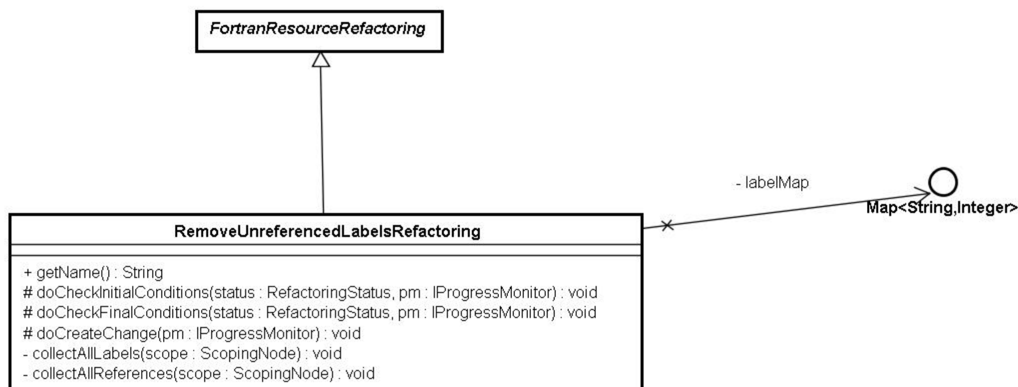

Figure 6.20: Photran Remove Unreferenced Labels Refactoring Class Diagram

2. Expose the name :

```
@Override
public String getName(){
   return Messages.RemoveUnreferencedLabelsRefactoring_Name;
}
```

3. Check the initial preconditions:

   This refactoring requires the existence of labels in the source code as a precondition. If none is found, the refactoring will not be performed. At the same time, the structure of the map is filled with the labels and their references by applying the visitor pattern. In order to obtain such references the collectAllLabels() and collectAllReferences() methods are used.

```
@Override
protected void doCheckInitialConditions(RefactoringStatus status,
IProgressMonitor pm) throws PreconditionFailure {

    labelMap= new HashMap<String, Integer>();
    ensureProjectHasRefactoringEnabled(status);
    collectAllLabels(this.astOfFileInEditor.getRoot());
    collectAllReferences(this.astOfFileInEditor.getRoot());
    if (labelMap.size()==0)
        fail(Messages.RemoveUnreferencedLabelsRefactoring_ThereMust);
}
```

4. Perform the transformation:

   To refactor the source code, we need to traverse the entire AST structure and visit all AST nodes; for each node, if it has a label with zero reference the label will be removed. In the event of being confronted with the zero referenced label and a CONTINUE statement, these two will be completely removed.

```
@Override
    protected void doCreateChange(IProgressMonitor pm)
 throws CoreException, OperationCanceledException
 {
        ScopingNode scope = this.astOfFileInEditor.getRoot();

        scope.accept(new ASTVisitor()
        {
            // Visit IActionStmt Nodes
            @Override public void visitIActionStmt (IActionStmt node)
            {
             // get the statements labeled
                if (hasLabel(node)) {
                    String key =node.getLabel().getText();
                    if ((labelMap.containsKey(key))&&(labelMap.get(key)==0){
                            // remove the label
                            node.setLabel(null);
                            if (node instanceof ASTContinueStmtNode){
                                node.removeFromTree();
                            }
                            Reindenter.reindent(node, astOfFileInEditor,
                            Strategy.REINDENT_EACH_LINE);
                        }
                    }
                }
            }

            private boolean hasLabel(IASTNode node){
                return(node.getLabel()!=null)
```

```
            }


        });
        this.addChangeFromModifiedAST(this.fileInEditor, pm);
        vpg.releaseAST(this.fileInEditor);
    }
```

5. Helper Methods : In order to perform the Replace Unreferenced labels refactoring some helper methods have been implemented. On this occasion the reference and the labels gathering must be performed separately to guarantee obtaining the complete set of labels before references are gathered, see figure 6.21. In order to do that, two visitors were implemented separately (collectAllLabels and collectAllReferences).

```
PROGRAM MAIN
 GOTO 100
 100 STOP
END PROGRAMMAIN
```

Figure 6.21: Fortran Program example

```
private void collectAllLabels(ScopingNode scope)
    {
        // Visit the AST
        scope.accept(new ASTVisitor(){
            // Visit IActionStmt Nodes
            @Override public void  visitIActionStmt (IActionStmt node){
                if (hasLabel(node)){
                    String key =node.getLabel().getText();
                    if (!labelMap.containsKey(key))
                        labelMap.put(key, new Integer(0));
                }
                traverseChildren(node);
            }

            private boolean hasLabel(IASTNode node){
                return(node.getLabel()!=null)
            }

        });
    }

    private void collectAllReferences(ScopingNode scope){
```

```
    // Visit the AST
    scope.accept(new ASTVisitor(){
        //ASTLblRefNode
        @Override public void visitASTLblRefNode(ASTLblRefNode node) {
            // get Label references count
            if (haslabel(node)){
                String key = node.getLabel().getText();
                if (labelMap.containsKey(key))
                    labelMap.put(key,(labelMap.get(key)+1));
            }
            traverseChildren(node);
        }

        private boolean hasLabel(IASTNode node){
            return(node.getLabel()!=null)
        }

    });
}
```

