UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

INCORPORAÇÃO DE NOVAS REFATORAÇÕES PARA LINGUAGEM FORTRAN NO IDE ECLIPSE

TRABALHO DE GRADUAÇÃO

Gustavo Rissetti

Santa Maria, RS, Brasil

2010

INCORPORAÇÃO DE NOVAS REFATORAÇÕES PARA LINGUAGEM FORTRAN NO IDE ECLIPSE

por

Gustavo Rissetti

Trabalho de Graduação apresentado ao Curso de Ciência da Computação
da Universidade Federal de Santa Maria (UFSM, RS), como requisito
parcial para a obtenção do grau de
Bacharel em Ciência da Computação

Orientador: Profª Drª Andrea Schwertner Charão

Trabalho de Graduação N. 290

Santa Maria, RS, Brasil

2010

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

INCORPORAÇÃO DE NOVAS REFATORAÇÕES PARA LINGUAGEM FORTRAN NO IDE ECLIPSE

elaborado por
Gustavo Rissetti

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Profª Drª Andrea Schwertner Charão
(Presidente/Orientador)

Profª Drª Iara Augustin (UFSM)

Prof. Dr. Giovani Rubert Librelotto (UFSM)

Prof. Dr. Eduardo Kessler Piveta (UFSM)

Santa Maria, 04 de Janeiro de 2010.

“O maior prazer de um homem inteligente é bancar o idiota diante de um idiota que banca o inteligente.”
— CONFÚCIO

“A mudança é a lei da vida. E aqueles que apenas olham para o passado ou para o presente irão com certeza perder o futuro.”
— JOHN LENNON

“Não sabendo que era impossível, ele foi lá e fez.”
— AUTOR DESCONHECIDO

AGRADECIMENTOS

Gostaria de agradecer primeiramente a Deus, por estar ao meu lado sempre que precisei, não deixando-me na mão até mesmo quando acreditei estar sozinho.

Aos meus pais e minha irmã, que com muita paciência sempre me apoiaram, vivenciando cada preocupação, cada momento de tristeza e alegria, sempre acreditando no meu potencial.

Aos meus colegas de aula, de PET, de curso, em especial ao Vitor Conrado, Vinícius Vielmo Cogo e Rodrigo Exterckötter Tjäder. Nunca esquecerei o apoio, a amizade eterna que obtive com alguns e a força nos trabalhos e nas provas.

À professora Andrea, sempre disposta a ouvir, passando dicas valiosas para o bom desenrolar do trabalho, e sem a qual nada disso seria possível.

E aos demais amigos, parentes e conhecidos que de uma forma ou de outra ajudaram no decorrer dessa jornada, muito obrigado!

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

INCORPORAÇÃO DE NOVAS REFATORAÇÕES PARA LINGUAGEM FORTRAN NO IDE ECLIPSE

Autor: Gustavo Rissetti
Orientador: Profª Drª Andrea Schwertner Charão
Local e data da defesa: Santa Maria, 04 de Janeiro de 2010.

Refatoração é uma técnica de engenharia de *software* que objetiva aplicar mudanças internas no código fonte de aplicações, sem que isso influencie em suas funcionalidades e resultados. Essa técnica não está restrita apenas ao código fonte, podendo ser aplicada também em diversos outros componentes de um sistema de *software*, como no projeto da aplicação, modelos de análise, bancos de dados, dentre outros. A refatoração é uma tarefa permanentemente presente no ciclo de vida do *software*, e com ela busca-se melhorar aspectos não funcionais das aplicações, como legibilidade e possibilidade de reutilização, podendo também ser conseguido algum ganho de desempenho na execução do *software* refatorado. Técnicas de refatoração são amplamente utilizadas em sistemas desenvolvidos para o paradigma da orientação a objetos e estão presentes de forma automatizada em diversas ferramentas que atuam neste paradigma. Na computação científica, na qual existem grandes quantidades de código legado, escritos em linguagens anteriores ao paradigma da orientação a objetos, a refatoração é pouco explorada, principalmente pelo fato de esses códigos serem escritos em uma linguagem pouco comercializada atualmente. A linguagem Fortran (*FORmula TRANslation*) normalmente é utilizada em aplicações de cunho científico, porém, apresenta uma grande carência de ferramentas para a refatoração de código. Neste contexto, este trabalho explora essa deficiência através da automatização de refatorações, utilizando-se do *framework* da ferramenta Photran (um *plugin* para edição de código Fortran integrado ao IDE Eclipse). Partindo-se da identificação de problemas em aberto referentes a ações de refatoração para códigos Fortran, algumas técnicas são desenvolvidas e integradas à ferramenta Photran. As técnicas automatizadas são utilizadas em aplicações escritas nesta linguagem, de forma a avaliar seu funcionamento e validá-las, para que elas possam ser utilizadas em qualquer aplicação escrita em linguagem Fortran pela comunidade de usuários do IDE Eclipse.

Palavras-chave: Refatoração; Fortran.

ABSTRACT

Trabalho de Graduação
Undergraduate Program in Computer Science
Universidade Federal de Santa Maria

INCORPORATION OF NEW REFACTORINGS FOR FORTRAN LANGUAGE IN ECLIPSE IDE

Author: Gustavo Rissetti
Advisor: Prof[a] Dr[a] Andrea Schwertner Charão

Refactoring is a software engineering technique that aims to perform internal changes in the application source code, without influence on its functionality and results. This technique is not restricted only to source code and can be applied also in several other components of a software system, such as application design, analysis models, databases, among others. Refactoring is a permanently present task in the life cycle of software, and it seeks to improve non-functional aspects of applications, such as readability and reusability of code, including a possible gain in performance on the refactored software. Refactoring techniques are widely used in systems developed for the object orientation paradigm and are present in a number of automated tools that work in this paradigm. In scientific computing, in which large amounts of legacy code written in languages before the object-oriented programming paradigm, refactoring is not enough explored, mainly because these codes are written in a little commercialized language today. The Fortran language (FORmula TRANslation) is usually used in scientific applications, but lacks tools for code refactoring. In this context, we explore this deficiency by automating refactorings, using the Photran framework (a plugin for editing Fortran code integrated into the Eclipse IDE). Some techniques are developed and integrated in Photran, based on the identification of open issues concerning actions of refactoring to Fortran code. The automated techniques are used in applications written in Fortran, to assess its operation and validate them to be used in applications written in Fortran by the user community of Eclipse IDE.

Keywords: Refactoring, Fortran.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

CPTEC	Centro de Previsão de Tempo e Estudos Climáticos
BRAMS	Brazilian Regional Atmospheric Modeling System
INPE	Instituto Nacional de Pesquisas Espaciais
ANSI	American National Standards Institute
IDE	Integrated Development Environment
UFSM	Universidade Federal de Santa Maria
ASA	American Standards Association
UML	Unified Modeling Language
CDT	C/C++ Development Tools
GCC	GNU Compiler Collection
EPL	Eclipse Public Licence
VPG	Virtual Program Graph
AST	Abstract Syntax Tree

SUMÁRIO

1 INTRODUÇÃO

Quando se considera o histórico das Tecnologias da Informação hoje utilizadas, nos deparamos com códigos empregados em grandes centros que, tipicamente, têm décadas de desenvolvimento, sendo compostos por centenas de milhares de linhas de código e programados por diversas equipes diferentes, cada uma utilizando técnicas de programação e costumes diferentes. Um efeito direto deste histórico é o uso de uma linguagem de programação antiga. Essas linguagens antigas possuem alguns estilos e técnicas de programação que, atualmente, são consideradas perigosas, dificultando a leitura de código e a identificação de possíveis erros de programação, assim como o processo de manutenção do código, que é fundamental durante todo o ciclo de vida do *software*. Muitas vezes, a opção de reprogramar todo o código é descartada, devido ao custo dessa recodificação ser muito elevado. Assim, uma maneira de contornar e melhorar tal problema, em geral, consiste em utilizar técnicas de reengenharia de *software*, incluindo técnicas de refatoração de código fonte.

Refatoração é uma técnica de engenharia de *software* que visa modificar um sistema para melhorar a estrutura interna do seu código, sem alterar o seu comportamento externo (resultados, funcionalidades e saídas). Essa técnica está permanentemente presente no ciclo de vida de uma aplicação e vem sendo amplamente utilizada nas linguagens de orientação a objetos mais recentes, como Java (FOWLER, 2004). Atualmente, um vasto número de refatorações podem ser encontradas catalogadas, porém, a maioria dessas destinam-se a linguagens mais recentes, voltadas ao paradigma da orientação a objetos, havendo uma carência de refatorações para linguagens mais antigas, como Fortran, por exemplo (FOWLER, 2009).

Fortran é uma linguagem de programação voltada para aplicações científicas, desenvolvida a partir da década de 1950 e que continua a ser usada. O nome tem como origem

a expressão *FORmula TRANslation* ou *Translator*, sendo que essa linguagem é usada principalmente em Ciência da Computação e em Cálculo Numérico. Diversos programas utilizados para aplicações de larga escala, que visam obter um bom desempenho, foram escritos na linguagem Fortran.

Devido à carência de recursos para refatoração em linguagens mais antigas, nesse Trabalho de Conclusão de Curso optou-se por estudar possibilidades de refatoração de programas escritos em Fortran, para melhorar a qualidade do código existente de aplicativos largamente utilizados com uma maior facilidade, sem depender de muito trabalho manual. Até então, a grande maioria das refatorações voltadas para tal linguagem eram feitas manualmente, sem um processo automatizado. Este trabalho visa dar continuidade a um trabalho desenvolvido junto ao Mestrado em Computação do Programa de Pós-Graduação em Informática da UFSM, que também tinha o objetivo de gerar processos automatizados de refatoração para a linguagem Fortran, utilizando o IDE Eclipse e o *plugin* Photran (BONIATI, 2009). Em seu trabalho (BONIATI, 2009; BONIATI; CHARAO; STEIN, 2009), Bruno Batista Boniati explorou técnicas de refatoração para programas Fortran de alto desempenho, desenvolvendo quatro técnicas de refatoração para Fortran, com o objetivo de fazer a evolução da linguagem, melhorar o projeto de código e explorar construções de código que oferecem melhor desempenho. As técnicas de refatoração implementadas em seu trabalho foram nomeadas, no Photran, como *Replace Obsolete Operators* (integrada oficialmente ao Photran), *Introduce INTENT*, *Extract Subroutine* e *Loop Unrolling*.

Neste trabalho são desenvolvidas três novas refatorações para Fortran, dando continuidade ao trabalho iniciado por Bruno Batista Boniati. Os objetivos principais das refatorações implementadas aqui consistem em melhorar a legibilidade do código e gerar possíveis ganhos de desempenho na execução da aplicação refatorada.

1.1 Contextualização e Justificativas

O processo de evolução é uma propriedade natural e que sempre está presente no processo de desenvolvimento de *software*. Normalmente, espera-se que um determinado aplicativo de *software* seja gerado a partir de um bom projeto, com planejamentos, estudos de caso, e estruturas necessárias para atingir a meta do mesmo, sendo posteriormente codificado. Durante o ciclo de vida, normalmente, é necessário que o aplicativo passe

por alguma evolução, seja para adicionar um novo requisito ou para alterar alguma das funcionalidades existentes.

Porém, em alguns casos, pode ocorrer que a aplicação não esteja preparada para receber novos requisitos ou suportar adaptações em suas funcionalidades, e, dependendo da forma como as alterações são feitas, elas podem acabar degradando a qualidade e enfraquecendo a relação da aplicação com o objetivo descrito no projeto original.

Ao longo do processo de desenvolvimento ou manutenção de *software*, é comum que mudanças em requisitos favoreçam a presença de código mal escrito, na medida em que os programadores têm de adaptar o código às novas funcionalidades, muitas vezes usando técnicas de programação consideradas inadequadas. Com a introdução de códigos ilegíveis é dificultada mais ainda a manutenção do *software*, que passa a ser incompreensível por grande parte dos programadores. Uma medida a ser tomada para melhorar a qualidade desse código é a reengenharia de código fonte e o uso de técnicas de refatoração, que podem fazer com que o código fique otimizado e com legibilidade melhorada.

O objetivo principal do uso da refatoração é melhorar atributos de qualidade da aplicação tais como legibilidade, simplicidade, organização e desempenho. Embora possam ser aplicadas manualmente sobre o código fonte, a facilidade no uso de refatorações se dá quando as técnicas de refatoração são automatizadas por meio de ferramentas que atuam sobre o código fonte previamente escrito (ROBERTS, 1999). Neste caso, pode haver a necessidade de se utilizar ferramentas específicas para a linguagem de programação em que a aplicação foi escrita.

Linguagens estruturadas, mais antigas, e pouco difundidas comercialmente na atualidade, possuem poucas e raras ferramentas, como é o caso do Fortran, caracterizada por ser uma linguagem de programação amplamente utilizada no meio científico e pioneira na adoção do paradigma imperativo (NYHOFF; LEESTMA, 1997). A grande quantidade de códigos Fortran legados que permanecem sendo utilizados passam por processos manuais de refatoração, pois existem poucas ferramentas que auxiliam o trabalho com essa linguagem, o que justifica o quão necessário é o desenvolvimento de técnicas automatizadas para a refatoração de códigos fonte em linguagem Fortran.

A lacuna existente entre a quantidade de código legado escrito em Fortran (em especial de aplicações científicas que exigem alto desempenho) e as raras técnicas e ferramentas de refatoração existentes para esta linguagem, servem de motivação para a realização deste

trabalho de pesquisa, pois visa-se obter uma melhora no aspecto relacionado às facilidades de refatoração dos aplicativos escritos em Fortran.

Um código fonte pode ser melhorado através de sucessivas refatorações, sem perder sua funcionalidade. Porém, para que isso seja possível, é necessário que a linguagem sendo trabalhada possua ferramentas de auxílio que ofereçam mecanismos automatizados de refatoração, para que essa possa ser aplicada em larga escala e possa ser aplicada em sistemas de *software* de grande porte, o que acontece na maioria dos casos.

Na literatura é possível encontrar diveras catalogações de refatorações independentes de linguagem, descritas muitas vezes em linguagem natural. A catalogação de técnicas de refatoração é uma atividade que pode ser desenvolvida de forma genérica, não associada a uma linguagem ou a um paradigma específico. Assim, é possível descrever os objetivos, pré-requisitos, limitações e a própria mecânica de funcionamento de uma técnica de refatoração usando-se a linguagem natural (DE, 2004).

A utilização de ferramentas com automatizações voltadas à refatoração garantem uma redução do risco de erros e inconsistências, além de reduzir também o trabalho e o custo de desenvolvimento de *software* (FOWLER, 2004). A refatoração automatizada por ferramentas torna a ação de refatorar cada vez menos separada da atividade de programar. Refatorar manualmente é uma atividade cara, que, por vezes, justificada pelo seu alto custo, não é feita. O objetivo de automatizar refatorações é reduzir o custo do processo de reestruturação de código.

1.2 Objetivos e Contribuições

As principais contribuições deste trabalho são a identificação e automatização de técnicas de refatoração para linguagem Fortran na ferramenta Photran (um *plugin* para edição de código Fortran do IDE Eclipse). O Photran é um projeto mantido pela *Eclipse Foundation* sob licença EPL (*Eclipse Public License*) e desenvolvido, inicialmente, por um grupo de pesquisadores ligados à Universidade de Illinois em Urbana-Champaign (EUA) e ao Laboratório Nacional de Los Alamos (EUA) (ECLIPSE.ORG, 2009a,b).

Foram automatizadas e integradas à ferramenta Photran três técnicas: Padronização na Declaração de Variáveis (*Standardize Statements*), Remoção de Variáveis não Utilizadas (*Remove Unused Variables*) e Transformação de Declarações do Tipo *Data* em Declarações do Tipo *Parameter* (*Data To Parameter*). Também foi feita uma atualização da

refatoração *Introduce INTENT* (BONIATI, 2009), para que funcionasse na nova versão do Photran.

Para validar e avaliar o impacto da utilização das técnicas de refatorações automatizadas desenvolvidas nesse trabalho, foram usados dois programas de cunho científico, que estão contidos no projeto *org.eclipse.photran-samples*, distribuído juntamente com o Photran, sendo desenvolvidos na linguagem Fotran 90. Também foi utilizada uma aplicação cedida pelo Laboratório de Micrometeorologia vinculado à Universidade Federal de Santa Maria, desenvolvida na linguagem Fortran 77, que é usada intensamente no laboratório para fazer análises micrometeorológicas.

1.3 Organização do Texto

Este trabalho está organizado da seguinte maneira: o Capítulo 2 apresenta uma revisão bibliográfica do tema central do trabalho, incluindo alguns requisitos necessários para que uma técnica de refatoração seja automatizada, apresentando também os recursos e características do Photran.

O Capítulo 3 detalha as características de implementação de algumas técnicas de refatoração que foram identificadas durante a realização do trabalho. Para cada técnica descreve-se sua motivação, sua mecânica, assim como sua implementação.

No Capítulo 4 são realizados estudos de caso com a utilização das técnicas de refatoração automatizadas em aplicações reais. Nesse capítulo são identificadas oportunidades de refatoração do código fonte dessas aplicações e são avaliados os benefícios detectados no uso das técnicas. Alguns trabalhos relacionados com o assunto de refatoração em Fortran e em outras linguagens de programação, assim como em outros componentes de um sistema de *software*, também podem ser observados nesse capítulo. Por fim, no Capítulo 5, apresentam-se as considerações finais do trabalho e, discutem-se algumas sugestões para a sua continuidade.

2 FUNDAMENTAÇÃO

Neste capítulo são descritos alguns conceitos que formaram o embasamento teórico necessário para o desenvolvimento deste trabalho.

2.1 Refatoração

Refatoração pode ser definida como uma alteração feita na estrutura interna de um *software* para torná-lo mais fácil de ser entendido e menos custoso de ser modificado, sem alterar o seu funcionamento aparente. Refatorar significa, portanto, reestruturar um *software*, aplicando-lhe uma série de modificações sem alterar o seu comportamento observável, como funcionalidades, entradas, saídas e resultados.

O termo refatorar faz parte de um domínio de pesquisa mais amplo, relacionado à reestruturação de *software* (GRISWOLD; NOTKIN, 1993). Comumente, é empregado para caracterizar reestruturações realizadas em *software* desenvolvido sob o paradigma da orientação a objetos. A idéia chave é redistribuir classes, variáveis e métodos através da hierarquia (estrutura) de *software*, de forma a facilitar futuras adaptações e extensões (MENS; TOURWé, 2004). Em geral, são alterações simples que atuam sobre características não funcionais de *software*, como por exemplo, extensibilidade, modularidade, reusabilidade, complexidade e eficiência.

Espera-se que uma ferramenta destinada a fazer refatorações automatizadas esteja integrada a ambientes de desenvolvimento de *software*. Uma ferramenta para automatização de técnicas de refatoração precisa contemplar pelo menos os seguintes requisitos (ROBERTS; BRANT; JOHNSON, 1996):

- **Representação abstrata do programa:** a ferramenta deve oferecer a representação abstrata do programa por meio de uma árvore sintática;
- **Banco de dados do programa:** é desejável que a ferramenta guarde de forma atu-

alizada um conjunto de informações ligadas à representação abstrata do programa;

- **Corretude:** ao final da refatoração a aplicação deve preservar o mesmo comportamento que tinha antes de sofrer a refatoração (CORNéLIO, 2004). A ferramenta deve dispor de recursos para minimamente detectar a introdução de um erro (e permitir desfazer as alterações).

Uma aplicação que esteja funcionando corretamente também pode estar necessitando de reestruturações internas, pois pode conter falhas de projeto. Se o projeto é falho e o código é ruim e mal estruturado, então mesmo que a aplicação esteja atendendo corretamente aos usuários, ela poderia ser refatorada e o projeto seria corrigido. Nesse sentido, a refatoração insiste em atributos de qualidade de *software* que são menos tangíveis, mas decisivos, como projeto, simplicidade, melhoria na compreensão do código fonte, legibilidade, dentre outras.