Some screenshots can be appreciated at Figures 6.22 and 6.23

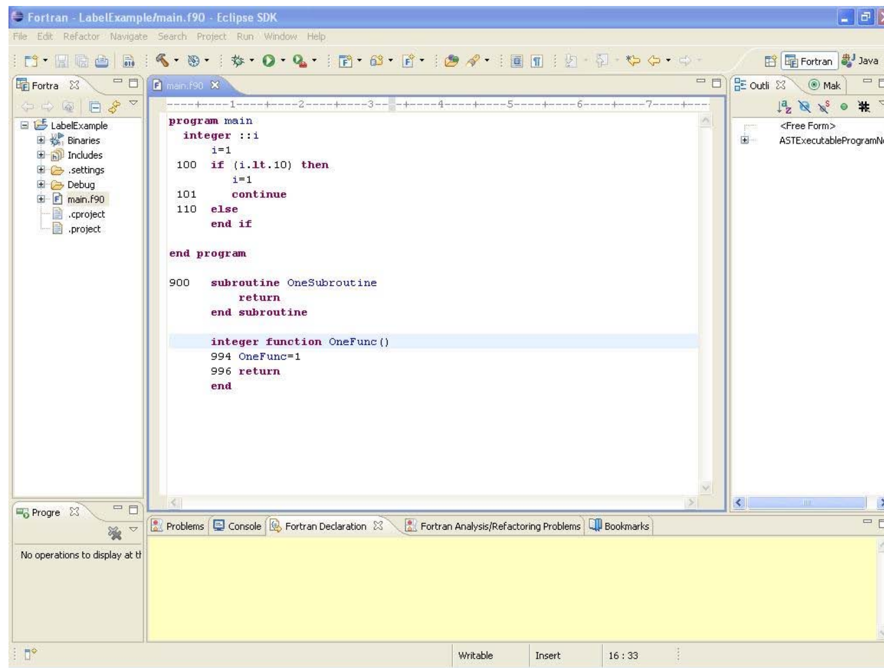

Figure 6.22: Fortran source code with unreferenced labels

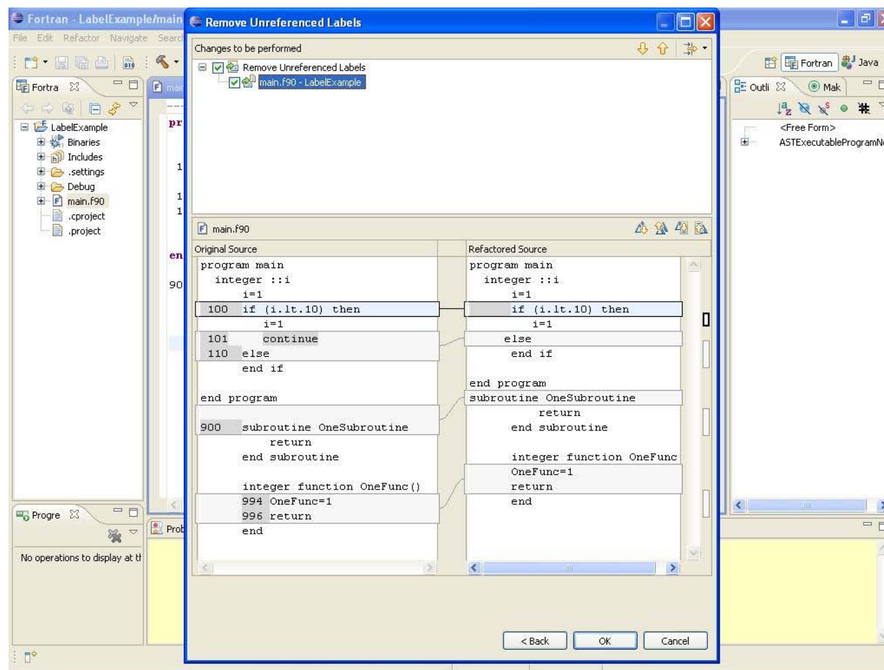

Figure 6.23: Diff-view of Remove Unreferenced Labels Refactoring

Chapter 7

Case Study

In this chapter, a real life source code will be transformed using the refactorings proposed in our catalog.

7.1 A Unit of Measurement

In the interest of understanding software improvement processes, it is natural for us to try to characterize the aspects of software that are affected in those processes. While the definition of a new readability or comprehensibility metric goes beyond the scope of this thesis, some kind of measurements are needed to be able to quantify the improvements achieved. One of the most important aspects that we aim to stress is that it is impossible to improve the design of unreadable source code. This unreadiness emerges from all those old and obsolete features of the language still valid, even in Fortran 2008. That is why refactoring is needed first to make the code readable.

There are different ways to measure source code readability [77, 73, 17]. These metrics are based on different source code characteristics, such as : McCabe Complexity [53], Halstead definition on complexity, LOC (Lines of Code), etc. On the other hand, it is also true that there are studies with a critical viewpoint on these metrics [39]. Since we believe that readability and comprehensibility are subjective to programmers we have defined a metric based on the premise

that the following language features make source code difficult to read and to understand:

1. GO TO statements by performing a one-way jump to another line of code.
2. Arithmetic IF, computed GO TO statements by jumping to one of several labels based on the value of an expression.
3. Labels and FORMAT statements by producing a tangled source code.
4. Obsolete operators, they are not compliant with modern operators.
5. COMMON BLOCKS by adding global behavior to variables.
6. Fixed Format by making rigid the way that source code is written.
7. Shared or Old Style DO LOOP, they use the old labeled notation.
8. Saved variables by adding complexity in the source code.

We define our magnitude as "Fortran Code Readability/Comprehensibility Scale" (FCRCS). This scale can be applied to a program, a module or a subroutine. The source code starts with a FCRCS=0 for each item from the list above found in the source code the FCRCS is increased by 1. It is convenient to note that the index is increased just only by one at the first occurrence of each language feature.

We make our hypothesis as:

"Source code with a high FCRCS is more difficult to read and understand, while source code with the FCRCS near to 0 is easier to read or understand."