Algumas das características que podem ser citadas como indicadores de necessidade de refatoração são: métodos ou classes muito extensos, métodos com excessivos parâmetros, falta de clareza ou legibilidade do código. Também pode-se descrever algumas situações para as quais é recomendável a utilização de refatoração (FOWLER, 2004):

- **Adição de um novo recurso:** quando um novo recurso precisa ser adicionado ao *software*, primeiramente ele deve passar por uma preparação, que também pode ser feita através de outras refatorações, como reorganização do código, extração de subrotinas, etc.

- **Correção de um erro (*bug*):** quando um erro é conhecido, na medida em que se pode utilizar um algoritmo que contenha a correção, pode-se, muitas vezes, simplificar esse algoritmo, diminuindo o tamanho do código total e também diminuindo o custo de processamento de tal algoritmo.

- **Revisão do código:** quando o código base é examinado, uma das metas deve ser simplificar a estrutura do código e os algoritmos no contexto da aplicação como um todo.

As técnicas de refatoração são geralmente simples e, quando aplicadas isoladamente, produzem pequenas modificações notáveis no código. Uma boa melhoria em um *software* se dá realmente quando diversas técnicas de refatoração são aplicadas, pois a aplicação

dessas técnicas é cumulativa, e juntando-se um conjunto de pequenas melhorias, obtém-se no final uma grande melhoria, que resulta em um código melhor estruturado e mais legível.

Também é possível que existam inter-relações entre técnicas de refatoração, por exemplo: para que determinada técnica seja aplicada é pré-requisito que mais alguma ou algumas técnicas sejam previamente aplicadas. Como exemplo de refatorações sucessivas, pode-se citar o caso da refatoração *Introduce INTENT*, que tem como pré-requisito a aplicação da refatoração *Introduce Implicit None* (BONIATI, 2009).

Uma importante recomendação que se deve considerar antes da refatoração de *software* é a existência de um conjunto de testes, preferencialmente automatizados (DEURSEN et al., 2001). A execução de um teste automatizado antes e depois de se aplicar uma refatoração poderá indicar se houve ou não a adição de alguma falha ao código refatorado.

Embora o termo refatoração tenha origem na orientação a objetos, o seu conceito pode ser usado para o paradigma estruturado, contendo algumas limitações. Em linguagens desse paradigma, os fluxos de controle e de dados são fortemente interligados, dificultando assim a manipulação do código para ser refatorado (DE, 2004). Em geral, reestruturações em códigos não orientados a objeto são limitadas no nível de subprograma, bloco de código ou entidades (variáveis, por exemplo). Existem, contudo, técnicas e ferramentas que atuam sobre paradigmas não orientados a objetos, como é o caso de um *plugin* do Eclipse que atua sobre código fonte Fortran, o Photran, que é utilizado no desenvolvimento deste trabalho.

2.2 Refatoração *versus* Otimização de Código

Muitas vezes o termo refatoração é confundido com a otimização de código, que é feita diretamente pelo compilador. A refatoração é um processo semi-automático, que normalmente necessita da intervenção do usuário para obter informações necessárias a sua execução, como o nome de um novo método, por exemplo. As refatorações são feitas normalmente através do uso de IDEs de programação que oferecem esse recurso de forma automatizada (semi-automáticas), através de menus, para facilitar o seu uso ao programador.

Já a otimização de código acontece na etapa final na geração de código pelo compilador, de forma totalmente automatizada, sem a intervenção do usuário. Como o código

gerado através da tradução orientada a sintaxe contempla expressões independentes, diversas situações contendo sequências de código ineficientes podem ocorrer. O objetivo principal da otimização de código consiste em aplicar um conjunto de heurísticas para detectar tais sequências e substituí-las por outras que removam as situações de ineficiência, normalmente resultando em melhora de desempenho da execução da aplicação (RICARTE, 2008).

As técnicas de otimização que são usadas nos compiladores, assim como as refatorações, devem manter as características do programa original. Também devem ser capazes de capturar a maior parte das possibilidades de melhoria do código dentro dos limites razoáveis do grau de otimização escolhido. Em geral, os compiladores permitem especificar qual o grau de otimização desejado no processo de otimização. Por exemplo, no compilador GCC (*GNU Compiler Collection*) (GNU, 1999) há opções na forma *-O0*, *-O1*, *-O2*, *-O3* e *-Os*, sendo que a opção *-O0* significa nenhuma otimização e a opção *-O3* significa a máxima otimização (aumentando o tempo de compilação). A opção *-Os* indica que o objetivo da otimização é reduzir a ocupação de espaço em memória.

2.3 Eclipse IDE

O Eclipse (ECLIPSE.ORG, 2009a) é um IDE de desenvolvimento que tem suporte para diversas linguagens de programação, como C, C++, Java, e recentemente Fortran, com o uso do *plugin* Photran, que é visto na seção 2.5.

Para a programação em Fortran, existe o *plugin* Photran, que é desenvolvido sobre o *plugin* CDT (*C/C++ Development Tools*) (ECLIPSE.ORG, 2009c), utilizando a maioria dos recursos já disponíves nele, como interfaces gráficas, interação com o usuário e gerenciamento de projetos. Porém, o menu de refatoração para projetos Fortran possui poucas alternativas, e as refatorações disponíveis por padrão no Photran são bastante simples.

O motivo da escolha do Eclipse IDE e o *plugin* Photran como ferramentas para o desenvolvimento deste trabalho é justificado pela intenção de dar continuidade a um trabalho já iniciado durante o mestrado de Bruno Batista Boniati (BONIATI, 2009). Bruno Batista Boniati usou as mesmas ferramentas em seu trabalho, pois as mesmas possuem a vantagem de ser *software* livre, permitindo incluir modificações, e possuem um *framework* pronto para receber novas funcionalidades, como refatorações automatizadas para Fortran, que é o alvo deste trabalho.

2.4 Fortran

Com o advento do Fortran, foi possibilitado que programas fossem escritos mais rapidamente que quando se trabalhava apenas com Assembler, por exemplo, com somente uma pequena perda de eficiência no processamento, uma vez que todo cuidado era dedicado na construção do compilador.

Pode-se dizer também que o Fortran foi determinante na evolução da programação, na medida em que os programadores substituíram o Assembler por uma linguagem de programação de mais alto nível, com alto desempenho, assim, concentrando-se mais na solução do problema em questão. A tarefa da programação não estava mais restrita a um pequeno número de programadores especialistas.

O Fortran disseminou-se rapidamente, principalmente nas áreas da física, engenharia e matemática, uma vez que satisfazia a necessidade de os cientistas programarem em uma linguagem de programação de alto desempenho. Inevitavelmente, dialetos da linguagem foram desenvolvidos, os quais levaram a problemas quando havia necessidade de se trocar programas entre diferentes computadores. Em 1966, após quatro anos de trabalho, a Associação Americana de Padrões (*American Standards Association*, ASA), posteriormente Instituto Americano Nacional de Padrões (*American National Standards Institute*, ANSI) originou o primeiro padrão para uma linguagem de programação, o Fortran 66 (ou Fortran IV). Essencialmente, era um subconjunto comum de vários dialetos, de tal forma que cada dialeto poderia ser reconhecido como uma extensão do padrão. Aqueles usuários que desejassem escrever programas portáveis deveriam evitar as extensões e restringir-se ao padrão.

O Fortran é uma linguagem de programação imperativa, baseada em um controle sequencial do fluxo de execução composto por construções de desvio como procedimentos e funções. As versões atuais da linguagem permitem várias características da programação orientada a objetos.

Há um grande legado de aplicações científicas escritas em Fortran. Geralmente, são aplicações que durante anos sofreram estudos e otimizações, e que executam tarefas especializadas, como análises climáticas, por exemplo. O BRAMS (*Brazilian Regional Atmospheric Modeling System*), aplicativo para análise de modelos climáticos, mantido pelo CPTEC (Centro de Previsão de Tempo e Estudos Climáticos) do INPE (Instituto Nacional de Pesquisas Espaciais), é um exemplo de aplicação de alto desempenho escrita

em linguagem Fortran.

Desempenho é a palavra-chave que explica a perpetuação histórica de Fortran. A linguagem sofreu várias revisões ao longo do tempo, ganhou bibliotecas ricas e otimizadas. Uma das suas principais vantagens, considerando a computação de alto desempenho (muito utilizada em aplicações de cunho científico), é a existência de operações para tratamento de dados multidimensionais que normalmente não são encontradas de forma nativa em outras linguagens de programação (KOFFMAN; FRIEDMAN, 1996). Pelo fato de o domínio da aplicação do Fortran ser bastante específico e ligado à pesquisa científica e não ao mercado, há uma lacuna entre a quantidade de código legado escrito em Fortran e ferramentas que oferecem técnicas automatizadas de refatoração existentes para essa linguagem.

2.5 Photran

Photran (ECLIPSE.ORG, 2009b) é um *plugin* desenvolvido para o IDE Eclipse para dar suporte à programação em linguagem Fortran, com ênfase na refatoração automatizada de códigos Fortran (Figura 2.1) (CHEN; OVERBEY, 2008). O Photran é um IDE para Fortran 77, 90, 95 e 2003, sendo uma extensão do CDT (*C/C++ Development Tools*).

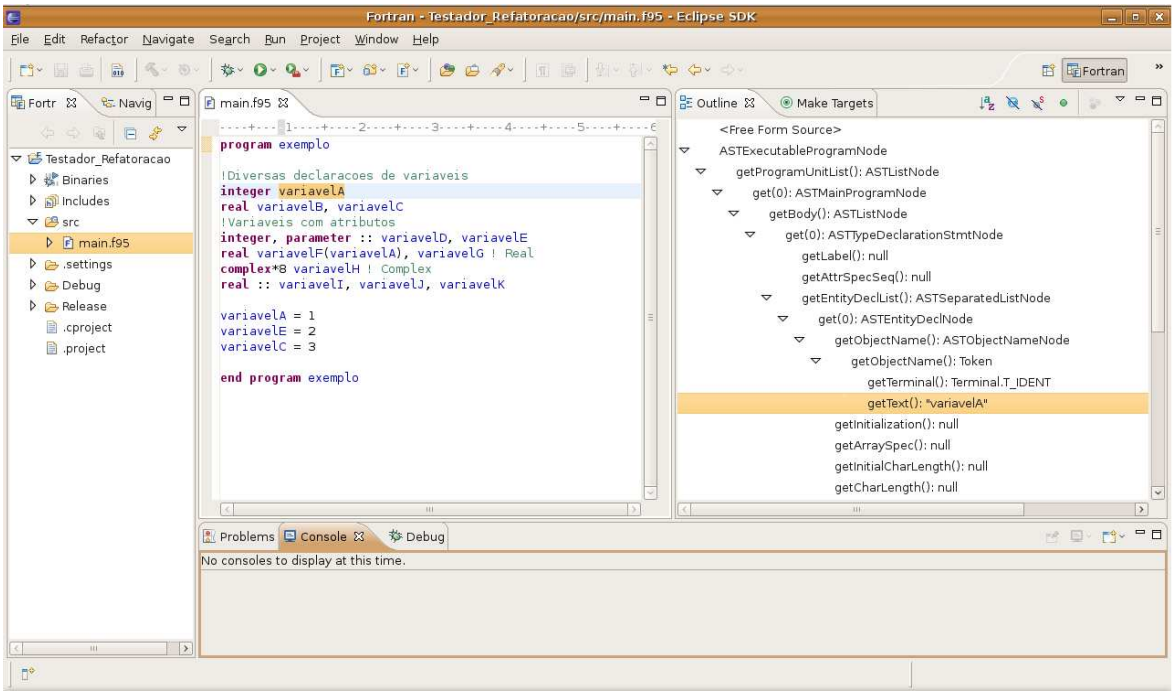

Figura 2.1: Visualização do IDE Photran em execução

Assim como outras ferramentas que são extensões do Eclipse, o Photran é desenvol-

vido em linguagem Java e compõe-se de *plugins* e recursos (*features*). *Plugins* adicionam funcionalidades ao Eclipse. Recursos são unidades de desenvolvimento (diversos *plugins* são empacotados em recursos, que são distribuídos aos usuários). Arquiteturalmente, o projeto Photran divide-se em subprojetos onde cada um se constitui de um *plugin* ou um recurso.

O IDE Photran já possui um analisador sintático completo e uma representação de programa, com o Photran VPG (*Virtual Program Graph*), que gera uma AST (*Abstract Syntax Tree*) do código fonte, permitindo a sua manipulação e edição, além de outras utilidades (OVERBEY, 2009).

No Photran, durante a edição do código fonte, é possível visualizar através do recurso *Outline View* a estrutura hierárquica de elementos que compõem o código fonte. Existindo erros sintáticos, é possível detectá-los durante a escrita do código (uma vez que a árvore sintática fica impossibilitada de ser construída e a *Outline View* mostra uma mensagem de erro).

Além destes e outros recursos, normalmente presentes de uma forma ou de outra em IDEs ou editores de código, Photran oferece uma infraestrutura para refatoração de código fonte. Essa infraestrutura consiste de representações abstratas do programa (que permitem navegar e alterar sua estrutura em alto nível), mecanismos para visualizar e comparar diferenças antes e depois da refatoração e, ainda, meios de cancelar ou desfazer uma refatoração efetuada.

No Photran (versão 4.0.5), as seguintes refatorações são disponibilizadas por padrão na IDE:

- *Rename*: pode ser vista como uma busca e substituição "inteligente", permitindo a alteração do nome de uma variável, subprograma, etc.

- *Introduce Implicit None*: adiciona o comando *IMPLICIT NONE* em um arquivo e acrescenta declarações explícitas para todas as variáveis que foram declaradas implicitamente.

- *Extract Local Variable*: remove uma subexpressão de uma expressão maior e atribui essa subexpressão a uma variável local, substituindo a subexpressão original por uma referência a essa variável. Esta refatoração é geralmente usada para eliminar subexpressões repetidas ou introduzir nomes de variáveis explicativos em

expressões complexas.

- *Extract Procedure*: remove uma seqüência de declarações de um procedimento, coloca-as em uma nova subrotina, e substitui as declarações originais por uma chamada para a nova subrotina. Esta refatoração é usada, geralmente, para diminuir o tamanho dos procedimentos.

- *Move Saved Variables to Common Block*: cria um *COMMON BLOCK* para todas as variáveis do tipo *save* de um subprograma. As declarações dessas variáveis no subprograma não serão mais do tipo *save*.

- *Replace Obsolete Operators*: substitui todas as utilizações dos antigos operadores de comparação (como .LT. e .EQ.) por seus equivalentes mais recentes (como símbolos < e ==), e acrescenta declarações explícitas para todas as variáveis que foram declaradas implicitamente.

2.5.1 Photran: AST e VPG

Uma árvore sintática abstrata (AST) é uma estrutura para representação do código do programa. Ela é composta por uma raiz da qual são derivados vários nós que, por sua vez, podem ser compostos de outros nós. O último nível dessa árvore, normalmente, representa os *tokens* da linguagem, com nós que caracterizam os operadores da linguagem de programação, por exemplo (JONES, 2003). Cada nó é classificado de acordo com seu funcionamento e suas ações. Por exemplo, um nó do tipo declaração derivará outros nós que representarão, por exemplo, o nome da variável declarada e o tipo da mesma, podendo ainda conter um valor para a inicialização da variável.

A construção de uma AST requer a existência de um analisador sintático específico para a linguagem de programação com a qual se deseja trabalhar. Caso o código fonte não possa ser decomposto segundo a estrutura gramatical, o processo de análise sintática acusa um erro de sintaxe.

Um analisador sintático é responsável por validar a estrutura de um código fonte (no caso deste trabalho, escrito em linguagem Fortran) e gerar uma sequência de derivação, ou seja, uma árvore sintática abstrata (AST). Uma AST é uma estrutura hierárquica na qual seus nós decompõem-se em conjuntos de nós filhos (terminais ou não terminais). Em geral, os nós terminais representam *tokens* (comandos, expressões, operadores, etc.) do código fonte.

A AST é uma peça chave para a automatização de ações de refatoração, uma vez que ela possibilita ao desenvolvedor navegar pela estrutura do código fonte, detectando correlações entre seus nós e identificando oportunidades de refatoração (OVERBEY; JOHNSON, 2009). Quando é disponibilizada uma estrutura de representação como a AST, a ação de refatoração consiste em introduzir as modificações diretamente na AST, fazendo a inclusão de nós, exclusão de nós ou, ainda, alterando o posicionamento de determinados nós da árvore. O Photran oferece um recurso que possibilita ao programador visualizar em tempo real a AST enquanto está programando. Esse recurso é disponibilizado através da *Outline View*. Na Figura 2.2 pode-se observar um código simples em Fortran e sua AST no editor de texto do IDE Photran.

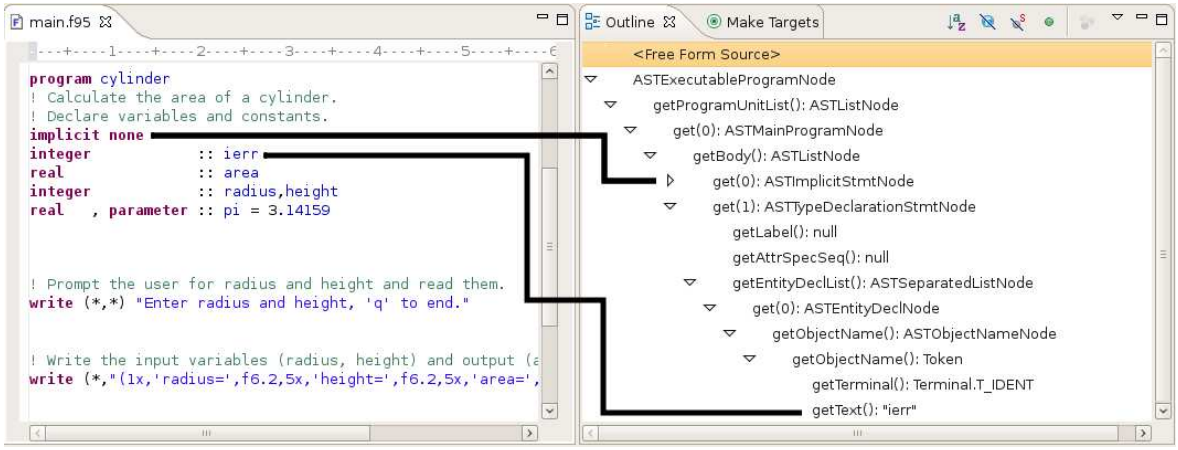

Figura 2.2: Visualização de um trecho de código Fortran e sua AST

O analisador sintático do Photran foi produzido utilizando-se da ferramenta para geração de analisadores sintáticos Ludwig (OVERBEY; JOHNSON, 2008, 2009). O projeto da ferramenta Ludwig originou-se a partir da idéia de produzir analisadores sintáticos e árvores sintáticas abstratas com foco na simplificação e rapidez do processo de automatização de refatorações. Utiliza a notação EBNF (ISO, 1996) para a especificação da gramática e produz o analisador sintático em código Java. Sua principal característica é a geração de árvores sintáticas abstratas que podem ser manipuladas, ou seja, cujos nós podem ser removidos, substituídos ou realocados de lugar, permitindo a partir dessa árvore reconstruir o código fonte original (ou modificado) preservando até mesmo comentários, espaços em branco, quebras de linha e formatações do programador. O Ludwig é parte importante da ferramenta Photran e peça chave na implementação de refatorações.

O analisador sintático do Photran, além de fornecer a AST, providencia o VPG (*Vir-*

tual Program Graph), que é outra estrutura muito importante, uma vez que possibilita a agregação de outras ligações entre os nós da árvore sintática abstrata, não representando necessariamente a hierarquia da árvore. Algumas das ligações adicionais podem ser, por exemplo, um vínculo entre a utilização de determinada variável (em uma expressão) e sua respectiva declaração. É por meio do VPG que se pode obter um conjunto rico de informações para subsidiar a manipulação dos nós de uma AST.