7.2 Source Code Examples

The following source code has been taken from Fortran Programs for Scientists and Engineers, Second Ed., Copyright 1988, (SYBEX) ISBN 0-89588-571-9. As it can be seen in this book published in 1988 there are examples written in FORTRAN 77. In order to show the difficulty of reading some Fortran code, we have worked with a book example.

```
      program simq5
c
c -- fortran program to solve simultaneous equations
c -- by gauss-jordan elimination
c -- there may be more equations than unknowns
c -- subroutines square, gaussj and swap are also needed
c -- figure 4.12
c
      logical error
      integer maxr, maxc, out, n, m, index(8,3), nvec
      real a(8,8), y(8), coef(8), b(8,8)
      common /inout/ out, maxr, maxc, error
      data nvec/1/
c
      out = 6
      maxr = 8
      maxc = 8
      write(out, 101)
10    call input(a, y, n, m)
      if (m .lt. 2) goto 100
      call square(a, y, b, coef, n, m, maxr, maxc)
      call gaussj(b, coef, index, m, maxr, nvec, error, out)
      if (.not. error) call output(a, y, coef, n, m)
      goto 10
100   stop
101   format('1 best fit to simultaneous equations',
     * ' by gauss-jordan elimination')
      end
      subroutine input(a, y, n, m)
c
c --  get values for n and arrays a and y
c
      integer n, m, out, i, j, maxr
      real a(8,8), y(8)
      common /inout/ out, maxr, maxc, error
c
5     write(out, 107)
      read(*, 106) m
      if (m .gt. maxc) goto 5
      if (m .lt. 2) return
7     write(out, 105)
      read(*, 106) n
      if (n .lt. m) goto 7
      do 20 i = 1, n
       write(out, 101) i
       do 10 j = 1, m
        write(out, 102) j
        read(*, 103) a(i,j)
10     continue
       write(out, 104)
       read(*, 103) y(i)
20    continue
      return
101   format(' equation ', i3/)
102   format('+',i4, ': ' )
103   format(f10.0)
104   format('+ c: ' )
105   format(' how many equations? ' )
106   format(i2)
107   format(' how many unknows? ' )
      end
      subroutine output(a, y, coef, n, m)
c
c -- print the answers
c
```

```
      logical error
      integer n, m, out, i, j, maxr, maxc
      real a(8,8), y(8), coef(8)
      common /inout/ out, maxr, maxc, error
c
      do 10 i = 1, n
       write(out, 101) (a(i,j), j = 1, m), y(i)
10    continue
      write(out,*) '   solution'
      if (error) return
      write(out, 101) (coef(i), i = 1, m)
      return
101   format(1p6e12.4)
      end
```

7.2.1 Method

So as to study the improvements that have been achieved on the source code we propose to follow the next three steps:

1. Calculate the FCRCS index in the original source code in order to obtain the initial index value.
2. Apply the Fortran refactorings.
3. Re-calculate the FCRCS index and compare results.

7.2.2 Example 1

Our first application is to apply FCRCS to the subroutine called **input**. As an initial step we must determine the FCRCS value. In order to obtain this value a source code analysis is required to compute the existence of the following features:

Feature	Value
Shared or Old Style Do Loop	1
GO TO Statement	1
Arithmetic IF statement	0
Computed GOTO	0
Labeled statements	1
FORMAT statement	1
Obsolete operators	1
COMMON BLOCK	1
Fixed Format	1
Saved variables	0

Table 7.1: Language features found in the input routine source code.

As a result we obtain a FCRCS = 7 before applying the refactorings to the source code, see table 7.1.

```
        subroutine input(a, y, n, m)
c
c --  get values for n and arrays a and y
c
      integer n, m, out, i, j, maxr
      real a(8,8), y(8)
      common /inout/ out, maxr, maxc, error
c
5     write(out, 107)
      read(*, 106) m
      if (m .gt. maxc) goto 5
      if (m .lt. 2) return
7     write(out, 105)
      read(*, 106) n
      if (n .lt. m) goto 7
      do 20 i = 1, n
       write(out, 101) i
       do 10 j = 1, m
        write(out, 102) j
        read(*, 103) a(i,j)
10     continue
       write(out, 104)
       read(*, 103) y(i)
20    continue
      return
101   format(' equation ', i3/)
102   format('+',i4, ': ' )
103   format(f10.0)
104   format('+ c: ' )
105   format(' how many equations? ' )
106   format(i2)
107   format(' how many unknows? ' )
      end
```

Table 7.2: The input routine before being refactored.

As a second step we will apply the following refactorings to the subroutine:

- Remove old style DO Loops: to remove shared do loops or old style do loops.
- Standardize Input Output: to remove format statements away from the Input - Output statements.
- Replace Obsolete Operators: to update old Fortran logical operators with the new ones.
- Remove Unreferenced Labels: to remove labels no longer referenced.
- Change to free format: to allow a better way of code formating.

After applying those refactorings, we have obtained the subroutine refactored code as follows:

```
subroutine input(a, y, n, m)
    character(len=22), parameter ::FMT107="' how many unknows? ' "
    character(len=2), parameter ::FMT106="i2"
    character(len=24), parameter ::FMT105="' how many equations? ' "
    character(len=8), parameter ::FMT104="'+ c: ' "
    character(len=5), parameter ::FMT103="f10.0"
    character(len=13), parameter ::FMT102="'+',i4, ': ' "
    character(len=17), parameter ::FMT101="' equation ', i3/"
!
! --  get values for n and arrays a and y
!
    integer n, m, out, i, j, maxr
    real a(8,8), y(8)
    common /inout/ out, maxr, maxc, error
!
5   write (out,FMT107)
    read (*,FMT106) m
    if (m > maxc) goto 5
    if (m < 2) return
7   write (out,FMT105)
    read (*,FMT106) n
    if (n < m) goto 7
    do i = 1, n
        write (out,FMT101) i
        do j = 1, m
            write (out,FMT102) j
            read (*,FMT103) a(i,j)
        end do
        write (out,FMT104)
        read (*,FMT103) y(i)
    end do
    return
end
```

Table 7.3: The input routine after being refactored.