O VPG pode ser visto como um conjunto de arestas que ligam os nós de uma AST de forma não hierarquizada (OVERBEY, 2009). Na Figura 2.3 é possível observar um exemplo de um programa e seu VPG (OVERBEY, 2009). Os nós (elipses) com ligações de linhas contínuas são os nós que compõem a AST. Os nós retangulares representam os *tokens* no código do programa fonte. As ligações extras (em linhas tracejadas), que são rotuladas, transformam a AST em um VPG. Nesse exemplo, o VPG apresenta dois tipos de ligação, a *binding*, que liga referencias de variáveis às suas respectivas declarações e, *scope of declaration*, que liga cada declaração com o nó que representa o seu escopo na AST.

2.5.2 Infraestrutura do Photran

No Photran, para que seja possível automatizar uma refatoração são necessários três passos: fazer uma pré-validação da ação a ser desenvolvida, fazer a manipulação da AST (a fim de realizar as alterações propostas) e fazer uma pós-validação para garantir a integridade da AST. A utilização do *framework* do Eclipse e do *plugin* Photran possibilita atuar sobre o código Fortran por meio da manipulação da AST e da utilização da base de informações existentes no VPG. Existem métodos que possibilitam navegar sobre a AST, recuperar informações acerca de seus nós e *tokens* e ainda executar operações de atualização sobre ela (remoção, adição ou substituição de nós).

Para implementar e integrar uma ação de refatoração ao *framework* do Photran, é necessário estender o comportamento de algumas classes presentes no *framework*. A primeira é responsável por receber a chamada do usuário e associar a ação de refatoração com seu respectivo assistente. Essa classe deve estender a classe *AbstractFortranRefactoringActionDelegate* e deve implementar os métodos de duas intefaces: *IWorkbenchWindowActionDelegate*, permitindo assim, que a chamada do usuário seja feita a partir do menu principal, e *IEditorActionDelegate* que permite ao usuário fazer a chamada a partir

```
program
  integer i = 3
  print i + " is the value of i"
end program
```

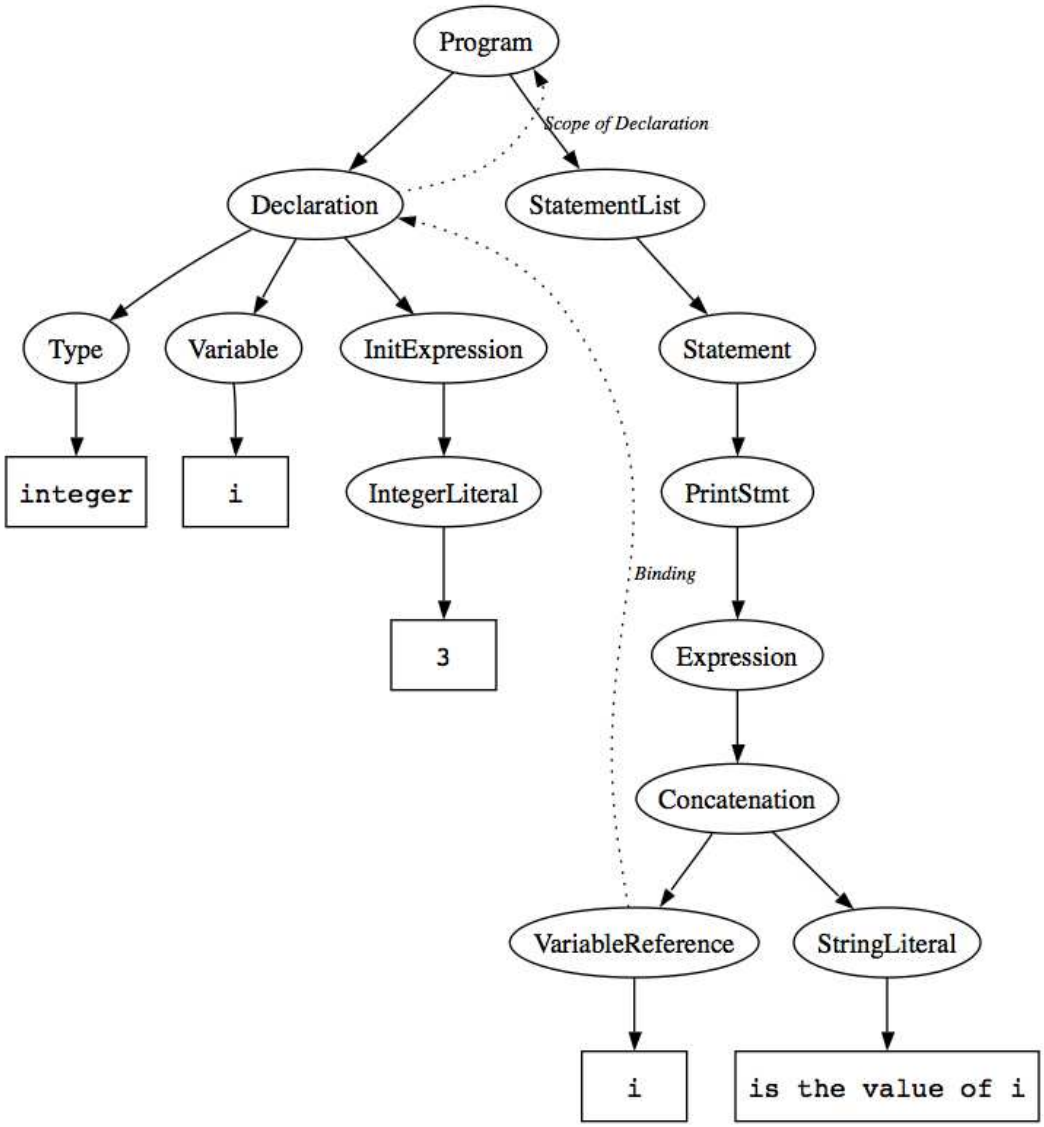

Figura 2.3: Visualização do VPG de um programa (OVERBEY, 2009)

de um menu de contexto no próprio editor.

A segunda classe a ser estendida é o assistente (*wizard*) da refatoração. É comum que uma ação de refatoração solicite ao usuário alguma informação adicional para que possa ser executada, como um novo nome para uma variável, um nome para um novo método, etc. Essa classe é estendida de *AbstractFortranRefactoringWizard* e tem objetivo de fazer

a construção gráfica da janela do assistente. Seu método construtor recebe como parâmetro um objeto para a terceira classe necessária, que faz a ação de refatoração propriamente dita.

A principal classe a ser codificada é a ação da refatoração. Essa classe pode estender a superclasse *SingleFileFortranRefactoring* ou a superclasse *MultipleFileFortranRefactoring*, sendo que ambas são classes específicas do Photran. A primeira se destina a fazer refatorações que são aplicadas a um arquivo por vez, que exigem entradas de dados pelo usuário, como a seleção de um trecho do código fonte a ser refatorado, por exemplo. Já a segunda é destinada às refatorações que podem ser aplicadas a lotes de arquivos, que não exigem entradas de dados pelo usuário, pois, normalmente, faz uma varredura do código fonte fazendo alterações independentes, como reindentação, eliminação de variáveis não usadas, etc. Essas duas classes, por sua vez, estendem a classe *AbstractFortranRefactoring*, que também é específica do Photran, e que estende a classe *Refactoring*, a qual faz parte do *framework* do Eclipse. Nessa classe (ação da refatoração), quatro métodos precisam ser codificados:

- ***getName()*:** esse método é responsável por fornecer o título da refatoração para que possa ser utilizado em janelas e caixas de diálogo do Eclipse;

- ***doCheckInitialConditions()*:** esse método serve para fazer uma pré-validação da AST e dos critérios exigidos pela ação de refatoração usada. O método pode propagar uma exceção da classe *PreconditionFailure* indicando que alguma condição para a ação de refatoração não foi satisfeita e que a mesma não será realizada;

- ***doCheckFinalConditions()*:** esse método é responsável por fazer uma pós-validação, que confirma ou não as alterações aplicadas sobre a AST. Assim como o item anterior, esse método pode propagar uma exceção da classe *PreconditionFailure* indicando que alguma condição para a ação de refatoração não foi satisfeita e que a mesma não será realizada;

- ***doCreateChange()*:** esse método realiza a refatoração propriamente dita, fazendo as modificações na AST, caso a pré-validação tenha sido positiva. Ao término de tais modificações, este método chama dois outros métodos, o *addChangeFromModifiedAST()* que tem o objetivo de adicionar as modificações feitas à AST utilizada

pelo Photran e o método *releaseAllASTs()*, visando forçar o Photran a atualizar os controles visuais da AST assim como sua validação.

Após realizadas as implementações dessas classes, é necessário alterar o arquivo *manifest* (*plugin.xml*) para indicar ao Eclipse que existem novos pontos de extensão que precisam ser disponibilizados.

2.6 Sumário do capítulo

Nesse capítulo foi possível entender melhor o conceito de refatoração, esclarecendo quais são suas vantagens e quais são suas possibilidades e indicações de uso.

Também foram apresentadas as ferramentas utilizadas no desenvolvimento deste trabalho, justificando os motivos de suas escolhas. Foi feita também uma pequena revisão bibliográfica sobre a linguagem de programação Fortran, a qual é um dos motivos do desenvolvimento deste trabalho.

Foram descritas em detalhes algumas características e recursos do *plugin* Photran, descrevendo-se também a sua infraestrutura e os requesitos necessários para a automatização e integração de novas técnicas de refatoração no mesmo.

No Capítulo 3 são detalhadas as características de implementação das refatorações identificadas durante a realização deste trabalho. Para cada técnica descreve-se sua motivação, sua mecânica e sua implementação.

3 REFATORAÇÕES PARA FORTRAN

Neste capítulo são apresentadas as técnicas de refatoração desenvolvidas neste trabalho. As técnicas utilizadas possuem diferentes objetivos, como legibilidade de código (melhores práticas que favorecem a interpretação do código, tornando-o mais simples) e desempenho (reestruturação do código com o objetivo de melhorar o desempenho da aplicação). Nas próximas seções as técnicas desenvolvidas são descritas em detalhes.

Foram desenvolvidas três técnicas de refatoração para código Fortran, com o propósito de melhorar a legibilidade do código e melhorar o desempenho do mesmo, além da atualização de uma refatoração previamente criada, chamada no Photran de *Introduce INTENT* (BONIATI, 2009). O código fonte referente às implementações realizadas neste trabalho pode ser consultado no Apêndice A. Na ferramenta Photran as técnicas desenvolvidas neste trabalho foram nomeadas como:

- *Standardize Statements*;
- *Remove Unused Variables*;
- *Data To Parameter*.

Após concluir a refatoração *Standardize Statements*, vista na seção 3.1, foi dado início à pesquisa de como automatizar uma ação de refatoração capaz de remover variáveis não utilizadas em um código fonte (*Remove Unused Variables*), tratada na seção 3.2. A proposta para tal refatoração, assim como a proposta para a refatoração *Data To Parameter*, vista na seção 3.3, encontram-se em um dos itens do projeto *Fortran Refactoring: FortiFact*, que vem sendo desenvolvido em conjunto entre as instituições UFRGS, UFSM e CPTEC (INPE).

3.1 Padronizar Declarações (*Standardize Statements*)

– Você possui um código programado por diversas pessoas, com costumes diferentes de programação, resultando em diversos tipos de declarações de variáveis, prejudicando a legibilidade do código.

– Essa refatoração faz uma padronização das declarações de variáveis, melhorando a legibilidade do código e normalizando as declarações.

3.1.1 Motivação

Na linguagem Fortran, as variáveis podem ser declaradas de maneiras diferentes, ou seja, podem ser declarações simples, onde apenas uma variável é declarada a cada comando, podendo ou não ter os tradicionais dois pontos (**::**) do Fortran, ou podem ser declaradas listas de variáveis.

Porém, como a linguagem permite tais tipos de declarações de variáveis, muitas vezes códigos que são escritos por grandes equipes passam pelas mãos de diferentes programadores, que têm hábitos diferentes de programar, e usam uma maneira diferente para declaração de variáveis. Essa mistura de gêneros de declarações podem dificultar a leitura de código, uma vez que uma imensa lista de variáveis pode ser declarada, e quando o leitor do código chega ao final de tal lista, já não lembra mais qual era o tipo de declaração das variáveis, por exemplo.

A refatoração, nomeada no Photran de *Standardize Statements* tem o objetivo de fazer uma padronização nas declarações de variáveis.

3.1.2 Mecânica

- Percorrer o código fonte em busca de identificadores de declaração de variáveis.
    - Verificar se cada declaração encontrada é simples ou uma lista de declaração.
    - Se for uma lista de declaração, devem-se criar novas declarações simples a partir dos elementos contidos na lista de declaração.
    - Remover o identificador que contém a lista de declaração.
    - Adicionar os dois pontos (**::**) nas declarações simples.
- Compilar e testar o código, verificando seu correto funcionamento.
- A refatoração está completa.

3.1.3 Implementação

O processo de implementação contou com algumas facilidades providas pela AST. Com o método *ast.getRoot().getAllContainedScopes()* é possível obter todos os escopos da árvore sintática abstrata, e, assim, tem-se liberdade de percorrer e alterar um escopo por vez. Cada escopo é percorrido com um laço de iteração, a fim de achar nós do tipo *ASTTypeDeclarationStmtNode*. Quando um desses nós é encontrado, constrói-se uma lista de variáveis, que é preenchida com as variáveis presentes na lista de declaração do nó (o tamanho da lista pode variar de 1 a N elementos). Posteriormente, através do método *getTypeSpec()* é conseguido ter acesso ao tipo da declaração da(s) variáveis do nó, para ser usado na criação dos nós padronizados.

Após preencher a lista de variáveis e ter posse do tipo da declaração, são criados novos nós contendo as novas declarações agora padronizadas (declarações simples, usando os dois pontos). Esses nós são adicionados no escopo da AST que está sendo alterado, abaixo dos nós originais.

Após a inserção de todos os novos nós, as declarações antigas (nós antigos), que não estavam padronizados, são removidos da AST, e as alterações realizadas na refatoração são adicionadas ao código fonte, através da chamada do método *addChangeFromModifiedAST()*.

3.1.4 Exemplo de uso

Na Figura 3.1 podem ser observados alguns exemplos dos diferentes tipos de declarações de variáveis permitidos no Fortran:

```
program statements

integer var_A
integer var_B, var_C
integer :: var_D
integer :: var_E, var_F, var_G

end program statements
```

Figura 3.1: Exemplos de tipos de declarações de variáveis em Fortran

Se a refatoração *Standardize Statements* fosse usada na Figura 3.1, obteria-se o código da Figura 3.2, que usa um padrão fixo de declaração e tornou-se um código de maior legibilidade, com um padrão normalizado de declarações.

```
program statements

integer :: var_A
integer :: var_B
integer :: var_C
integer :: var_D
integer :: var_E
integer :: var_F
integer :: var_G

end program statements
```

Figura 3.2: Refatoração aplicada no código da Figura 3.1

Na Figura 3.3 é possível observar um exemplo da refatoração em execução no Photran, na tela de *Preview* do editor. Observe que, após a refatoração, o código fonte apresentado ficou com sua legibilidade ligeiramente melhorada, e é possível observar uma padronização nas declarações de variáveis.

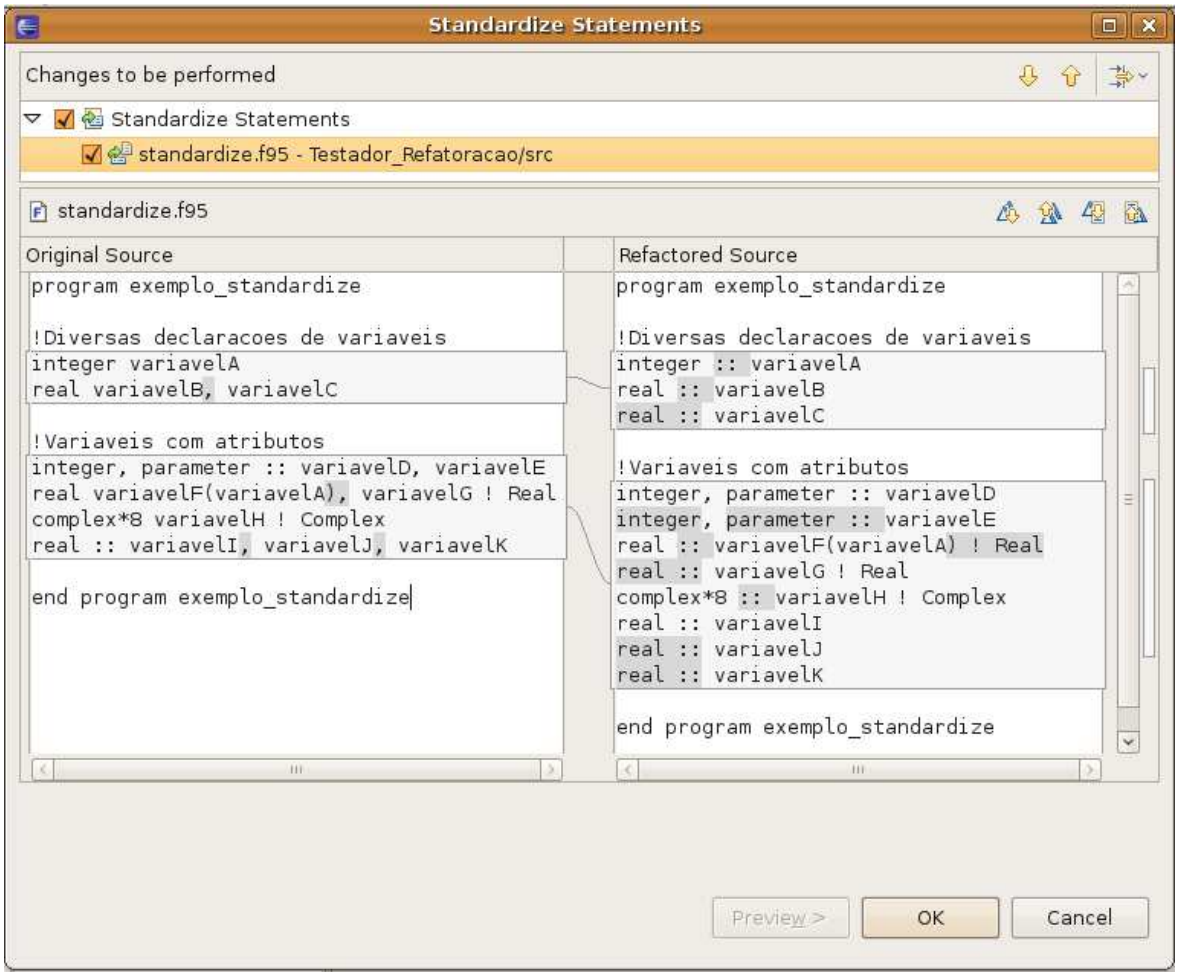

Figura 3.3: Exemplo de aplicação da refatoração *Standardize Statements*

3.2 Remover Variáveis Não Utilizadas (*Remove Unused Variables*)

– Você possui um código com diversos blocos comentados, porém, as variáveis usadas apenas nesses blocos comentados não foram comentadas ou eliminadas da seção de declarações, podendo assim prejudicar a legibilidade do código e aumentar o tamanho do código executável gerado.

– Essa refatoração elimina as variáveis não utilizadas no código fonte, melhorando sua legibilidade, podendo também diminuir o tamanho do código executável gerado.

3.2.1 Motivação

Normalmente, quando se programa um *software*, podem sobrar variáveis inutilizadas, muitas vezes, por descuido do próprio programador, que comenta algum trecho do código que não será mais utilizado, e esquece de comentar as declarações das variáveis que estavam contidas no bloco comentado. Um exemplo típico de tal erro, ou descuido, pode ser observado na Figura 3.4, onde está clara a evidência do esquecimento de variáveis não utilizadas ao longo do código.

```
program cylinder
! Calculate the area of a cylinder.

implicit none
real                :: area
integer             :: radius,height
real    , parameter :: pi = 3.14159
...
...
...

!area = 2*pi*(radius**2 + radius*height)

...
...
...
...

end program cylinder
```

Figura 3.4: Exemplo típico do descuido de um programador

Observe que a linha salientada da Figura 3.4 foi comentada, e assim as variáveis *area, radius, height* e *pi* não foram mais utilizadas ao longo do código, tornando-se uma situação clássica de esquecimento de comentar ou retirar a declaração das mesmas do início do código.

Tendo em vista que tais descuidos ocorrem com grande frequência, essa refatoração foi criada com a finalidade de eliminar tais variáveis. A eliminação dessas variáveis possibilita diminuir o tamanho do código executável gerado, e também é de grande valia quando se quer paralelizar o código, uma vez que ter menos variáveis facilita o controle dos acessos concorrentes na memória, gerando uma melhora no desempenho da execução do *software* (quando o próprio compilador não elimina as variáveis não utilizadas no processo de compilação) e melhorando a sua legibilidade. A refatoração responsável por eliminar as variáveis não usadas foi nomeada *Remove Unused Variables* no Photran.

3.2.2 Mecânica

- Verificar se o código usa declarações implícitas ou explícitas.
    - Se as declarações são implícitas é necessário transformá-las em explícitas.
- Percorrer o código fonte em busca de identificadores de declaração de variáveis.
- Verificar cada variável encontrada quanto ao número de referências no código fonte.
    - Se alguma variável verificada não possuir nenhuma referência, deve-se removê-la do código fonte.
- Compilar e testar o código, verificando seu correto funcionamento.
- A refatoração está completa.

3.2.3 Implementação

O processo de implementação contou com algumas facilidades providas pela AST do Photran. Com o método *ast.getRoot().getAllContainedScopes()* é possível obter todos os escopos da árvore sintática abstrata, e, assim, tem-se liberdade de percorrer e alterar um escopo por vez. Em cada escopo é possível obter a lista de todas as definições de variáveis nele contidas, com o uso do método *scope.getAllDefinitions()*. Uma vez que essa lista de definições é obtida, percorre-se a mesma em busca de definições de variáveis locais, que são identificadas através do método provido pelo Photran, *definition.isLocalVariable()*, que retorna verdadeiro caso a definição seja uma variável local, ou retorna falso, caso contrário.

Uma variável possui uma lista de referências, onde é descrito quais referências são feitas a ela no código fonte. Uma vez encontrada a variável local, é necessário obter-se essa lista, que com o uso do método *definition.findAllReferences()* se torna uma operação simples. Se o tamanho da lista de referências for igual a um (a declaração da variável é contada como uma referência, assim se o tamanho da lista de referências for igual a um, significa que a variável foi apenas declarada), a variável não foi usada, e ela deve ser removida da AST, com o método *remove()*, provido pelo *framework* do Photran.

Porém, no Photran não é possível fazer múltiplas execuções da mesma refatoração automaticamente, pois a AST modificada só é atualizada em arquivo após o usuário confirmar a sua ação de refatoração. Sendo assim, o usuário é alertado, antes de fazer a refatoração, que é necessário aplicar o processo novamente para que todas as variáveis não utilizadas sejam realmente removidas (como pode ser visto na Figura 3.5).

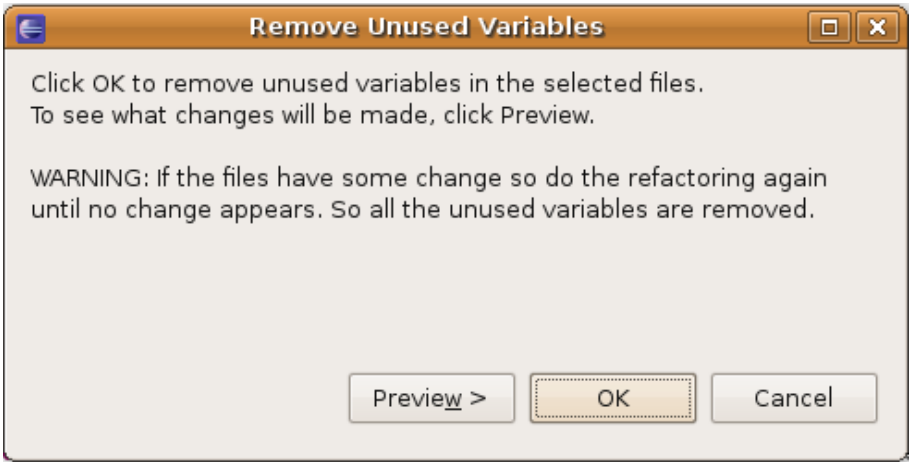

Figura 3.5: Aviso sobre como proceder com a refatoração

Além do aviso prévio quanto a refatoração, é mostrada uma informação de quando é necessário ou não refazer a refatoração, na tela de informação, que é visualizada antes de o usuário concordar com a aplicação das modificações no código (essa informação é obtida pela variável de *status* da refatoração). Nesta tela, como pode ser visto nas Figuras 3.6 e 3.7, o usuário fica ciente de quando deve fazer mais uma rodada da refatoração, ou quando todas as variáveis não usadas foram removidas do código fonte.

Após a remoção das variáveis não usadas, as alterações realizadas na refatoração são adicionadas ao código fonte, através da chamada do método *addChangeFromModifiedAST()*.

Um detalhe importante dessa refatoração, é que ela possui como requisito necessário

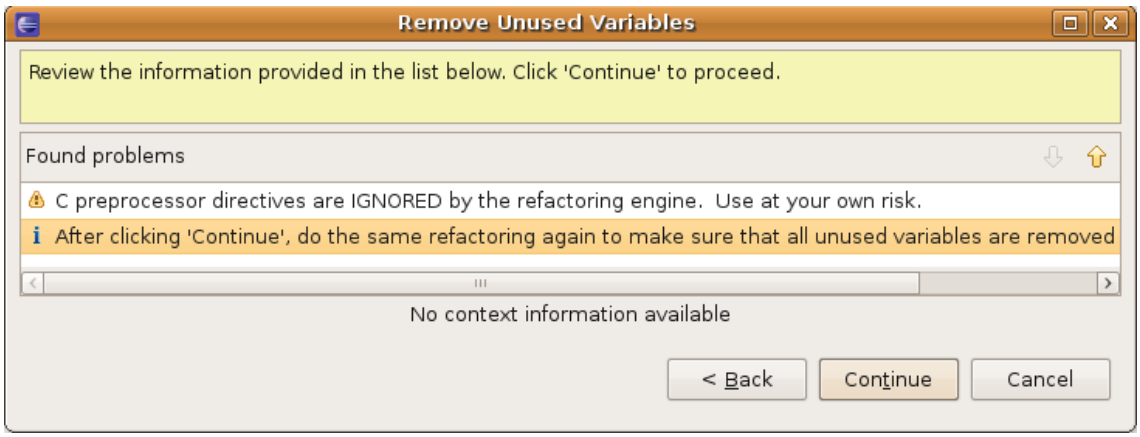

Figura 3.6: Aviso que o usuário deve fazer novamente a refatoração

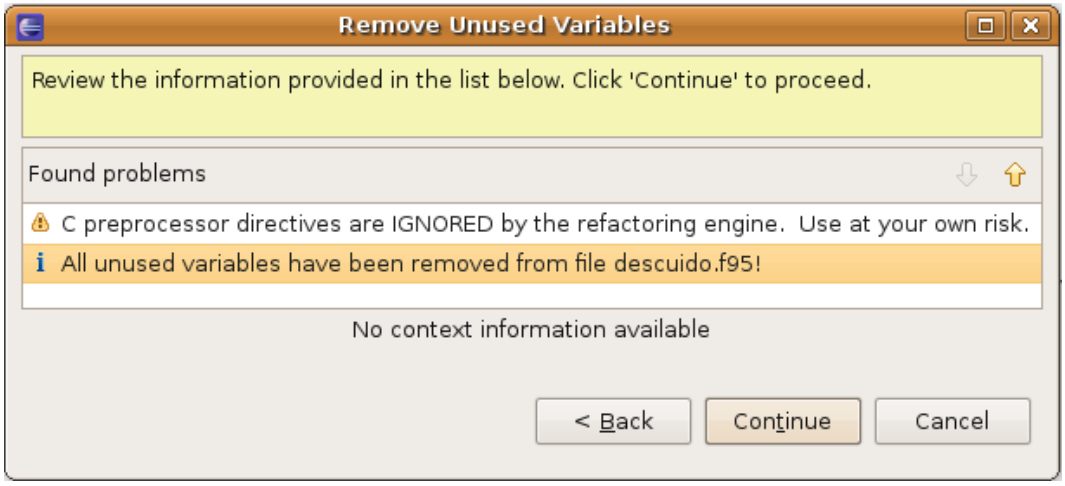

Figura 3.7: Aviso que todas as variáveis não usadas já foram removidas

à sua aplicação, que o código fonte a ser refatorado seja *Implicit None*. Para isso, caso o usuário tente aplicar essa técnica a um código que não respeite esse pré-requisito, a mensagem da Figura 3.8 é mostrada, indicando ao usuário que primeiramente ele deve usar a refatoração nomeada no Photran como *Introduce Implicit None*. Assim, o código fonte estará obecendo o pré-requisito necessário e poderá ser refatorado para a eliminação das variáveis.

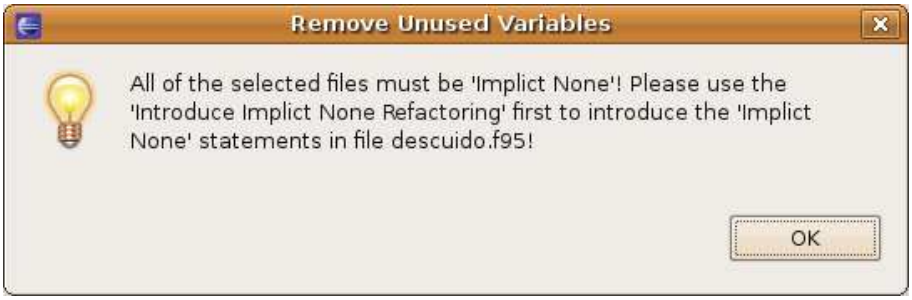

Figura 3.8: Aviso sobre o pré-requisito do *Implicit None*

3.2.4 Exemplo de uso

Na Figura 3.9 é possível observar um exemplo da refatoração em execução no Photran, na tela de *Preview* do editor. Observa-se que, após a refatoração, o código ficou com sua legibilidade ligeiramente melhorada, uma vez que aparecem nele apenas as variáveis realmente utilizadas, não ficando um código poluído.

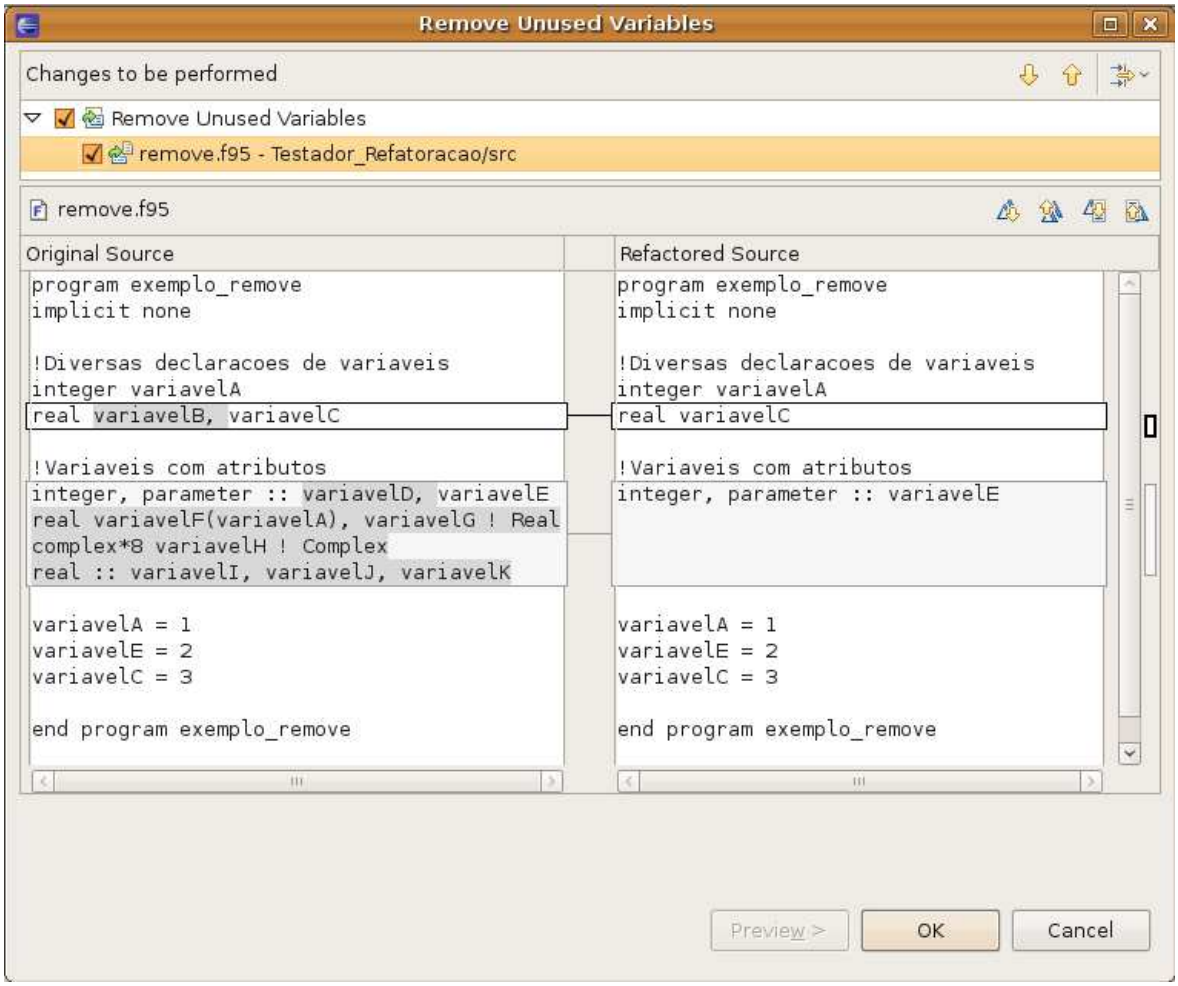

Figura 3.9: Exemplo de aplicação da refatoração *Remove Unused Variables*

3.3 Transformar declarações do tipo *Data* em *Parameter* (*Data To Parameter*)

– Você possui variáveis declaradas com a intenção de serem constantes, propagadas diretamente pelo compilador em tempo de compilação. Porém, devido a um erro de programação, essas variáveis são declaradas como variáveis estáticas, necessitando acessos para obter o seu conteúdo, causando perda de desempenho na execução da aplicação.

– Essa refatoração identifica ocasiões em que variáveis são declaradas como estáticas,

quando havia a intenção de serem declaradas como constantes (propagadas pelo compilador), fazendo a troca do tipo de declaração das mesmas, transformando-as em constantes.

3.3.1 Motivação

No Fortran, uma variável pode ser inicializada de diversas formas e, usualmente, em programas existem algumas variáveis que são constantes, isto é, tem o seu valor definido no início do código e ele nunca é alterado, sendo apenas referenciado (propagado pelo compilador) para usar em cálculos ou operações em que ele seja requerido. No Fortran existe um atributo que serve para esse fim, fazendo com que uma variável seja inicializada e seja uma constante, a declaração (atributo) *Parameter*.

Uma variável declarada como *parameter* tem um valor constante que o compilador propaga diretamente em todo o código fonte, porém, as vezes essa declaração é confundida com a declaração do tipo *data*.

Uma variável do tipo *data* é automaticamente *save*, ou seja, ela é uma variável estática. Quando se pensa em paralelizar um código fonte, a fim de ganhar desempenho, as variáveis estáticas podem ser acessadas em concorrência por várias *threads*[1], causando uma complicação na programação. Por isso é vantajoso usar *parameter* quando se quer apenas tornar uma variável constante em todo o código fonte.

Tendo em vista que tais erros ocorrem com frequência, essa refatoração, nomeada no Photran como *Data To Parameter*, foi criada com a finalidade de substituir essas declarações do tipo *data* que são confundidas com as declarações do tipo *parameter*.

3.3.2 Mecânica

- Percorrer o código fonte em busca de variáveis declaradas com o tipo *Data*.
    - Quando alguma dessas variáveis não sofre alteração do valor inicial deve-se:
        - Criar uma nova variável vazia do tipo *Parameter*;
        - Nomear e inicializar a nova variável com os valores da variável *Data*;
        - Remover a variável declarada como *Data* do código fonte.
- Compilar e testar o código, verificando seu correto funcionamento.
- A refatoração está completa.

[1] uma *thread*, ou linha de execução em português, é uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas simultaneamente.

3.3.3 Implementação

O processo de implementação desta técnica de refatoração automatizada consiste em, primeiramente, obter todos os escopos do código fonte, usando o método do *framework* do Photran que retorna uma lista de escopos, *ast.getRoot().getAllContainedScopes()*.

Para poder tranformar uma declaração do tipo *data nome / valor /* em uma declaração do tipo *parameter (nome = valor)*, a variável *nome* não pode ser alterada no programa, podendo ser apenas referenciada. Sendo assim, é necessário fazer uma verificação de todas as variáveis que tiveram seu valor alterado no código fonte, para ter certeza na hora de fazer a alteração do tipo de declaração que não seja transformada uma declaração do tipo *data* que tenha o seu valor alterado no código fonte. Caso uma das variáveis do tipo *data* tiver sido alterada, ela permanece sendo do tipo *data*. Com essa verificação uma lista de variáveis que foram alteradas é preenchida para poder ser referenciada na hora da aplicação das modificações.

Após obtidas todas as referências de variáveis alteradas, cada escopo é percorrido em busca de nós do tipo *ASTDataStmtNode*, que se referem a declarações do tipo *data*. Alguns exemplos de declarações do tipo *data* podem ser vistos na Figura 3.10.

```
data var_A /10/

data var_C /1/, var_D /2/

data var_E, var_F /10, 20/

data var_G /4/, var_H, var_I /7, 8/
```

Figura 3.10: Exemplos de declarações do tipo *Data*

Quando um nó do tipo *ASTDataStmtNode* é encontrado, é necessário percorrê-lo em busca das declarações contidas em sua lista de declarações e valores (*ASTDataStmtSetNode*). Então, faz-se uma comparação com os nomes das variáveis contidas na lista do nó e as variáveis modificadas obtidas anteriormete. Caso alguma das variáveis contida na lista do nó não esteja na lista de variáveis modificadas, significa que ela nunca foi modificada e, portanto, deve ser transformada em uma declaração do tipo *parameter*.

A cada variável encontrada que deve ter sua declaração modificada, é criado um novo nó na AST, contendo uma declaração do tipo *parameter*, com nome e inicialização idênticos aos da variável que estava na lista do nó tipo *data*.

Após serem criados os novos nós, eles são adicionados ao corpo do escopo, e as variáveis que foram adicionadas aos novos nós são removidas do nó com declaração do tipo *data*.

Assim, pode ser gerado um ganho de desempenho, uma vez que o valor das variáveis que são constantes passam a ser propagados em todas as suas referências pelo compilador, quando se está usando a forma de declaração correta.

Após a substituição das variáveis do tipo *data* que não sofrem alteração no código por variáveis do tipo *parameter*, que se destinam a ser valores constantes, as alterações realizadas na refatoração são adicionadas ao código fonte, através da chamada do método *addChangeFromModifiedAST()*.