At this point we can have recalculated the FCRCS index, obtaining a new FCRCS value of 3.

This example brings about some remarkable aspects. First, the source code has 12 labels all around the code, after applying some refactorings the total labels amount decreases to 2. Labels make source code more difficult to understand and read.

Second, in the original source code seven FORMAT statements made it unclear and tangled. As a consequence of removing format statements from the source code body and using strings with the format options instead, the code became more readable.

Third, old style do loops were replaced with the proper END-DO statement producing more structured source code and by removing this obsolete construction from the code (see figure 7.1).

Fourth, the obsolete operators like .lt. .gt. etc. were replaced with the corresponding modern operators (see figure 7.2).

Finally, the free format allows programmers to determine properly the DO loops nested levels. At this point, it can be seen that by applying some refactorings on the old Fortran source code the program has been enhanced. Furthermore, the source code has become more similar to modern standards and it looks familiar to programmers working these days.

Shared or Old Style Do Loop	0
GO TO Statement	1
Arithmetic IF statement	0
Computed GOTO	0
Labeled statements	1
FORMAT statement	0
Obsolete operators	0
COMMON BLOCK	1
Fixed Format	0
Saved variables	0
Other features	0

Table 7.4: Language features found in the input routine source code after refactorings have being applied.

```
main.f90
program main
    implicit none
end program main

subroutine input(a, y, n, m)
!
! --  get values for n and arrays a and y
!
    integer n, m, out, i, j, maxr
    real a(8,8), y(8)
    common /inout/ out, maxr, maxc, error
!
5   write(out, 107)
    read(*, 106) m
    if (m > maxc) goto 5
    if (m < 2) return
7   write(out, 105)
    read(*, 106) n
    if (n < m) goto 7
    do i = 1, n
        write(out, 101) i
        do j = 1, m
            write(out, 102) j
            read(*, 103) a(i,j)
10          continue
        END DO
        write(out, 104)
        read(*, 103) y(i)
20      continue
    END DO
      return
101   format(' equation ', i3/)
102   format('+',i4, ': ' )
103   format(f10.0)
104   format('+ c: ' )
105   format(' how many equations? ' )
106   format(i2)
107   format(' how many unknows? ' )
      end
```

Figure 7.1: Fortran source code after Replace Old Style Do Loop refactoring.

```
main.f90
program main
    implicit none
end program main

subroutine input(a, y, n, m)
c
c --  get values for n and arrays a and y
c
    integer n, m, out, i, j, maxr
    real a(8,8), y(8)
    common /inout/ out, maxr, maxc, error
c
5   write(out, 107)
    read(*, 106) m
    if (m > maxc) goto 5
    if (m < 2) return
7   write(out, 105)
    read(*, 106) n
    if (n < m) goto 7
    do 20 i = 1, n
       write(out, 101) i
       do 10 j = 1, m
           write(out, 102) j
           read(*, 103) a(i,j)
10     continue
       write(out, 104)
       read(*, 103) y(i)
20    continue
      return
101   format(' equation ', i3/)
102   format('+',i4, ': ' )
103   format(f10.0)
104   format('+ c: ' )
105   format(' how many equations? ' )
106   format(i2)
107   format(' how many unknows? ' )
      end
```

Problems | Console | Fortran Declaration | Fortran Analysis/Refactoring Problems | Bookmarks

0 items

Description	Resource	Path	Locat...	Type

Figure 7.2: Fortran source code after Replace Obsolete Operators refactoring

7.2.3 Example 2

The subroutine called **output()** was analyzed as a second case of study, so it has been measured by applying the FCRCS obtaining a value of 5. Therefore, we can proceed to apply four refactorings:

- Change Fixed to Free Form
- Replace Old Style Do Loops
- Standardize Input Output
- Remove Unreferenced Labels

The initial source code is listed below:

```
     subroutine output(a, y, coef, n, m)
c
c -- print the answers
c
      logical error
      integer n, m, out, i, j, maxr, maxc
      real a(8,8), y(8), coef(8)
      common /inout/ out, maxr, maxc, error
c
      do 10 i = 1, n
       write(out, 101) (a(i,j), j = 1, m), y(i)
10    continue
      write(out,*) '   solution'
      if (error) return
      write(out, 101) (coef(i), i = 1, m)
      return
101   format(1p6e12.4)
```

Table 7.5: The output() routine before being refactored.

After the refactorings were applied, the source code was downgraded to FCRCS=1:

This code is almost equal to a current programming language code, except for the common block. The upgrading process can be seen in the Figures 7.3, 7.4. A more comprehensive measurement can be done but it is beyond the scope of this thesis.

```
subroutine output(a, y, coef, n, m)
      character(len=8), parameter ::FMT101="1p6e12.4"
!
! -- print the answers
!
      logical error
      integer n, m, out, i, j, maxr, maxc
      real a(8,8), y(8), coef(8)
      common /inout/ out, maxr, maxc, error
!
      do i = 1, n
          write (out,FMT101) (a(i,j), j = 1, m), y(i)
      end do
      write(out,*) '   solution'
      if (error) return
      write (out,FMT101) (coef(i), i = 1, m)
      return
end
```

Table 7.6: The output() routine after being refactored.