3.3.4 Exemplo de uso

Na Figura 3.11 é possível observar um exemplo de uso de declarações do tipo *data* e *parameter*. Observe que a última variável está declarada como *data*, mas como ela não tem o seu valor modificado ao longo do código, a sua declaração deveria ser do tipo *parameter*, que é observado na Figura 3.12, onde o código da Figura 3.11 passou pelo processo de refatoração *Data To Parameter*.

```
program data_parameter

parameter ( constante_A = 5 )
data constante_B /10/
data constante_C /20/

...
...
...

constante_B = 30

write (*,*) "Constante_A =", constante_A
write (*,*) "Constante_B =", constante_B
write (*,*) "Constante_C =", constante_C

...
...
...


end program data_parameter
```

Figura 3.11: Exemplo típico de uso incorreto de declaração do tipo *data*

Na Figura 3.13 é possível observar um exemplo da refatoração em execução no Photran, na tela de *Preview* do editor. Observe que, após a refatoração, o código ficou com os tipos de declarações corretos, mantendo os nomes e valores originais das variáveis.

```
program data_parameter

parameter ( constante_A = 5 )
data constante_B /10/
parameter ( constante_C = 20 )

...
...
...

constante_B = 30

write (*,*) "Constante_A =", constante_A
write (*,*) "Constante_B =", constante_B
write (*,*) "Constante_C =", constante_C

...
...
...


end program data_parameter
```

Figura 3.12: Refatoração aplicada ao código da Figura 3.11

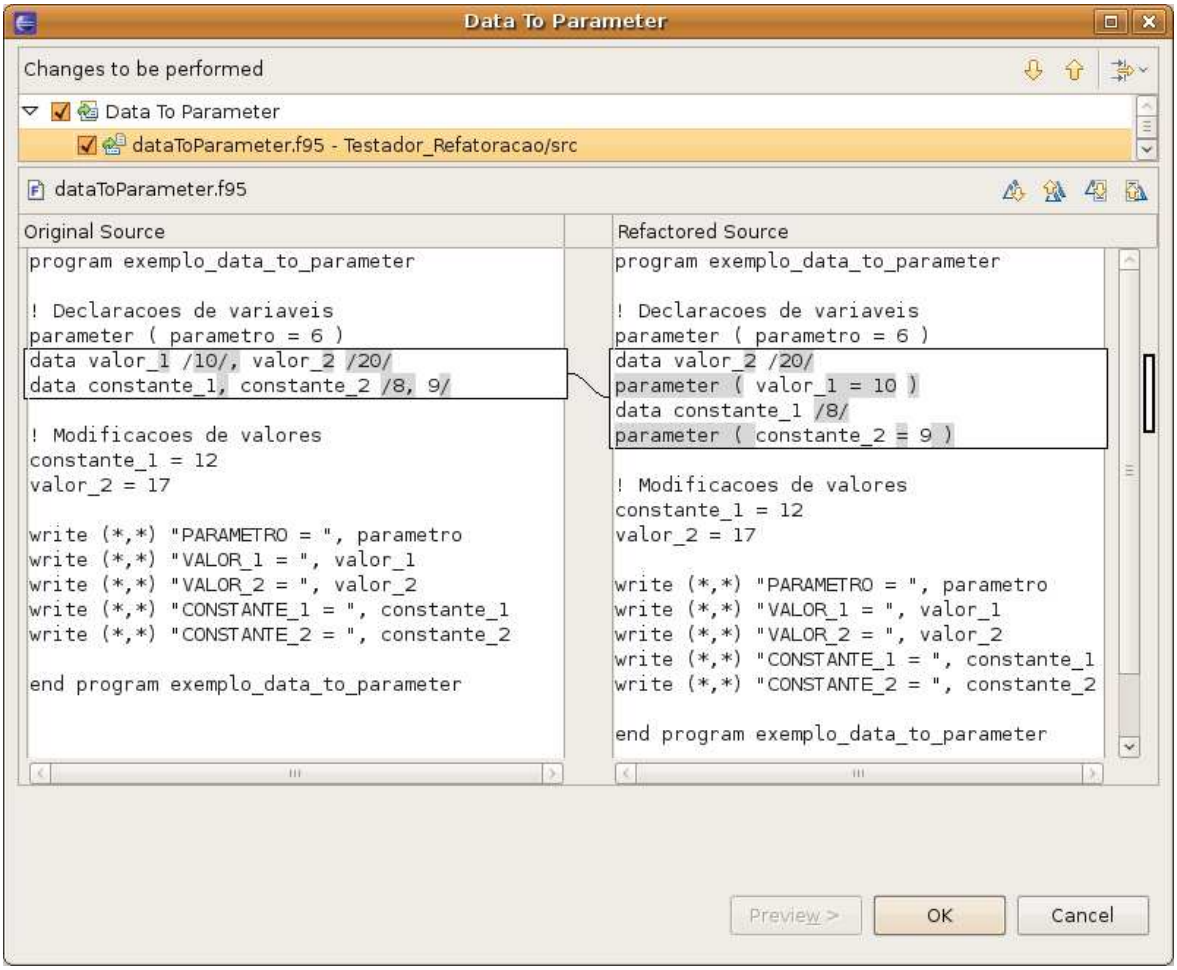

Figura 3.13: Exemplo de aplicação da refatoração *Data To Parameter*

3.4 Introduzir atributo *INTENT* (*Introduce INTENT*)

Essa refatoração foi desenvolvida durante o mestrado de Bruno Batista Boniati (BONIATI, 2009), com o objetivo de introduzir o atributo *INTENT* em declarações de argumentos de subprogramas, sendo utilizada para especificar a intenção de uso de determinado argumento de um subprograma.

Recentemente, o Photran sofreu uma atualização que causou mudanças internas nas hierarquias de classes. Como essa técnica de refatoração foi feita para a versão do *plugin* anterior à atualização, ela deixou de funcionar. Partindo deste princípio, neste trabalho foi feita a atualização desta refatoração para que ela voltasse a funcionar na nova versão do Photran.

3.4.1 Motivação

O atributo *INTENT* é uma construção recente (versão 90 do Fortran) e é utilizada para especificar a intenção de uso de determinado argumento em um subprograma. São três formas possíveis de indicar a intenção de uso de um argumento com o atributo *INTENT* (ADAMS et al., 2008):

- *INTENT(IN)*: especifica que o argumento está sendo utilizado para passar dados como entrada para o subprograma;
- *INTENT(OUT)*: especifica que o argumento está sendo usado para retornar resultados da chamada feita pelo programa e não pode ser usado para fornecer dados de entrada;
- *INTENT(INOUT)*: especifica que o argumento que está sendo passado define um valor de entrada e se for alterado seu valor permanecerá disponível após a execução do subprograma.

Com o uso do *INTENT*, é permitido ao programador conhecer a intenção dos argumentos de um subprograma apenas pela sua interface, sem a necessidade de ler todo seu código, favorecendo a legibilidade do mesmo. Também permite meios para o compilador realizar verificações mais apuradas (detectar se o uso do argumento está inconsistente com a intenção declarada, por exemplo) e utilizar estratégias de otimização.

A principal motivação dessa refatoração, é o fato de que existe uma grande quantidade de código legado que despreza o uso de tal técnica (o parâmetro *INTENT* é opcional e

foi adicionado à linguagem Fortran em suas versões mais recentes). Da mesma forma, identificar de forma manual todas as chamadas a subprogramas e a intenção de utilização de seus argumentos é uma tarefa um tanto exaustiva, o que justifica a necessidade de uma ação de refatoração automatizada.

3.4.2 Mecânica

A mecânica desta refatoração consiste na introdução do atributo *INTENT*, que deve ser realizada após a seleção, por parte do programador, de um subprograma do código fonte (no caso de Fortran, um bloco *SUBROUTINE* ou *FUNCTION*). Os parâmetros do subprograma devem receber o atributo *INTENT* de acordo com a forma que são utilizados dentro do subprograma (referenciado, modificado, ou ambos).

Em função da utilização que um argumento tem, pode-se definir e alterar seu comando de declaração para *INTENT(IN)* se ele é apenas referenciado, *INTENT(OUT)* se ele é apenas modificado ou *INTENT(INOUT)* se ele sofre modificações e é referenciado (antes de ser modificado).

Um requisito importante da refatoração *Introduce INTENT*, é que o subprograma que está sendo refatorado (ou o programa principal que o contém) precisa utilizar-se da declaração explícita de tipos (usar o comando *IMPLICIT NONE* exigindo que todas as declarações explicitem o tipo de dado). Assim, se faz necessário primeiramente usar a refatoração nomeada no Photran de *Introduce Implicit None*, para que essa condição seja satisfeita.

3.4.3 Implementação

A implementação dessa refatoração permaneceu a mesma descrita no trabalho de Bruno Batista Boniati (BONIATI, 2009), exceto que foram feitas pequenas modificações para adequar o código à nova versão do Photran.

Os trechos modificados do código, assim como explicações devido às mudaças aplicadas podem ser observados nas Figuras 3.14, 3.15, 3.16 e 3.17.

```
// Na versão original da refatoração o seguinte import era usado:
import org.eclipse.photran.internal.core.refactoring.infrastructure.FortranRefactoring;

// Na versão atualizada, o import acima foi substituído pelo seguinte:
import org.eclipse.photran.internal.core.refactoring.infrastructure.SingleFileFortranRefactoring;
```

Figura 3.14: Diferenças nas classes importadas

```
// Alterações no método processInterfaces(IFile file). Na versão original, quando havia uma
// declaração composta, e um argumento estava entre ela, essa declaração era ignorada, e o
// atributo INTENT não era adicionado na declaração da variável.
public void processInterfaces(IFile file){
    ...
    if (d.isSubprogramArgument()) {
        ASTTypeDeclarationStmtNode a = getTypeDeclarationStmtNode(d.getTokenRef().
            findToken().getParent());
        // Limitação ... só atua se a declaração for simples
        if (a.getEntityDeclList().size() == 1) {
            ASTAttrSpecNode at = getASTAttrSpecNode(a);
            spi.addParam(d.getDeclaredName(), at);
        }
    }
    ...
}

// Na versão atualizada da refatoração, quando ocorre de um dos argumentos estar em uma lista
// de declaração, é exibido um aviso, para que o usuário use a técnica de refatoração
// Standardize Statements, para que todas as variáveis fiquem normalizadas, e seja possível
// introduzir corretamente os atributos INTENT.
public void processInterfaces(IFile file){
    ...
    if (d.isSubprogramArgument()) {
        ASTTypeDeclarationStmtNode a = getTypeDeclarationStmtNode(d.getTokenRef().
            findToken().getParent());
        // Só atua se a declaração for simples.
        if (a.getEntityDeclList().size() == 1) {
            ASTAttrSpecNode at = getASTAttrSpecNode(a);
            spi.addParam(d.getDeclaredName(), at);
        }else
            // Se alguma declaração for composta, é necessário padronizá-la usando a
            // refatoração Standardize Statements.
            fail("All␣Statements␣must␣be␣simple␣(integer␣::␣a).␣To␣Standardize␣all␣\n" +
            "statements␣in␣file,␣use␣Standardize␣Statements␣Refactoring␣first!");
    }
    ...
}
```

Figura 3.15: Diferenças no método *processInterfaces(IFile file)*

3.5 Sumário do capítulo

Nesse capítulo foram apresentadas as refatorações implementadas neste trabalho. As três técnicas implementadas (*Santadardize Statements*, *Remove Unused Variables* e *Data To Parameter*) tiveram sua motivação, mecânica e implementações descritas em detalhes. Foram mostradas também, as alterações introduzidas na refatoração *Introduce INTENT* (BONIATI, 2009). O Photran passou por uma atualização, após a criação dessa refatora-

```
// Na versão original, quando se tratava do processamento de nós de subprograma, o trecho do
// código fonte era o que está apresentado abaixo:
private void processCallsAndFunctions(IASTNode node){
    ...
    if (node instanceof ASTCallStmtNode) {
        SubProgramInformation spi = new SubProgramInformation(((ASTCallStmtNode)node).
            getDataRef().get(0).getName().getText(), fileInEditor.getName() );
        for (IASTNode child1 : node.getChildren()) {
            if (child1 instanceof ASTSeparatedListNode){
                IASTNode n_aux = (ASTSeparatedListNode)child1;
                for (IASTNode child2 : n_aux.getChildren()) {
                    if (child2 instanceof ASTSectionSubscriptNode) {
                        IASTNode n_aux2 = ((ASTSectionSubscriptNode)child2).getExpr();
                        if (n_aux2 instanceof ASTVarOrFnRefNode)
                            spi.addParam(((ASTVarOrFnRefNode)n_aux2).getName().getName().
                                getText(), null);
                    }
                }
            }
        }
        callProgram.add(spi);
    }
    ...
}

// Na versão atualizada da refatoração, houveram melhorias nesse trecho do código, através de
// melhorias implementadas na atualização do Photran, como pode ser visto abaixo:
private void processCallsAndFunctions(IASTNode node){
    ...
    if (node instanceof ASTCallStmtNode) {
        SubProgramInformation spi = new SubProgramInformation(((ASTCallStmtNode)node).
            getSubroutineName().getText(), fileInEditor.getName());
        IASTListNode<ASTSubroutineArgNode> argumentos = ((ASTCallStmtNode)node).
            getArgList();
        for(ASTSubroutineArgNode arg : argumentos){
            ASTVarOrFnRefNode exp = (ASTVarOrFnRefNode)arg.getExpr();
            spi.addParam(exp.getName().getName().getText(), null);
        }
        callProgram.add(spi);
    }
    ...
}
```

Figura 3.16: Diferenças no método *processCallsAndFunctions(IASTNode node)*

ção, e a mesma parou de funcionar, necessitando de manutenção para voltar a funcionar.

No Capítulo 4, são descritas as modalidades de testes realizados nas refatorações desenvolvidas neste trabalho, com a finalidade de avaliá-las e validá-las. Foram feitos testes de funcionamento em diferentes versões da linguagem Fortran, através de códigos For-

```
// Na versão original, o método para buscar a posição do início do nome de uma variável era
// como apresentado abaixo. Porém, nesse método, quando existem comentários antes da
// declaração, ele deixa de funcionar, retornando a posição errada da variável.
int getPosicaoDaVariavel(String s){
    int i, j;
    boolean comecou = false;
    i = 0;
    for (j=0; j<s.length(); j++){
        if ( (s.charAt(j) == '␣') & (comecou)){
            i = j;
            break;
        }else if (s.charAt(j) != '␣')
            comecou = true;
    }
    return i;
}

// Na versão atualizada, foi corrigido o problema do retorno errado da posição do nome da
// variável, ficando como apresentado abaixo:
int getVariablePosition(String s, String attr){
    String[] coments = s.split("\n");
    int size = coments.length − 1;
    int i, j;
    boolean start = false;
    int attr_length = attr.length()−1;
    for(int k=0; k<coments.length − 1; k++)
        size += coments[k].length();
    String statement = coments[coments.length − 1];
    i = 0;
    for (j=0; j<statement.length(); j++){
        if ( (statement.charAt(j) == '␣') & (start)){
            i = j;
            break;
        }else if (statement.charAt(j) != '␣')
            start = true;
    }
    if(attr_length < 0)
        attr_length = 0;
    return i + size + attr_length;
}
```

Figura 3.17: Diferenças no método *getPosicaoDaVariavel(String s)*

tran distribuídos juntamente com o Photran, assim como testes de análise no impacto das refatorações no desempenho de aplicações usando uma aplicação de alto desempenho cedida pelo Laboratório de Micrometeorologia vinculado à Universidade Federal de Santa Maria. Ainda nesse capítulo, é feita uma breve comparação deste trabalho com outros correlatos, servindo também como uma forma de avaliação.

4 AVALIAÇÃO

Para avaliar o impacto da utilização de técnicas de refatoração em aplicações de alto desempenho escritas em linguagem Fortran, o trabalho utilizou-se do código fonte de uma aplicação cedida pelo Laboratório de Micrometeorologia vinculado à Universidade Federal de Santa Maria. O objetivo do estudo deste código é avaliar de que forma a utilização de técnicas de refatoração podem influenciar no desempenho (de forma positiva ou negativa) desta aplicação, escrita em Fortran 77.

A aplicação foi submetida a duas formas de compilação para a realização dos testes, compilando-se, primeiramente, sem o uso de otimizações do compilador, e posteriormente, compilando-se com o uso de otimizações do compilador. O objetivo da aplicação dessas duas formas de compilação foi para analisar se o desempenho era afetado quando se utilizava a refatoração em conjunto com a otimização. A metodologia utilizada para os testes de desempenho podem ser vistos na seção 4.1.

Após avaliado o impacto das refatorações em aplicações de alto desempenho, passou-se a avaliar o comportamento das mesmas em diferentes códigos Fortran, de diferentes versões da linguagem, com o objetivo de verificar o seu correto funcionamento nas versões do Fortran suportadas pelo *plugin* Photran.

Para a realização dos testes em códigos de diferentes versões do Fortran, as refatorações foram aplicadas nos códigos Fortran contidos no projeto *org.eclipse.photran-samples*, distribuído juntamente com o Photran. Nesse projeto estão contidos diversos códigos fonte de Fortran 77, 90, 95 e 2003.

Os principais códigos do projeto *org.eclipse.photran-samples* utilizados para os testes foram os códigos *bstfit.f90*, que é um programa que faz o cálculo de distâncias usando o método do Ajuste de Curvas (*Curve Fitting*), e o código *gauselim.f90*, que é um programa que resolve um sistema de equações lineares com o método da Eliminação Gaussiana.

Os testes realizados nesses dois códigos podem ser vistos em detalhes nas seções 4.2.1 e 4.2.2.

Pode-se observar que os códigos usados para os testes são de aplicações científicas, e como descrito durante este trabalho, códigos como esses podem apresentar erros.

Na seção 4.3 são mostrados alguns trabalhos relacionados com o assunto de refatoração, mostrando que também existem pesquisas sobre refatoração em outras linguagens de programação, salientando que as técnicas de refatoração não estão restritas apenas ao uso em códigos fonte, fazendo-se uma breve comparação dessas pesquisas com este trabalho, servindo também como uma forma de avaliação do mesmo.

4.1 Avaliação de Desempenho: Análises Micrometeorológicas

A micrometeorologia é uma subdivisão das ciências atmosféricas que estuda fenômenos físicos de pequena escala espaço-temporal que ocorrem na camada atmosférica que faz contato com a superfície da Terra (com espessura de 1 km em média). Análises micrometeorológicas contribuem para a solução de problemas de ordem ambiental bem como para a previsão do tempo e do clima.

Normalmente, as análises micrometeorológicas se utilizam de observações de campo. O conjunto de dados coletados em estações meteorológicas passa por análises e interpretações demandando alto poder de processamento. A aplicação a ser estudada é utilizada para processar conjuntos de dados de um dia de coleta (um arquivo com 100 *MBytes* em média). Ao final do processamento, os dados processados são gravados em arquivos e resultam em aproximadamente 70 *MBytes* de resultados.

A aplicação estudada é importante para os experimentos do laboratório, mas dado seu tempo de vida e algumas construções obsoletas utilizadas no seu desenvolvimento, apresenta uma alta complexidade de compreensão e manutenção, representando um problema quando novas funcionalidades ou análises precisam ser adicionadas à aplicação.

Para a realização dos testes com a referida aplicação, foi utilizado o compilador GCC (*gfortran*), no sistema operacional GNU/Linux Ubuntu 8.10, em um computador Intel Core 2 Duo 1.8 *GHz*, com 3 *GBytes* de memória.

Inicialmente, a aplicação foi compilada sem nenhuma otimização, tanto a versão original, quanto a versão refatorada, sendo, posteriormente, ambas as versões compiladas com a opção de otimização *-O1* do compilador. Cada versão da aplicação compilada foi

executada três vezes, para se obter uma média de tempo de execução, resultado na tabela de tempos de execução que pode ser vista na Figura 4.1.

Código Original Sem Otimizações	Código Refatorado Sem Otimizações	Código Original Otimização -O1	Código Refatorado Otimização -O1
21m29.201s	21m35.038s	6m49.508s	6m53.127s
21m26.721s	21m36.021s	6m55.437s	6m57.490s
21m31.631s	21m37.149s	6m54.820s	6m50.374s
Média de tempo:	Média de tempo:	Média de tempo:	Média de tempo:
21m32.184s	21m36.069s	6m53.225s	6m53.664s
	Diferença de tempo em relação ao original:		Diferença de tempo em relação ao original:
	3.885s (0,3%)		0.439s (0,1%)

Figura 4.1: Tabela de tempos de execução da aplicação

Quanto às otimizações, foi utilizada apenas a opção *-O1*, pois a aplicação passou a comportar-se indevidamente quando utilizadas as opções *-O2* e *-O3*, que são mais agressivas, na medida em que seu grau de otimização é maior. Normalmente, quando utilizadas essas duas últimas opções em aplicações de alto desempenho, é bastante comum ocorrerem erros na execução, pois muitas mudanças podem ocorrer nesse processo, podendo gerar pequenos erros (anomalias) que não são identificados no processo de compilação.

Como pode ser visto na Figura 4.1, o tempo médio de execução do código original com um arquivo de entrada de 106.8 *MBytes*, foi de 21m32.184s, com as opções de otimização do compilador desligadas.

Após a verificação do tempo de execução do programa original, foram aplicadas as refatorações nas subrotinas do programa (10 subrotinas). Apenas as duas primeiras refatorações (*Standardize Statements* e *Remove Unused Variables*) apresentaram resultados significativos, pois na aplicação não foi usado o tipo de declaração *Data*, na qual a refatoração *Data To Parameter* atua.

Com o uso da refatoração *Remove Unused Variables* algumas variáveis não usadas presentes em alguns dos arquivos das subrotinas foram removidas. Na subrotina *autocorrelacao* a variável *corre(n)* foi removida. Já, na subrotina *detrend* foi encontrada uma variável nomeada *f*, que não era usada e foi removida. Na subrotina *rotacao3d* foram removidas as variáveis *cor1*, *var* e *dp*. Como é possível se observar, na aplicação não havia muitas variáveis não utilizadas, mas a técnica de refatoração foi capaz de identificar e remover todas as variáveis não utilizadas, cumprindo sua descrição. Já com o uso da re-

fatoração *Standardize Statements*, as 10 subrotinas do *software* tiveram suas declarações de variáveis padronizadas.

O tempo médio de execução da aplicação, com o mesmo arquivo de entrada, tendo todas as suas subrotinas refatoradas com as refatorações *Standardize Statements* e *Remove Unused Variables*, foi de 21m36.069s, também com as diretivas de otimização do compilador desativadas.

Posteriormente, compilou-se a aplicação original com as diretivas de otimização do compilador ativadas (*-O1*), obtendo-se, uma média de 6m53.225s de tempo de execução. Com o uso da otimização houve uma diminuição significativa no tempo de execução da aplicação, sendo gasto apenas cerca de 32% do tempo que a aplicação sem otimizações demorou para executar.

Não muito diferente da aplicação original, quando a aplicação refatorada foi compilada com as opções de otimização ativadas (*-O1*), foi obtida uma média de 6m53.664s de tempo de execução, sendo gasto também apenas cerca de 32% do tempo de execução da aplicação refatorada sem otimizações.

Com esses resultados, pode-se concluir que com os uso das refatorações implementadas neste trabalho, o código fonte fica com maior legibilidade e melhor estruturado e o desempenho não é afetado negativamente, nem mesmo quando se usam as opções de otimização do compilador. A diferença de tempo de execução entre a aplicação original e a aplicação refatorada foi de apenas 3.885s (representando aproximadamente 0,3% do tempo), sem o uso de otimizações na compilação. Já a diferença de tempo de execução da aplicação original e a aplicação refatorada, com o uso da otimização *-O1*, foi de apenas 0.439s (representando aproximadamente 0,1% do tempo).

4.2 Avaliação das refatorações em diferentes versões do Fortran

Nesta seção pode-se observar os testes realizados em dois dos códigos fonte do projeto *org.eclipse.photran-samples*, distribuído juntamente com o Photran. O objetivo dos testes nesses códigos é mostrar o correto funcionamento das refatorações em diferentes versões da linguagem Fortran suportadas pelo *plugin* Photran.

4.2.1 Eliminação Gaussiana

Em álgebra linear, a Eliminação Gaussiana é um algoritmo para resolver sistemas de equações lineares, encontrando o posto de uma matriz, e calculando a inversa de uma matriz inversível quadrada.

Operações elementares de linha são utilizadas para reduzir uma matriz à forma escalonada de linha, sendo que a Eliminação Gaussiana é suficiente para resolver muitas aplicações.

O processo de Eliminação Gaussiana é divido em duas partes. A primeira parte (*Forward Elimination*) reduz um sistema dado a um sistema de forma triangular ou de forma escalonada, ou resulta em uma degeneração da equação sem solução, indicando que o sistema não tem solução. Isto é conseguido através do uso de operações elementares de linha. O segundo passo utiliza a substituição de volta para encontrar a solução do sistema obtido pela primeira parte do algoritmo. O algoritmo é interessante também porque calcula uma matriz de decomposição. As três operações elementares lineares utilizadas no método (multiplicando as linhas, trocando as linhas, e adicionando múltiplos de linhas a outras linhas) elevam-se multiplicando a matriz original com matrizes inversíveis à esquerda. A primeira parte do algoritmo calcula uma decomposição LU[1], enquanto a segunda parte escreve a matriz original como o produto de uma matriz inversível unicamente determinada e uma matriz linha-reduzida escalonada unicamente determinada (WIKIPEDIA.ORG, 2009a; STEINBRUCH; WINTERLE, 1987).

O primeiro código do projeto *org.eclipse.photran-samples* usado para fazer a aplicação foi o código *gauselim.f90*, que possui cerca de 240 linhas, sendo um código científico. A refatoração *Standardize Statements* proporcionou uma melhoria na legibilidade das declarações em algumas regiões do código, como pode ser visto nas Figuras 4.2, 4.3, 4.4 e 4.5.

Com as demais refatorações (*Remove Unused Variables* e *Data To Parameter*), o código não sofreu alterações, pois não possui nenhuma variável não usada e também não usa nenhuma constante declarada como *data*. Mesmo assim, foi possível perceber a melhora na legibilidade, quando há uma padronização na declaração de variáveis.

[1]Em álgebra linear, a decomposição LU (em que LU vem do inglês *lower* e *upper*) é uma forma de fatoração de uma matriz não singular como o produto de uma matriz triangular inferior (*lower*) e uma matriz triangular superior (*upper*).

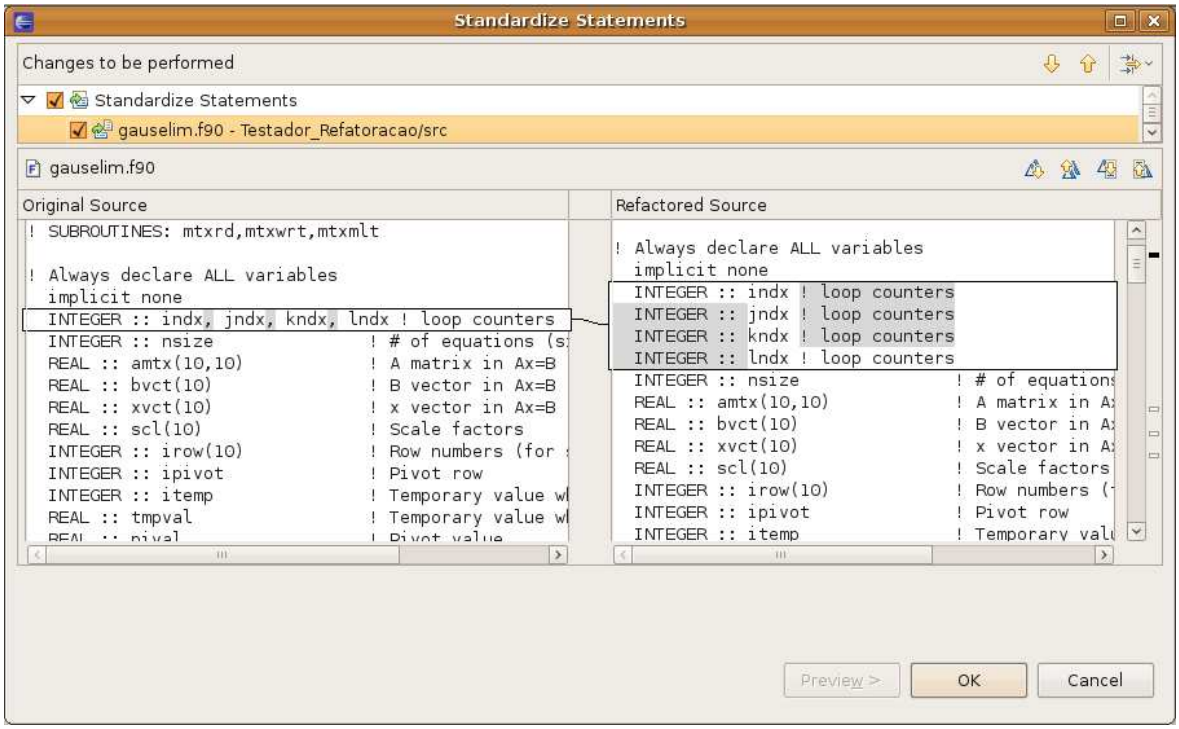

Figura 4.2: Refatoração *Standardize Statements* no código *gauselim.f90* (Bloco 1)

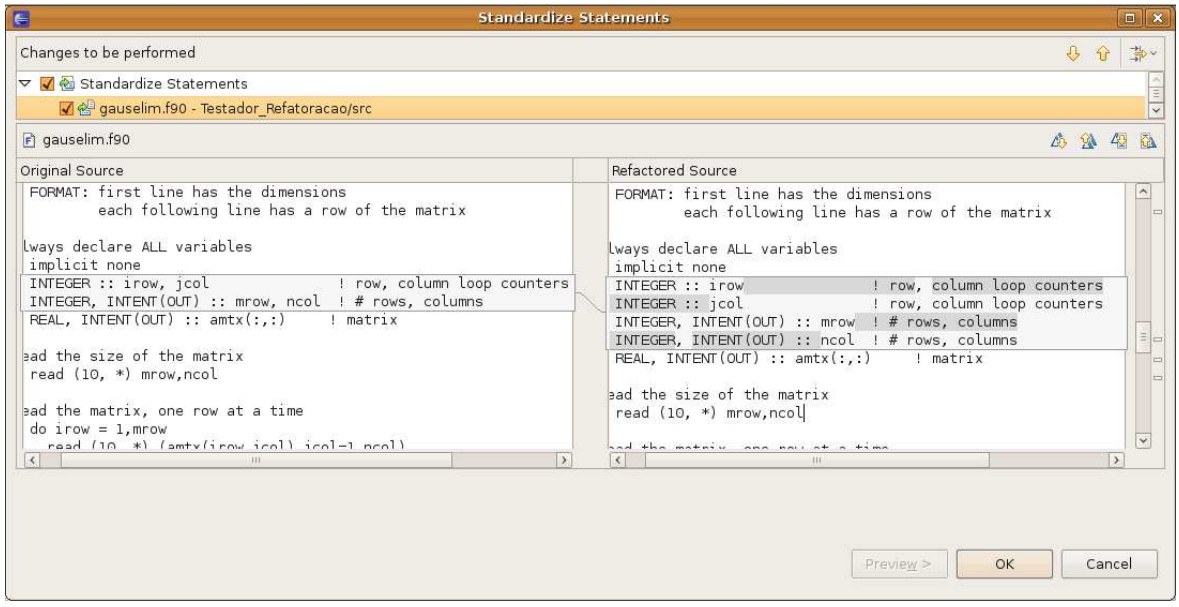

Figura 4.3: Refatoração *Standardize Statements* no código *gauselim.f90* (Bloco 2)

4.2.2 Ajuste de Curvas (*Curve Fitting*)

O método de ajuste de curvas (*Curve Fitting*) é um método que consiste em encontrar uma curva que se ajuste a uma série de pontos e que, possivelmente, cumpra uma série de parâmetros adicionais. O ajuste de curvas é muito utilizado para, a partir de dados conhecidos, fazer-se extrapolações. Por exemplo, conhece-se os dados de consumo anual

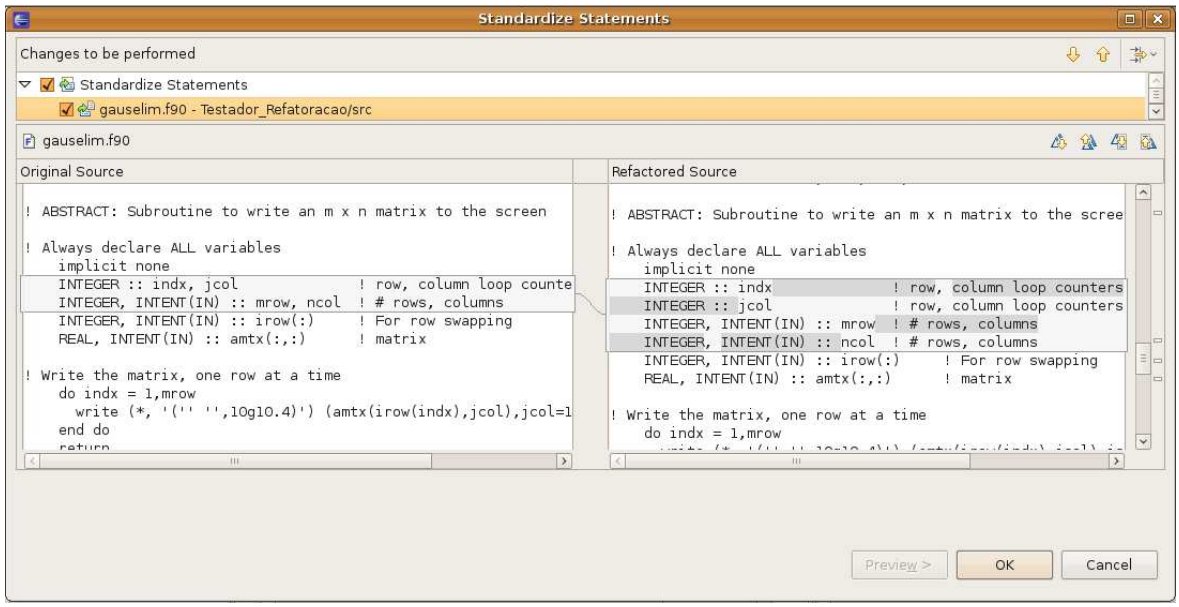

Figura 4.4: Refatoração *Standardize Statements* no código *gauselim.f90* (Bloco 3)

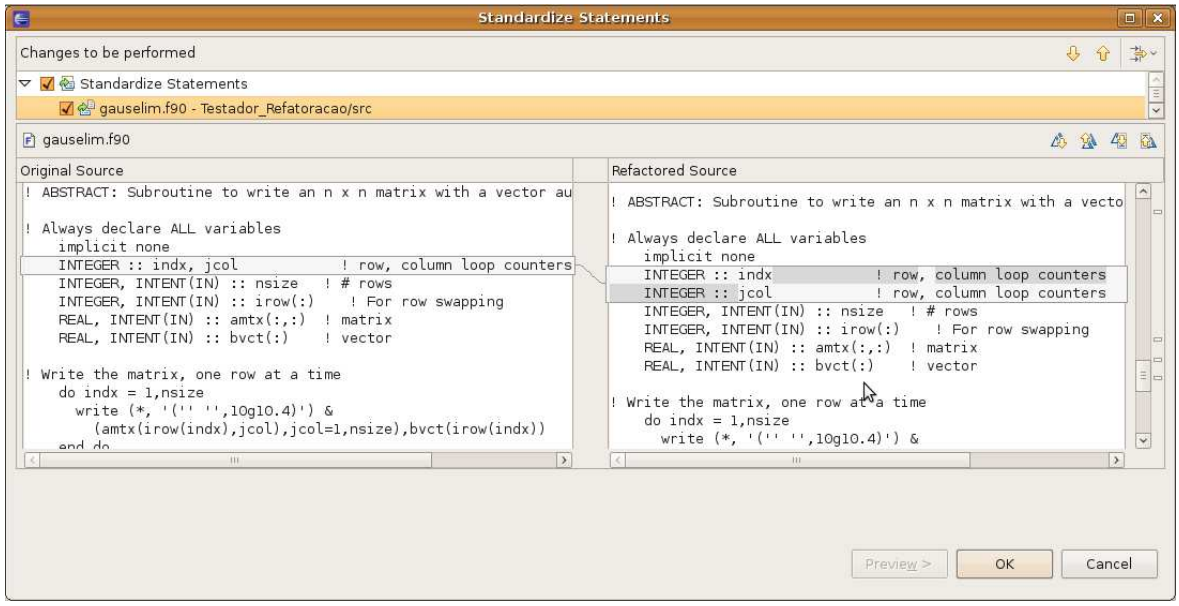

Figura 4.5: Refatoração *Standardize Statements* no código *gauselim.f90* (Bloco 4)

de carga elétrica de uma cidade. A partir destes dados conhecidos, pode-se fazer projeções para o futuro e, com isso, fazer-se um planejamento para que a cidade seja suprida de forma adequada nos anos subsequentes. A idéia é obter uma curva que melhor se ajuste aos dados disponíveis. Conhecida a equação da curva, pode-se determinar valores fora do intervalo conhecido (WIKIPEDIA.ORG, 2009b).

O segundo código do projeto *org.eclipse.photran-samples* usado para realização dos testes, um código maior, com cerca de 870 linhas, foi o código nomeado *bstfit.f90*. Nesse código foram adicionadas propositalmente variáveis não utilizadas em regiões aleatórias,

afim de testar a refatoração *Remove Unused Variables*, e também foram adicionadas constantes declaradas como *data*, afim de testar a refatoração *Data To Parameter*.

Nas Figuras 4.6, 4.7 e 4.8, pode-se observar o resultado de regiões modificadas quando foi aplicada a refatoração *Remove Unused Variables*. Observe que o código preservou a identação e os comentários que havia no local onde encontravam-se as variáveis não utilizadas.