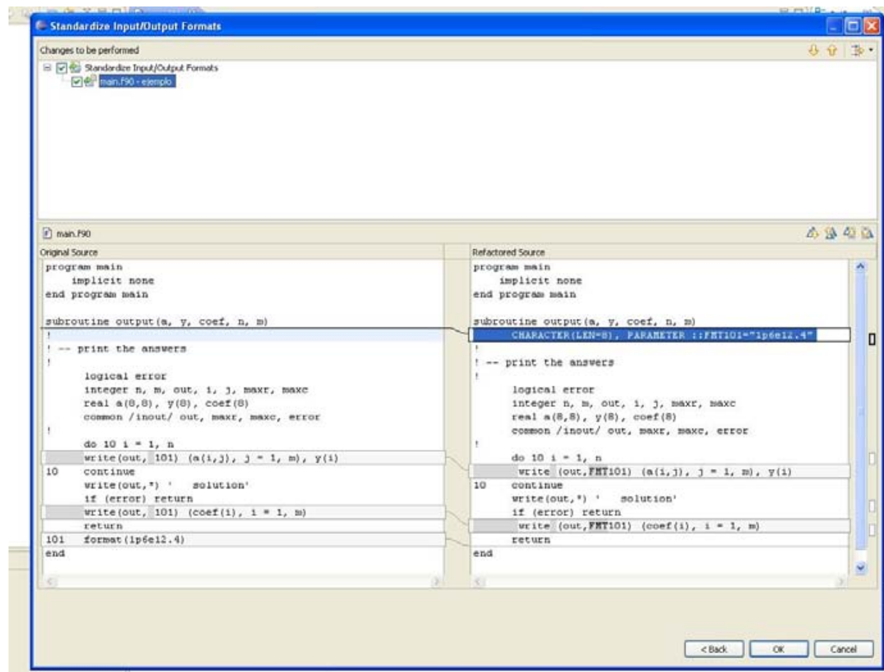

Figure 7.3: Fortran source code after Standardize Input Output

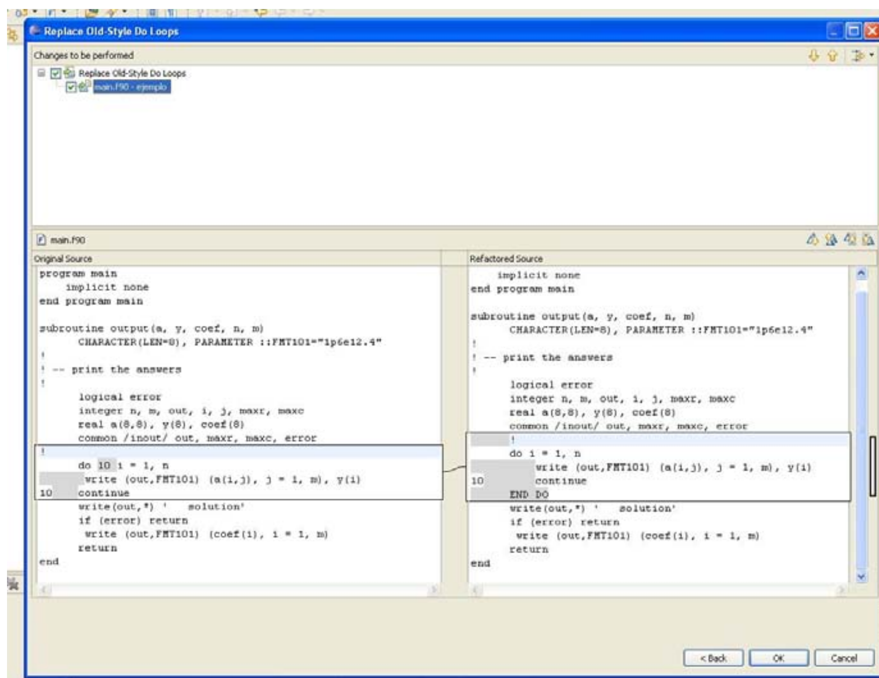

Figure 7.4: Fortran source code after Replace Old Style Do Loop refactoring.

7.2.4 Future Applications

In this chapter we have shown the fact that Fortran refactorings help source code to become updated, more comprehensible, more readable, etc. As a future application, large scale legacy systems can be refactored. A more comprehensive index can be specified, and as a consequence it can be applied to other programming languages.

Chapter 8

Conclusions

In this chapter we provide some concluding remarks as it revisits the contributions of this thesis and outlines the future work.

8.1 Results

This work has explored the way to update Fortran legacy system by using software refactoring as a main tool, using this technique as the first approach to get more readable and understandable legacy source code. Furthermore, long-lived programming languages need tools for allowing them to evolve. This kind of tools are not easily built, they require a refined engine to allow programmers to build refactorings.

Fortran has had a particular evolutionary process through different versions across time, about ten language versions have been published in the last 50 years (six of them were standards). These versions have transformed Fortran into a language with a rich set of syntactical constructions. As a consequence, programs written years ago are hard to read because of the lack of modern software engineering concepts such as software quality, development processes, etc.

Four new Fortran refactorings, built in this thesis, have been integrated into Photran's refactoring menu. These four refactorings are now part of the next public version of the tool called Photran 7.0. As an open source product, the

programmers contribution around the world makes it the most complete refactoring tool for Fortran ever built, with 70 % of the refactorings, proposed in this thesis, being implemented.

As a consequence of this research three articles have been published:

- " A Catalog and Classification of Fortran Refactoring" was presented in the 11th Argentine Symposium on Software Engineering (ASSE 2010). This short research article presents a catalog of source code refactorings that are intended to improve different quality attributes of Fortran programs. We have to classify the refactorings according to their purpose, that is, the internal or external quality attribute(s) that each refactoring targets to improve. We have proposed the implementation of one refactoring in Photran [55].

- " A Catalog and Two Possible Classifications of Fortran Refactorings" a more comprehensive description of each refactoring proposed in ASSE article has been presented as a technical report [56].

- "Refactorización en Código Fortran Heredado" (In Spanish) was presented in the XVI Congreso Argentino de Ciencias de la Computación(CACIC 2010). In this article a detailed review of Fortran evolution was presented together with a description of some implemented refactoring [54].

The following contributions have been made:

1. **A Classification of Fortran refactorings**: The way in which the refactorings were proposed is the result of how we think programmers need to use refactoring in their daily work. So we present the refactorings classified from the programmer's point of view.

2. **A Detailed catalog of Fortran refactorings**: Each refactoring proposed in this catalog has emerged from the Fortran programmer's needs. Our description rests on each refactoring motivation.

3. **A proposal of refactorings for parallelizing and performance improvements**: For some of these refactorings it has been proved that a much better performance existed [72]. A set of these transformations are closely related to those conducted by compilers to improve performance, like loop fusion or loop fission [27].

4. **A specification of some refactorings**: The implementation of a set of refactorings was explained in detailed and documented with the aim of providing a guide to be used in the initial steps in the refactoring built process.

5. **The use of refactorings on Fortran legacy systems**: In this work we have shown how to employ refactorings in the field of legacy systems. Furthermore, we have used refactoring applied to one of the most long-lived programming language such as Fortran.

6. **A metric definition**: We have presented a way to measure the source code transformation impact on source code readability as a metric called "FCRCS".

7. **A Contribution to Photran Project**: The refactorings implemented in this thesis will be all included in Photran 7.0 release.

8. **A public web site containing the catalog in different languages**: Aligned with the aims of this research, a public access web site was created to integrate and to promote Fortran refactorings and the eclipse-based-refactoring tool (Photran). This site was published in July 2010 [3].

8.2 Future Work

Although the refactoring has become an assessed technique for improving object oriented software without changing its external behavior, it manifests its utility in the process of understanding and maintaining software which was developed a long time ago, even when software development processes still had not seen

the light of the day. This work has brought about the need of a legacy software Process Model to help people understand, update and maintain this kind of software as we can still find requests from people who need to update or port FORTRAN IV program written years ago. The construction of a process model which helps programmers to deal with legacy system remains a challenge to this day.

As time goes by, legacy software has gained more and more lines of code, each one of them has increased the software complexity. For this reason components like refactoring tools are paramount on the day to day work, helping programmers to deal with legacy software.

Future work includes implementing more refactorings on Photran and applying them on some case studies to measure the overall improvement. Another important factor is to encourage the scientific world to use Photran. This will require not only successful stories about the use of Photran in large applications but also a formal foundation that ensures behavior preservation.

Another important aspect to be included as future work is closely related to the capacity of a tool to identify automatically, the places where a refactoring can be applied. These “refactoring points” will be automatically notified to the users so as to help them in the process of improving the internal structure of the software.

Bibliography

[1] http://www.quadibloc.com/comp/fort03.htm. [cited at p. 38, 39]

[2] http://en.wikipedia.org/wiki/Fortran. [cited at p. 40, 41, 45]

[3] http://www.fortranrefactoring.com.ar/. [cited at p. 11, 177]

[4] Cray Inc. http://www.cray.com/. [cited at p. 52]

[5] Photran, an Integrated Development Environment and Refactoring Tool for Fortran. http://www.eclipse.org/photran/. [cited at p. 14]

[6] RS Arnold. Software restructuring. *Proceedings of the IEEE*, 77(4):607–617, 1989. [cited at p. 15, 16]

[7] E. Ashcroft and Z. Manna. The translation of'goto'programs to'while'programs. *Information Processing*, 71:250–255, 1972. [cited at p. 17]

[8] J. Backus. The History of Fortran I, II, and III. *ACM SIGPLAN Notices*, 13(8):165–180, 1978. [cited at p. 35, 38]

[9] J. Backus. The history of Fortran I, II, and III. *ACM SIGPLAN Notices*, 13(8):165–180, 1978. [cited at p. 36]

[10] JW Backus, RJ Beeber, S. Best, R. Goldberg, LM Haibt, HL Herrick, RA Nelson, D. Sayre, PB Sheridan, H. Stern, et al. The FORTRAN Automatic Coding System. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 188–198. ACM, 1957. [cited at p. 36]

[11] Brenda S. Baker. An algorithm for structuring flowgraphs. *J. ACM*, 24(1):98–120, 1977. [cited at p. 17]

[12] I. Baxter, P. Pidgeon, and M. Mehlich. DMS: Program Transformations for Practical Scalable Software Evolution. In *Proceedings of the International Conference on Software Engineering, IEEE Press*, 2004. [cited at p. 13]

[13] K. Bennett. Legacy systems: Coping with success. *IEEE Software*, 12(1):19–23, 1995. [cited at p. 7]

[14] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371, 1966. [cited at p. 17]

[15] M.L. Brodie and M. Stonebraker. *Migrating legacy systems*. Morgan Kaufmann Publishers, 1995. [cited at p. 7]

[16] F.P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE computer*, 20(4):10–19, 1987. [cited at p. 5, 9]

[17] R.P.L. Buse and W.R. Weimer. A metric for software readability. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 121–130. ACM, 2008. [cited at p. 163]

[18] E Bush. The automatic restructuring of cobol. In *The Institute of Electrical and Electronics Engineers, Inc on Conference on software maintenance–1985*, pages 35–41, Piscataway, NJ, USA, 1985. IEEE Press. [cited at p. 17]

[19] R. Center. Perspectives on Legacy System Reengineering. 1995. [cited at p. 8]

[20] E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE software*, 7(1):13–17, 1990. [cited at p. 16]

[21] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, 7(1):13–17, 1990. [cited at p. 6]

[22] I.D. Chivers and J. Sleightholme. Compiler support for the Fortran 2003 and 2008 standards. In *ACM SIGPLAN Fortran Forum*, volume 28, pages 15–20. ACM, 2009. [cited at p. 49]

[23] M. Cohen. Fortran: A few historical details. http://www.nag.co.uk/nagware/np/doc/fhistory.asp, Oct. 2004. [cited at p. 36, 47]