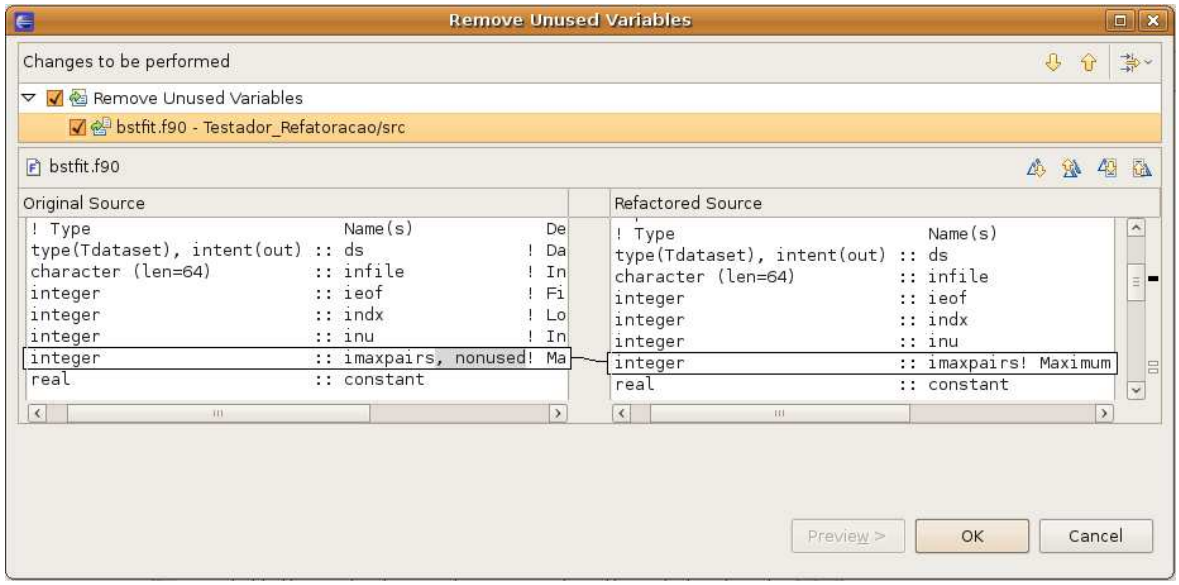

Figura 4.6: Refatoração *Remove Unused Variables* no código *bstfit.f90* (Bloco 1)

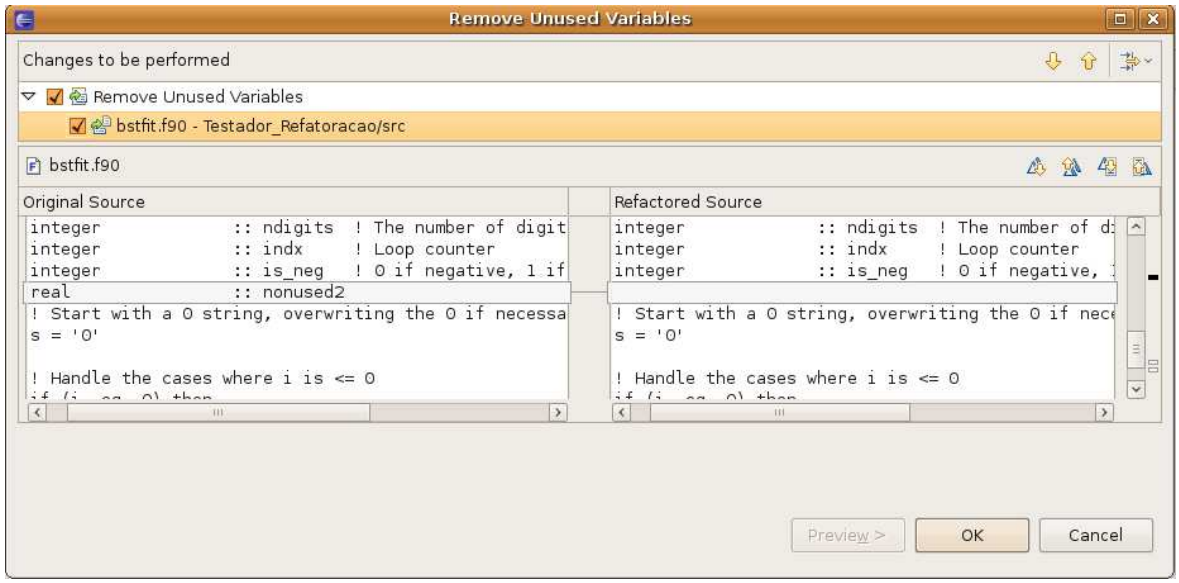

Figura 4.7: Refatoração *Remove Unused Variables* no código *bstfit.f90* (Bloco 2)

Agora, nas Figuras 4.9 e 4.10, pode-se observar a aplicação da refatoração *Data To Parameter*, que foi aplicada após a refatoração *Remove Unused Variables*, pois como

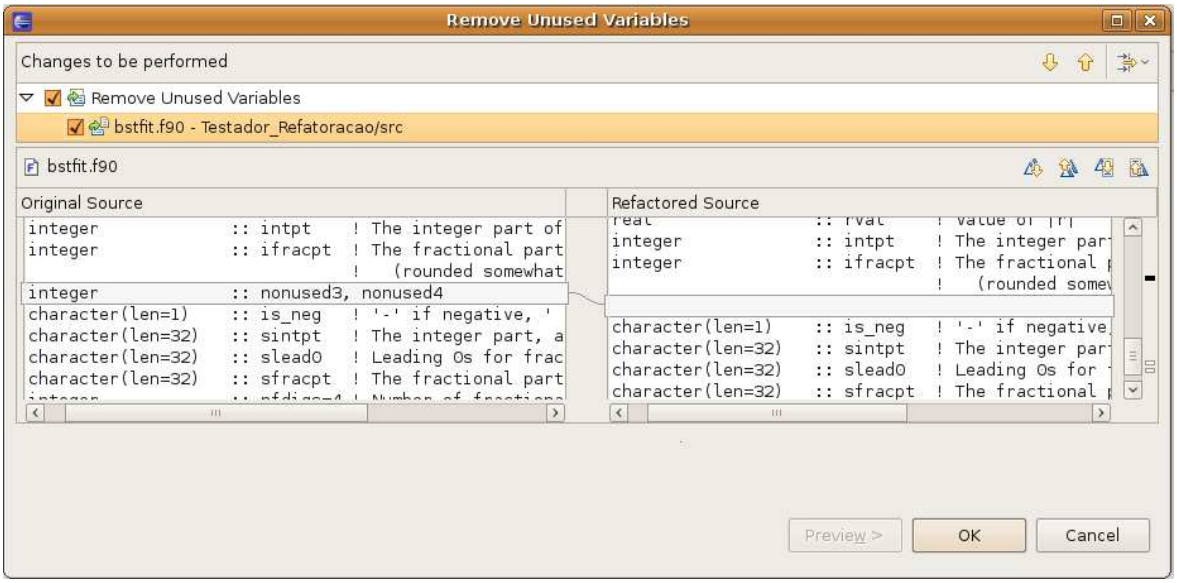

Figura 4.8: Refatoração *Remove Unused Variables* no código *bstfit.f90* (Bloco 3)

mencionado anteriormente, um código fonte pode passar por diversas refatorações sucessivas, afim de atingir o melhoramento desejado, obtido por um conjunto de pequenas modificações que geram uma grande modificação.

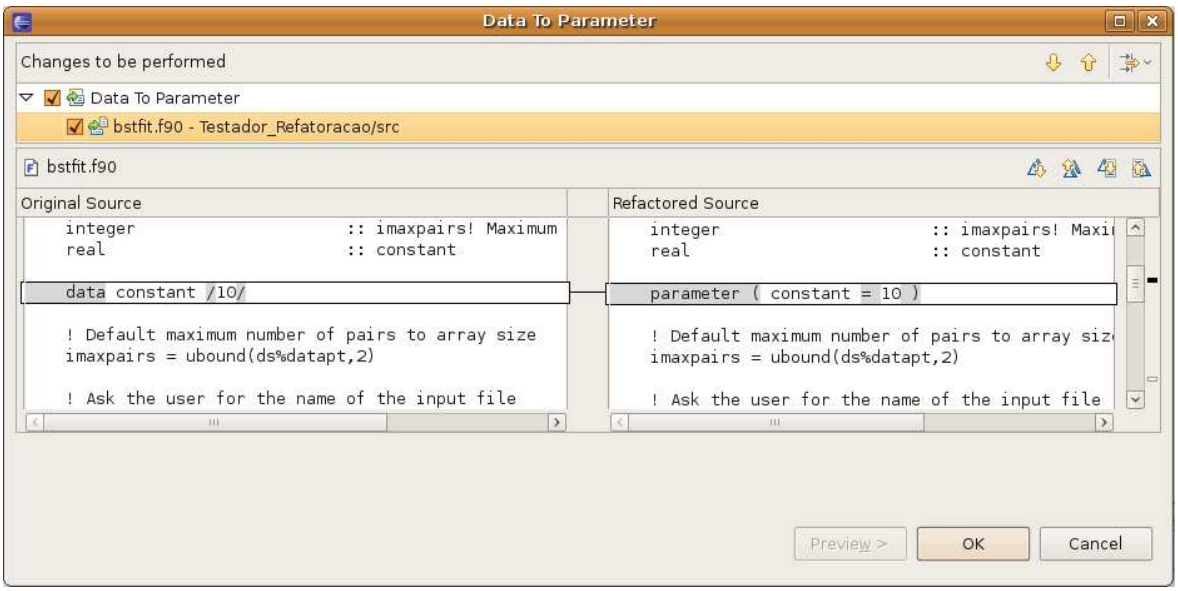

Figura 4.9: Refatoração *Data To Parameter* no código *bstfit.f90* (Bloco 1)

4.3 Trabalhos Relacionados

O uso de refatoração não está restrito apenas ao código fonte de aplicações, podendo também ser aplicado a outros artefatos do *software*, como no projeto da aplicação, modelos de análise, bancos de dados, dentre outros. Alguns trabalhos abordam o uso de

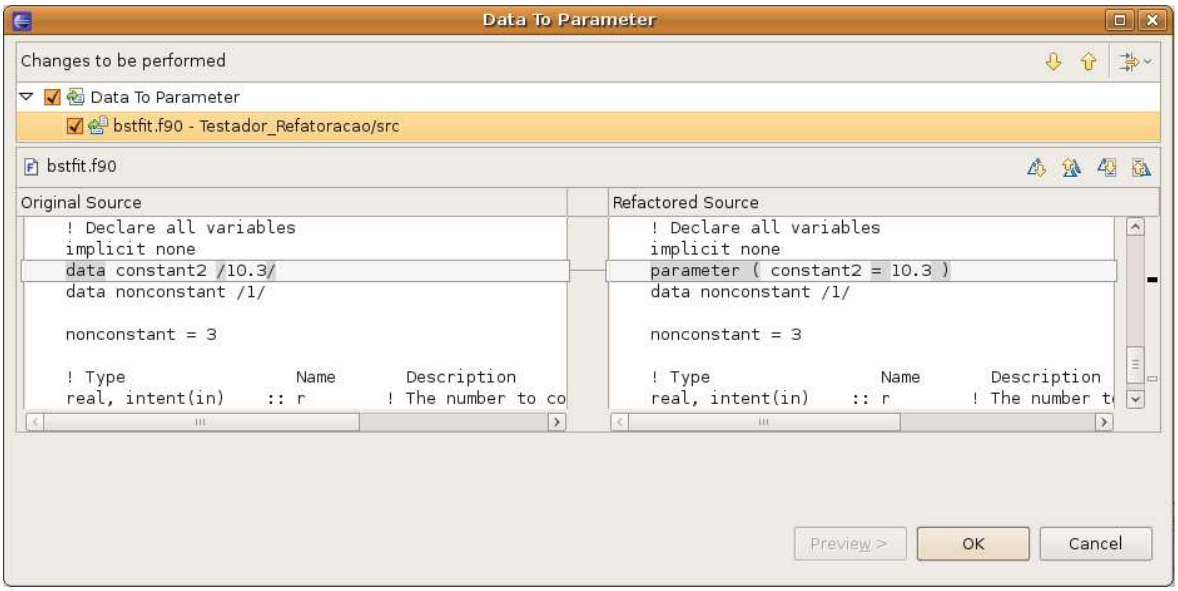

Figura 4.10: Refatoração *Data To Parameter* no código *bstfit.f90* (Bloco 2)

refatorações em diagramas UML (ASTELS, 2002) assim como a automatização de refatorações integradas a editores UML (BOGER; STURM; FRAGEMANN, 2003). Existem também, na área de banco de dados, trabalhos que discutem a refatoração no contexto da evolução de esquemas relacionais (BOEHM et al., 2007). Um dos grandes desafios é definir mecanismos para manter a consistência entre diferentes artefatos (implementação e modelo) de *software* com o uso de refatoração.

Na pesquisa de refatoração para Fortran, podem-se encontrar trabalhos que fazem a reestruturação de código fonte seqüencial para código fonte paralelo/distribuído (EVERAARS; ARBAB; BURGER, 1996). Nesse caso, com a refatoração de *software* é possível se aproveitar o melhor desempenho oferecido pelos sistemas paralelo/distribuídos, sem ter de reescrever todo o código legado para conseguir a transformação, que muitas vezes é um processo de alto custo.

O presente trabalho explora as refatorações de código Fortran, não visando apenas o ganho de desempenho, mas também almeja-se obter como resultado um código de melhor qualidade, maior legibilidade, e de mais fácil manutenção.

Também, vale lembrar que as pesquisas relacionadas com refatoração abrangem muitas linguagens de programação diferentes. No Eclipse (ECLIPSE.ORG, 2009a), podem-se encontrar 23 refatorações para a linguagem Java, sendo que para as outras linguagens suportadas pelo IDE, o número de refatorações é bastante reduzido, sugerindo a carência de pesquisa e desenvolvimento relacionados a refatoração para determinadas linguagens

de programação.

Existem pesquisas que estudam os desafios da refatoração de código fonte da linguagem de programação C (GARRIDO; JOHNSON, 2002). Nesse caso, assim como o Fortran, existe uma grande dificuldade em refatorar códigos C legados. Embora exista uma grande quantidade de códigos legados nessa linguagem, ferramentas de refatoração para C com suporte completo para as diretivas de pré-processamento para essa linguagem ainda não existem. Logo, a semelhança dos problemas de refatoração tratados neste trabalho, com Fortran, também encontam-se em aberto quando se trata da linguagem C.

Encontram-se, também, pesquisas que visam construir modelos de ferramentas para refatorações independentes de linguagens. Normalmente, a maioria das ferramentas que atuam em processos de refatoração são dependentes das linguagens para as quais as refatorações são desenvolvidas, impedindo uma maior integração de técnicas de refatoração a ambientes de programação em geral.

No trabalho *A Meta-Model for Language-Independent Refactoring* (TICHELAAR et al., 2000), são estudadas as similaridades entre as refatorações das linguagens Java e Smalltalk, onde é criado um meta-modelo de linguagem independente, mostrando que é possível construir uma ferramenta de refatoração independente de linguagens. Usando um meta-modelo independente de linguagens pode-se eliminar a dependência de existir uma ferramenta de refatoração para cada tipo de linguagem diferente.

No *plugin* Photran, pode-se encontar um pouco da idéia de se ter uma única ferramenta para a refatoração de diversas linguagens, uma vez que o Photran possui um menu de refatoração para cada tipo de linguagem suportada. Porém, as refatorações do Photran são dependentes das linguagens para as quais foram desenvolvidas. Por exemplo, uma refatoração de Fortran não funciona em um código Java. Assim, no Photran ainda é necessário implementar refatorações para uma linguagem específica desejada, desde que o *plugin* tenha suporte de refatorações para essa linguagem.

4.4 Sumário do capítulo

Conforme visto nas seções anteriores, a aplicação das refatorações desenvolvidas nesse trabalho mostraram resultados satisfatórios, uma vez que o desempenho das aplicações não é afetado negativamente, e resulta em um código melhor estruturado e com melhor legibilidade. Também, durante os testes foram usados três códigos fonte diferentes,

de tamanhos diferentes e versões da linguagem diferentes, para avaliar o funcionamento das refatorações em diferentes versões da linguagem.

Além das conclusões relativas ao tempo de execução é preciso registrar que a utilização de técnicas automatizadas agilizam e dão maior segurança ao programador, não inibindo a possibilidade de erros, mas diminuindo a chance de o usuário introduzir erros no processo de refatoração. O uso da ferramenta Photran também agrega a funcionalidade do programador pré-visualizar o código refatorado e decidir ou não por sua utilização.

Também neste capítulo, foi feita uma breve comparação deste trabalho com outros correlatos, mostrando que existem pesquisas sobre refatoração em outras linguagens de programação, salientando também que as técnicas de refatoração não estão restritas apenas ao uso em códigos fonte.

Assim, é possível inferir que o objetivo do trabalho foi alcançado, e o resultado são as refatorações aqui implementadas, que já foram submetidas ao comitê de avaliação do Photran, para serem integradas oficialmente no *plugin* e serem usadas livremente por qualquer usuário do Photran.

5 CONSIDERAÇÕES FINAIS

Aplicações de cunho científico geralmente se preocupam com o emprego de metodologias e técnicas para a execução eficiente de aplicações, abrangindo técnicas de programação, melhorias de código e distribuição ou paralelização de tarefas. Os melhores resultados em termos de desempenho estão associados à utilização de uma arquitetura de *hardware* específica com construções apropriadas de *software*.

As técnicas de refatoração têm por objetivo amenizar a degradação natural pela qual o código fonte de aplicações sofre com o tempo, uma vez que se podem aplicar diversas técnicas sobre um mesmo código para que este fique adequado com as novas necessidades.

O processo de refatoração pode ser executado manualmente, mas o ganho de qualidade em escala se dá quando técnicas são automatizadas e integradas a ferramentas para desenvolvimento de *software* (IDEs). A utilização do suporte de ferramentas automatizadas para refatorar reduz o risco de erros e inconsistências, além de reduzir também o trabalho e o custo de desenvolvimento e manutenção de *softwares*.

Neste trabalho foi explorada a utilização de técnicas de refatoração sobre aplicações escritas em linguagem Fortran (geralmente, de cunho científico), objetivando melhorar o projeto de código, assim como a legibilidade do mesmo, e detectar oportunidades de ganho de desempenho (reduzindo o número de acesso à variáveis quando se substitui *data* por *parameter* nos locais corretos, por exemplo). As técnicas estudadas foram automatizadas e integradas à ferramenta Photran, um *plugin* do Eclipse que oferece recursos para o ciclo de desenvolvimento de aplicações Fortran e que oferece um *framework* que permite estender funcionalidades de refatoração para essa linguagem de programação.

As principais contribuições do trabalho são a identificação e automatização de técnicas de refatoração para linguagem Fortran, sendo que as técnicas desenvolvidas neste trabalho (*Standardize Statements*, *Remove Unused Variables* e *Data To Parameter*) se mostraram

eficientes em todos os testes, atendendo aos requisitos para os quais as mesmas foram desenvolvidas, sem prejudicar o desempenho das aplicações e gerando um código mais legível e melhor estruturado, como pode ser observado no Capítulo 4, com os *screenshots* dos resultados das aplicações das refatorações.

A utilização do Photran como ferramenta de apoio à codificação e automatização de técnicas de refatoração também merece destaque. Ferramentas como o Photran representam um importante avanço no sentido de se preencher a lacuna existente entre a grande quantidade de código Fortran legado (em especial de aplicações científicas) e o limitado número de ferramentas de apoio ao desenvolvimento com técnicas de refatoração integradas. Desde seu advento, em meados de 2004, é possível observar um crescimento importante no número de usuários que se utilizam da ferramenta como também de voluntários que implementam melhorias e agregam funcionalidades à mesma, para contribuir e poder torná-la uma ferramente de excelência na área de refatoração de aplicações escritas em linguagem Fortran.

Como sugestões para trabalhos futuros, pretende-se aprofundar os conhecimentos sobre refatorações, fazendo uma maior revisão bibliográfica sobre o tema, desenvolvendo-se também novas técnicas de refatoração para Fortran. No projeto *Fortran Refactoring: FortiFact*, que vem sendo desenvolvido em conjunto entre as instituições UFRGS, UFSM e CPTEC (INPE), e de onde foram tiradas as propostas para duas das técnicas de refatoração desenvolvidas nesse trabalho, encontra-se uma grande lista de propostas de novas refatorações para Fortran, e deseja-se dar continuidade à colaboração com o projeto, implementando-se algumas das técnicas descritas no mesmo.

Algumas das técnicas descritas no projeto *Fortran Refactoring: FortiFact* que poderiam ser implementadas como trabalhos futuros são:

- *Árvore de Chamadas*: Para cada subrotina, incluir um comentário a precedendo, que indica a lista de subrotinas que podem chamá-la, bem como a árvore de subrotinas que ela chama. O uso dessa refatoração pode ser fundamental para poder entender a lógica de um programa.

- *Associação de Use*: Quer-se determinar a árvore de *uses* e usá-la para determinar em qual módulo foi declarada uma variável usada em outro módulo.

- *Use Only*: Quer-se substituir todos os *use* por *use only*, a fim de listar explicitamente

e exatamente apenas os símbolos necessários ao módulo que os quer incluir.

- *Modularização*: Criar um módulo a partir de um subconjunto de subrotinas, gerando sua interface e colocando o *use* no lugar apropriado de onde se quer chamar os procedimentos. Também se deve privatizar o que deve ser privatizado e tornar público o que pode ser público.

REFERÊNCIAS

ADAMS, J. C.; BRAINERD, W. S.; HENDRICKSON, R. A.; MAINE, R. E.; MARTIN, J. T.; SMITH, B. T. **The Fortran 2003 Handbook: the complete syntax, features and procedures**. [S.l.: s.n.], 2008.

ASTELS, D. Refactoring with UML. In: INT'L CONF. EXTREME PROGRAMMING AND FLEXIBLE PROCESSES IN SOFTWARE ENGINEERING (XP), 2002. **Proceedings...** [S.l.: s.n.], 2002. p.67–70. Alghero, Sardinia, Italy.

BOEHM, A. M.; SEIPEL, D.; SICKMANN, A.; WETZKA, M. Squash: a tool for analyzing, tuning and refactoring relational database applications. In: . [S.l.: s.n.], 2007. p.82–98.

BOGER, M.; STURM, T.; FRAGEMANN, P. Refactoring Browser for UML. In: . [S.l.: s.n.], 2003. p.366–377.

BONIATI, B. B. **Refatoração de Programas Fortran de Alto Desempenho**. 2009. Dissertação (Mestrado) — Universidade Federal de Santa Maria, Santa Maria, RS.