[24] International Business Machines Corporation. *Reference manual [S]: FORTRAN II for the IBM 704 data processing system*. 1958. [cited at p. 37]

[25] Vaishali De. A Foundation for Refactoring Fortran 90 in Eclipse. Master's thesis, University of Illinois, 2004. [cited at p. 14]

[26] O.E. Dictionary. Oxford English Dictionary. [cited at p. 6]

[27] D. Dig. A Refactoring Approach to Parallelism. [cited at p. 11, 177]

[28] B. Foote and J. Yoder. Big ball of mud. *Pattern languages of program design*, 4(654-692):99, 2000. [cited at p. 6]

[29] A. FORTRAN. X3. 9-1978. *American National Standards Institute, New York*, 1978. [cited at p. 41]

[30] A. FORTRAN. X3.198-1992. *American National Standards Institute, New York*, 1992. [cited at p. 44, 45, 46]

[31] A.S. FORTRAN. X3. 9-1966. *American National Standards Institute Incorporated, New York*, 1966. [cited at p. 40]

[32] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999. [cited at p. 5, 53]

[33] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Reading, MA, 1995. [cited at p. 115]

[34] A. Garrido. Software Refactoring Applied to C Programming Language. Master's thesis, University of Illinois, 2000. [cited at p. 3, 6, 9, 14, 51]

[35] A. Garrido and R. Johnson. Refactoring C with Conditional Compilation. In *18th IEEE Int. Conf. on Automated Software Engineering*, 2003. [cited at p. 14]

[36] A. Garrido and R. Johnson. Program Refactoring in the Presence of Preprocessor Directives. *University of Illinois at Urbana-Champaign, Champaign, IL*, 2005. [cited at p. 3, 14]

[37] N.E. Gold. *The meaning of" legacy systems"*. Univ. of Durham, Dept. of Computer Science, 1998. [cited at p. 7]

[38] C. Greenough and D. Worth. The Transformation of Legacy Software: Some Tools and a Process. Technical report, RAL Technical Report TR-2003 012, 2004. [cited at p. 13]

[39] P.G. Hamer and G.D. Frewin. MH Halstead's Software Science-a critical examination. In *Proceedings of the 6th international conference on Software engineering*, pages 197–206. IEEE Computer Society Press, 1982. [cited at p. 163]

[40] JG HUST and CO. CRYOGENIC ENGINEERING LAB National Bureau of Standards, Boulder. Ibm 7090 fortran ii program for thermodynamic property computations. enthalpy-pressure or pressure-density as independent coordinates(Two Fortran II subroutine for calculation of thermodynamic properties of oxygen using density and pressure or enthalpy and pressure as independent coordinates). Technical report, National Bureau of Standards, Boulder, CO. CRYOGENIC ENGINEERING LAB, 1965. [cited at p. 38, 187]

[41] ISO. ANSI/ISO/IEC 1539-1:1997: Information technology — programming languages — Fortran — part 1: Base language. [cited at p. 47]

[42] ISO. ANSI/ISO/IEC 1539-1:2004(E): Information technology — Programming languages — Fortran Part 1: Base Language. pages xiv + 569, May 2004. [cited at p. 48]

[43] R. Johnson. Developing the refactoring browser. *Proceedings of XP2000*, 2000. [cited at p. 5, 14, 19]

[44] R. Johnson and W. Opdyke. Refactoring and aggregation. *Object Technologies for Advanced Software*, pages 264–278, 1993. [cited at p. 4]

[45] R.E. Johnson and B. Foote. Designing reusable classes. *Journal of object-oriented programming*, 1(2):22–35, 1988. [cited at p. 4]

[46] Brian W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. Computing Mcgraw-Hill, January 1978. [cited at p. 17]

[47] M.M. Lehman et al. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980. [cited at p. 5]

[48] H. Li, C. Reinke, and S. Thompson. Tool support for refactoring functional programs. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 27–38. ACM New York, NY, USA, 2003. [cited at p. 32]

[49] RC LINGER. MILLS˜ HD A case study m Cleanroom software engineering: The IBM˜ Cobol restructuring facility. *Proceedtngs of COMPSAC*, 88. [cited at p. 17]

[50] R.C. Linger, B.I. Witt, and HD Mills. *Structured Programming; Theory and Practice the Systems Programming Series*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1979. [cited at p. 17]

[51] DB Loveman. High Performance Fortran. *IEEE [see also IEEE Concurrency] Parallel & Distributed Technology: Systems & Applications*, 1(1):25–42, 1993. [cited at p. 52]

[52] Michael J. Lyons. Salvaging your software asset: (tools based maintenance). In *AFIPS '81: Proceedings of the May 4-7, 1981, national computer conference*, pages 337–341, New York, NY, USA, 1981. ACM. [cited at p. 17]

[53] T.J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, pages 308–320, 1976. [cited at p. 163]

[54] M. Méndez, A. Garrido, J. Overbey, F.G. Tinetti, and R. Johnson. Refactorización en Código Fortran Heredado. [cited at p. 176]

[55] M. Mendez, J. Overbey, A. Garrido, F. Tinetti, and R. Johnson. A catalog and classification of fortran refactorings. In *11th Argentine Symposium on Software Engineering (ASSE 2010*, pages 1–10, 2010. [cited at p. 176]

[56] M. Méndez, J. Overbey, A. Garrido, F. Tinetti, and R. Johnson. A Catalog and Two Possible Classifications of Fortran Refactorings. *Technical Report*, 2010. [cited at p. 176]

[57] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004. [cited at p. 3]

[58] HW Morgan. Evolution of a software maintenance tool. In *Proceedings of the 2nd Natwnal Conference EDP Software Maintenance*, pages 268–278, 1984. [cited at p. 17]

[59] W.F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Citeseer, 1992. [cited at p. 5, 9, 14, 53]

[60] W.F. Opdyke and R.E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, 1990. [cited at p. 4, 5, 14]

[61] W.F. Opdyke and R.E. Johnson. Creating abstract superclasses by refactoring. In *Proceedings of the 1993 ACM conference on Computer science*, pages 66–73. ACM, 1993. [cited at p. 4, 14]

[62] J. Overbey and C. Rasmussen. Instant IDEs: supporting new languages in the CDT. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, page 79. ACM, 2005. [cited at p. 105]

[63] J. L. Overbey and N. Chen. Photran 6.0 Developer's Guide, december 2009. [cited at p. 92, 99]

[64] J. L. Overbey and N. Chen. Photran 6.0 Developer's Guide, december 2009. [cited at p. 105, 106, 142, 187]

[65] J. L. Overbey and N. Chen. Photran 6.0 Developer's Guide, december 2009. [cited at p. 105, 108, 113]

[66] J. L. Overbey, S. Xanthos, R. Johnson, and B. Foote. Refactorings for Fortran and High-Performance Computing. In *SE-HPCS '05: Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 37–39, New York, NY, USA, 2005. ACM. [cited at p. 9, 14]

[67] J.L. Overbey, S. Negara, and R.E. Johnson. Refactoring and the Evolution of Fortran. In *2nd International Workshop on Software Engineering for Computational Science and Engineering (SECSE'09)*, 2009. [cited at p. 9, 14, 52, 141]

[68] D. Pigott. An interactive historical roster of computer languages. *hopl. murdoch. edu. au, last visited: March*, 2005. [cited at p. 4]

[69] J. Reid. The new features of Fortran 2003. In *ACM SIGPLAN Fortran Forum*, volume 26, page 33. ACM, 2007. [cited at p. 48]

[70] J. Reid. The new features of Fortran 2008. In *ACM SIGPLAN Fortran Forum*, volume 27, pages 8–21. ACM, 2008. [cited at p. 49]

[71] Don Roberts and John Brant. Refactoring browser. `http://st-www.cs.illinois.edu/users/brant/Refactory/`, 1999. [cited at p. 20]

[72] Diego Luis Rodrigues. Optimizacin de Software Mediante BLAS Aplicado a un Modelo Climtico. 2008. [cited at p. 11, 177]

[73] T. SCHORSCH. Increasing the readability and comprehensibility of programs(M. S. Thesis). 1990. [cited at p. 163]

[74] Fernando G. Tinetti, Pedro G. Cajaraville, Juan C. Labraga, Mónica A. López, and G.Olguín María. Reverse Engineering Applied to Numerical Software: Climate Models (in Spanish). *X Workshop de Investigadores en Ciencias de la Computación*, pages 434–438, 2008. http://hpclinalg.webs.com/hpclinalg_en.html. [cited at p. 9, 10, 14, 52]

[75] Fernando G. Tinetti, Mónica A. López, and Pedro G. Cajaraville. Fortran Legacy Code Performance Optimization: Sequential and Parallel Processing with OpenMP. *World Congress on Computer Science and Information Engineering*, pages 471–475, 2009. [cited at p. 8, 52, 95, 141]

[76] Edward Yourdon. *Techniques of Program Structure and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1986. [cited at p. 17]

[77] C. Yung. Simplified Readability Metrics. [cited at p. 163]

List of Figures

2.1 Microsoft Visual Studio 18
2.2 Eclipse 19
2.3 Refactoring Browser's hierarchy view. 21
2.4 Refactoring Browser's normal view 21
2.5 The Refactoring Browser's navigator 22
2.6 XRefactory screenshot 23
2.7 Visual Assist rename refactoring 28
2.8 Visual Assist document method refactoring 28
2.9 Refactor! extract method screenshot 30
2.10 Wrangler, an Erlang refactoring tool 31

3.1 IBM 7090 FORTRAN II code example extracted from [40]. 38

5.1 Photran, Fortran View 104
5.2 Photran Architecture [64] 106
5.3 An example of Photran AST 109

6.1 Photran refactoring class diagram 112
6.2 Photran replace character star refactoring class diagram 115
6.3 Code before applying Transform CHARACTER* to CHARACTER(LEN =) refactoring.122
6.4 Transform CHARACTER* to CHARACTER(LEN =) Diff-view. 123
6.5 Transform CHARACTER* to CHARACTER(LEN =) after the refactoring was applied.124
6.6 Photran Standardize I/O Refactoring Class Diagram 127

6.7 InputOutputStatement Visitor Class Diagram 128
6.8 Fortran source code before applying the refactoring 138
6.9 The Diff-view of Standardize IO Refactoring 139
6.10 Fortran source code after applying the refactoring 140
6.11 Old-Style Fortran Do Loops 141
6.12 New-Style Fortran Do Loops 141
6.13 Photran Replace Old Style Do Loops Refactoring Class Diagram . . . 143
6.14 AST Node Rewriting 146
6.15 ASTDoConstructNode Class Diagram 147
6.16 ASTProperLoopConstructNode Class Diagram 148
6.17 Old style do loop source code 153
6.18 Replace Old Style Do Loop Diff-view 154
6.19 The source code refactored 155
6.20 Photran Remove Unreferenced Labels Refactoring Class Diagram . . . 157
6.21 Fortran Program example 159
6.22 Fortran source code with unreferenced labels 161
6.23 Diff-view of Remove Unreferenced Labels Refactoring 162

7.1 Fortran source code after Replace Old Style Do Loop refactoring. . . 170
7.2 Fortran source code after Replace Obsolete Operators refactoring . . 170
7.3 Fortran source code after Standardize Input Output 172
7.4 Fortran source code after Replace Old Style Do Loop refactoring. . . 173