BONIATI, B.; CHARAO, A.; STEIN, B. Automação de Refatorações para Programas Fortran de Alto Desempenho. In: WSCAD-SSC'09: X WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO / X SIMPóSIO EM SISTEMAS COMPUTACIONAIS, 2009, São Paulo. **Anais...** [S.l.: s.n.], 2009.

CHEN, N.; OVERBEY, J. **Photran 4.0 Developer's Guide**. 2008.

CORNéLIO, M. **Refactorings as Formal Refinements**. 2004. Tese (Doutorado) — Universidade Federal de Pernambuco.

DE, V. **A foundation for refactoring FORTRAN 90 in Eclipse**. 2004. Dissertação (Mestrado) — University of Illinois at Urbana-Champaign, Urbana-Champaign.

DEURSEN, A. van; MOONEN, L.; BERGH, A. van den; KOK, G. **Refactoring Test Code**. Anais...

ECLIPSE.ORG. **Eclipse**. Disponível em: `http://www.eclipse.org/`. Acesso em: dezembro de 2009.

ECLIPSE.ORG. **Photran - An Integrated Development Environment for Fortran**. Disponível em: `http://www.eclipse.org/photran/`. Acesso em: dezembro de 2009.

ECLIPSE.ORG. **Eclipse C/C++ Development Tooling - CDT**. Disponível em: `http://www.eclipse.org/cdt/`. Acesso em: dezembro de 2009.

EVERAARS, C. T. H.; ARBAB, F.; BURGER, F. J. Restructuring sequential Fortran code into a parallel/distributed application. In: ICSM '96: PROCEEDINGS OF THE 1996 INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, 1996, Washington, DC, USA. **Anais...** IEEE Computer Society, 1996. p.13–22.

FOWLER, M. **Refatoração: aperfeiçoando o projeto de código existente**. [S.l.: s.n.], 2004.

FOWLER, M. **Refactoring Home Page**. Disponível em: `http://www.refactoring.com/`. Acesso em: dezembro de 2009.

GARRIDO, A.; JOHNSON, R. Challenges of refactoring C programs. In: IWPSE '02: PROCEEDINGS OF THE INTERNATIONAL WORKSHOP ON PRINCIPLES OF SOFTWARE EVOLUTION, 2002, New York, NY, USA. **Anais...** ACM, 2002. p.6–14.

GNU. **GCC, the GNU Compiler Collection**. Disponível em: `http://gcc.gnu.org/`. Acesso em: dezembro de 2009.

GRISWOLD, W. G.; NOTKIN, D. Automated assistance for program restructuring. **ACM Trans. Softw. Eng. Methodol.**, New York, NY, USA, v.2, n.3, p.228–269, 1993.

ISO. **ISO/IEC 14977: information technology: syntactic metalanguage: extended bnf**. 1.ed. [S.l.: s.n.], 1996.

JONES, J. **Abstract Syntax Tree Implementation Idioms**. Disponível em: http://jerry.cs.uiuc.edu/~plop/plop2003/Papers/Jones-ImplementingASTs.pdf Acesso em: dezembro de 2009.

KOFFMAN, E. B.; FRIEDMAN, F. L. **FORTRAN**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.

MENS, T.; TOURWé, T. A Survey of Software Refactoring. **IEEE Trans. Softw. Eng.**, Piscataway, NJ, USA, v.30, n.2, p.126–139, 2004.

NYHOFF, L.; LEESTMA, S. **Fortran 90 for Engineers and Scientists**. [S.l.: s.n.], 1997.

OVERBEY, J. L. **Virtual Program Graph**. Disponível em: http://jeff.over.bz/software/vpg/doc/. Acesso em: dezembro de 2009.

OVERBEY, J. L.; JOHNSON, R. E. A Survey of Software Refactoring: a foundation for the rapid development of source code transformation tools. In: **Software Language Engineering**: first international conference, 2008, toulouse, france. [S.l.: s.n.], 2008.

OVERBEY, J. L.; JOHNSON, R. E. Generating Rewritable Abstract Syntax Trees. In: **Software Language Engineering**: first international conference, 2009, toulouse, france. Berlin, Heidelberg: Springer-Verlag, 2009. p.114–133.

RICARTE, I. L. M. **Introdução à Compilação**. [S.l.: s.n.], 2008.

ROBERTS, D. B. **Practical analysis for refactoring**. [S.l.: s.n.], 1999.

ROBERTS, D.; BRANT, J.; JOHNSON, R. An Automated Refactoring Tool. In: . Chicago, EUA: [s.n.], 1996.

STEINBRUCH, A.; WINTERLE, P. **Álgebra Linear**. São Paulo, BR: [s.n.], 1987.

TICHELAAR, S.; DUCASSE, S.; DEMEYER, S.; NIERSTRASZ, O. A Meta-Model for Language-Independent Refactoring. In: INTERNATIONAL SYMPOSIUM ON PRINCIPLES OF SOFTWARE EVOLUTION, 2000, 2000. **Anais...** [S.l.: s.n.], 2000. p.154–164.

WIKIPEDIA.ORG. **Gaussian Elimination**. Disponível em: http://en.wikipedia.org/wiki/Gaussian_elimination. Acesso em: dezembro de 2009.

WIKIPEDIA.ORG. **Curve Fitting**. Disponível em: `http://en.wikipedia.org/wiki/Curve_fitting`. Acesso em: dezembro de 2009.

APÊNDICE A CÓDIGO FONTE DAS IMPLEMENTAÇÕES

Neste apêndice se encontram os códigos fonte referentes às técnicas de refatoração desenvolvidas neste trabalho, chamadas *Standardize Statements*, *Remove Unused Variables* e *Data To Parameter*, no *plugin* Photran do IDE Eclipse. Também encontra-se o código fonte da refatoração *Introduce INTENT*, que foi atualizada para funcionar na versão utilizada do *plugin*. Os códigos fonte de cada uma delas podem ser vistos nas seções A.1, A.2, A.3, A.4, respectivamente.

A.1 *Standardize Statements*

A.1.1 Interface com o usuário (*StandardizeStatementsAction.java*)

Nesse código fonte estão contidas duas classes, sendo que uma é responsável por receber a chamada do usuário e associar a ação de refatoração com seu respectivo assistente e a outra é o próprio assistente (*wizard*) da refatoração.

```
package org.eclipse.photran.internal.refactoring.ui;

import java.util.ArrayList;

import org.eclipse.core.resources.IFile;
import org.eclipse.ltk.ui.refactoring.UserInputWizardPage;
import org.eclipse.photran.internal.core.refactoring.StandardizeStatementsRefactoring;
import org.eclipse.photran.internal.core.refactoring.infrastructure.AbstractFortranRefactoring;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.ui.IEditorActionDelegate;
import org.eclipse.ui.IWorkbenchWindowActionDelegate;

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

```
    public StandardizeStatementsAction() {
        super(StandardizeStatementsRefactoring.class,
                FortranStandardizeStatementsRefactoringWizard.class);
    }

    @Override
    protected AbstractFortranRefactoring getRefactoring(ArrayList<IFile> files) {
        return new StandardizeStatementsRefactoring(files);
    }

    public static class FortranStandardizeStatementsRefactoringWizard extends
            AbstractFortranRefactoringWizard {

        protected StandardizeStatementsRefactoring standardizeStatementsRefactoring;

        public FortranStandardizeStatementsRefactoringWizard(
                StandardizeStatementsRefactoring r) {
            super(r);
            this.standardizeStatementsRefactoring = r;
        }

        @Override
        protected void doAddUserInputPages() {
            addPage(new UserInputWizardPage(standardizeStatementsRefactoring.getName()) {

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

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

                    Label lbl = new Label(top, SWT.NONE);
                    lbl.setText("Click␣OK␣to␣standardize␣the␣statements␣in␣the␣selected␣files.␣
                            To␣see␣what␣changes␣will␣be␣made,␣click␣Preview.");
                }
            });
        }
    }
}
```

A.1.2 Arquivo de ação da refatoração (*StandardizeStatementsRefactoring.java*)

Nesse código fonte está a principal classe a ser codificada, que consiste na ação propriamente dita da refatoração.

```
package org.eclipse.photran.internal.core.refactoring;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
```

```
import org.eclipse.core.resources.IFile;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.OperationCanceledException;
import org.eclipse.ltk.core.refactoring.RefactoringStatus;
import org.eclipse.photran.core.IFortranAST;
import org.eclipse.photran.internal.core.analysis.binding.ScopingNode;
import org.eclipse.photran.internal.core.lexer.Token;
import org.eclipse.photran.internal.core.parser.ASTDerivedTypeDefNode;
import org.eclipse.photran.internal.core.parser.ASTEntityDeclNode;
import org.eclipse.photran.internal.core.parser.ASTExecutableProgramNode;
import org.eclipse.photran.internal.core.parser.ASTTypeDeclarationStmtNode;
import org.eclipse.photran.internal.core.parser.ASTTypeSpecNode;
import org.eclipse.photran.internal.core.parser.Parser.IASTListNode;
import org.eclipse.photran.internal.core.parser.Parser.IASTNode;
import org.eclipse.photran.internal.core.refactoring.infrastructure.
    MultipleFileFortranRefactoring;
import org.eclipse.photran.internal.core.refactoring.infrastructure.Reindenter;
import org.eclipse.photran.internal.core.refactoring.infrastructure.SourcePrinter;

/**
 * Refatoração para padronizar as declarações de variáveis.
 *
 * @author Gustavo Rissetti
 **/

public class StandardizeStatementsRefactoring extends MultipleFileFortranRefactoring {

    public StandardizeStatementsRefactoring(ArrayList<IFile> myFiles) {
        super(myFiles);
    }

    @Override
    public String getName() {
        return "Standardize␣Statements";
    }

    @Override
    protected void doCheckInitialConditions(RefactoringStatus status, IProgressMonitor pm)
        throws PreconditionFailure {
        ensureProjectHasRefactoringEnabled(status);
        removeFixedFormFilesFrom(this.selectedFiles, status);
    }

    @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();
    }
}

// Método auxiliar utilizado para verificar se uma declaração contém os dois pontos (::).
// Foi necessário construir esse método para tratar do caso em que existem os dois pontos
// em um comentário depois da declaração, por exemplo:
// integer variável ! comentario :: fim do comentário
// Se fosse usado o recurso do java (if(s.indexOf("::") != -1)), não seria tratado o caso
// do exemplo acima.
private boolean points(String s) {
    for (int i = 0; i < s.length() - 1; i++) {
        char p1 = s.charAt(i);
        char p2 = s.charAt(i + 1);
        if (p1 == '!' || p2 == '!') {
            return false;
        } else if (p1 == ':' && p2 == ':') {
            return true;
        }
    }
    return false;
}

private void makeChangesTo(IFile file, IFortranAST ast, RefactoringStatus status,
        IProgressMonitor pm) throws PreconditionFailure {
    List<ScopingNode> scopes = ast.getRoot().getAllContainedScopes();
    for (ScopingNode scope : scopes) {
        if (!(scope instanceof ASTExecutableProgramNode) && !(scope instanceof
                ASTDerivedTypeDefNode)) {
            IASTListNode<IASTNode> body = (IASTListNode<IASTNode>) scope.getBody();
            List<ASTTypeDeclarationStmtNode> statements = new LinkedList<
                    ASTTypeDeclarationStmtNode>();
            for (IASTNode node : body) {
                if (node instanceof ASTTypeDeclarationStmtNode) {
                    IASTListNode<ASTEntityDeclNode> variables = ((
                            ASTTypeDeclarationStmtNode) node).getEntityDeclList();
                    ASTTypeSpecNode type_node = new ASTTypeSpecNode();
                    // Pega o tipo das variáveis.
                    String type = ((ASTTypeDeclarationStmtNode) node).getTypeSpec().toString
                            ().trim();
                    String[] typeWithoutComments = type.split("\n");
                    type = typeWithoutComments[typeWithoutComments.length - 1].trim();
                    Token text_type = new Token(null, type);
```

```
            type_node.setIsInteger(text_type);
            for (int i = 0; i < variables.size(); i++) {
                ASTTypeDeclarationStmtNode new_statement = (
                        ASTTypeDeclarationStmtNode) node.clone();
                if (i > 0) {
                    new_statement.setTypeSpec(type_node);
                }
                IASTListNode<ASTEntityDeclNode> new_variable = (IASTListNode<
                        ASTEntityDeclNode>) variables.clone();
                List<ASTEntityDeclNode> list_variables_to_remove = new LinkedList<
                        ASTEntityDeclNode>();
                for (int j = 0; j < variables.size(); j++) {
                    if (j != i) {
                      list_variables_to_remove.add(new_variable.get(j));
                    }
                }
                new_variable.removeAll(list_variables_to_remove);
                new_statement.setEntityDeclList(new_variable);
                // Coloca os :: caso na declaração original não tenha.
                String source = SourcePrinter.getSourceCodeFromASTNode(
                        new_statement);
                int position_type = new_statement.getTypeSpec().toString().length();
                String twoPoints = "";
                String source_1 = source.substring(0, position_type);
                String source_2 = source.substring(position_type, source.length());
                if (!points(source_2)) {
                    twoPoints = "␣::";
                }
                // Nova declaração, já com os ::.
                source = source_1 + twoPoints + source_2;
                new_statement = (ASTTypeDeclarationStmtNode) parseLiteralStatement(
                        source);
                // Adiciona uma referência da antiga declaração.
                statements.add((ASTTypeDeclarationStmtNode) node);
                // Adiciona a nova declaração.
                statements.add(new_statement);
            }
        }
    }
    // Insere as novas declarações, já padronizadas na AST.
    for (int i = 0; i < statements.size(); i += 2) {
        body.insertBefore(statements.get(i), statements.get(i + 1));
        Reindenter.reindent(statements.get(i + 1), ast);
    }
    // Remove da AST as antigas declarações que estavam fora do padrão.
    for (int i = 0; i < statements.size(); i += 2) {
        ASTTypeDeclarationStmtNode delete = statements.get(i);
        if (body.contains(delete)) {
          delete.removeFromTree();
        }
    }
```

```
            }
        }
        // Adiciona as mudanças na AST.
        addChangeFromModifiedAST(file, pm);
    }

    @Override
    protected void doCreateChange(IProgressMonitor pm) throws CoreException,
            OperationCanceledException {
        // A mudança é feita no método makeChangesTo(...).
    }
}
```

A.2 *Remove Unused Variables*

A.2.1 Interface com o usuário (*RemoveUnusedVariablesAction.java*)

Nesse código fonte estão contidas duas classes, sendo que uma é responsável por receber a chamada do usuário e associar a ação de refatoração com seu respectivo assistente e a outra é o próprio assistente (*wizard*) da refatoração.

```
package org.eclipse.photran.internal.refactoring.ui;

import java.util.ArrayList;

import org.eclipse.core.resources.IFile;
import org.eclipse.ltk.ui.refactoring.UserInputWizardPage;
import org.eclipse.photran.internal.core.refactoring.RemoveUnusedVariablesRefactoring;
import org.eclipse.photran.internal.core.refactoring.infrastructure.AbstractFortranRefactoring;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.ui.IEditorActionDelegate;
import org.eclipse.ui.IWorkbenchWindowActionDelegate;

public class RemoveUnusedVariablesAction extends AbstractFortranRefactoringActionDelegate
        implements IWorkbenchWindowActionDelegate, IEditorActionDelegate {

    public RemoveUnusedVariablesAction() {
        super(RemoveUnusedVariablesRefactoring.class,
                FortranRemoveUnusedVariablesRefactoringWizard.class);
    }

    @Override
    protected AbstractFortranRefactoring getRefactoring(ArrayList<IFile> files) {
        return new RemoveUnusedVariablesRefactoring(files);
    }
```

```
    public static class FortranRemoveUnusedVariablesRefactoringWizard extends
        AbstractFortranRefactoringWizard {

      protected RemoveUnusedVariablesRefactoring removeUnusedVariablesRefactoring;

      public FortranRemoveUnusedVariablesRefactoringWizard(
          RemoveUnusedVariablesRefactoring r) {
        super(r);
        this.removeUnusedVariablesRefactoring = r;
      }

      @Override
      protected void doAddUserInputPages() {

        addPage(new UserInputWizardPage(removeUnusedVariablesRefactoring.getName()) {

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

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

            Label lbl = new Label(top, SWT.NONE);
            lbl.setText("Click␣OK␣to␣remove␣unused␣variables␣in␣the␣selected␣files.␣\
                nTo␣see␣what␣changes␣will␣be␣made,␣click␣Preview.\n\nWARNING:␣
                If␣the␣files␣have␣some␣change␣so␣do␣the␣refactoring␣again␣\nuntil␣no
                ␣change␣appears.␣So␣all␣the␣unused␣variables␣are␣removed.");
          }
        });
      }
    }
}
```

A.2.2 Arquivo de ação da refatoração (*RemoveUnusedVariablesRefactoring.java*)

Nesse código fonte está a principal classe a ser codificada, que consiste na ação propriamente dita da refatoração.

```
package org.eclipse.photran.internal.core.refactoring;

import java.util.ArrayList;
import java.util.List;
import java.util.Set;

import org.eclipse.core.resources.IFile;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.OperationCanceledException;
import org.eclipse.ltk.core.refactoring.RefactoringStatus;
import org.eclipse.photran.core.IFortranAST;
```

```
import org.eclipse.photran.core.vpg.PhotranTokenRef;
import org.eclipse.photran.internal.core.analysis.binding.Definition;
import org.eclipse.photran.internal.core.analysis.binding.ScopingNode;
import org.eclipse.photran.internal.core.parser.ASTEntityDeclNode;
import org.eclipse.photran.internal.core.parser.ASTExecutableProgramNode;
import org.eclipse.photran.internal.core.parser.ASTObjectNameNode;
import org.eclipse.photran.internal.core.parser.ASTTypeDeclarationStmtNode;
import org.eclipse.photran.internal.core.parser.Parser.IASTListNode;
import org.eclipse.photran.internal.core.parser.Parser.IASTNode;
import org.eclipse.photran.internal.core.refactoring.infrastructure.
    MultipleFileFortranRefactoring;

/**
 * Refatoração para remover variáveis não utilizadas no código.
 *
 * @author Gustavo Rissetti
 **/

public class RemoveUnusedVariablesRefactoring extends MultipleFileFortranRefactoring {

    public RemoveUnusedVariablesRefactoring(ArrayList<IFile> myFiles) {
        super(myFiles);
    }

    @Override
    public String getName() {
        return "Remove␣Unused␣Variables";
    }

    @Override
    protected void doCheckInitialConditions(RefactoringStatus status, IProgressMonitor pm)
        throws PreconditionFailure {
        ensureProjectHasRefactoringEnabled(status);
        removeFixedFormFilesFrom(this.selectedFiles, status);
        // Essa refatoração tem como pré−requisito que o código seja Implicit None.
        // É necessário primeiramente usar a refatoração Introduce Implicit None do Photran.
        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.");
                }
                List<ScopingNode> scopes = ast.getRoot().getAllContainedScopes();
                for (ScopingNode scope : scopes) {
                    if (!(scope instanceof ASTExecutableProgramNode)) {
                        if (!scope.isImplicitNone()) {
                            fail("All␣of␣the␣selected␣files␣must␣be␣'Implict␣None'!␣Please␣use␣
                                the␣'Introduce␣Implict␣None␣Refactoring'␣first␣to␣introduce␣the␣'
                                Implict␣None'␣statements␣in␣file␣" + file.getName() + "!");
                        }
```

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

@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();
    }
}

private ASTTypeDeclarationStmtNode getTypeDeclarationStmtNode(IASTNode node) {
    if (node == null) {
        return null;
    }
    if (node instanceof ASTTypeDeclarationStmtNode) {
        return (ASTTypeDeclarationStmtNode) node;
    }
    return getTypeDeclarationStmtNode(node.getParent());
}

private void makeChangesTo(IFile file, IFortranAST ast, RefactoringStatus status,
      IProgressMonitor pm) throws PreconditionFailure {
    boolean hasChanged = false;
    List<ScopingNode> scopes = ast.getRoot().getAllContainedScopes();
    for (ScopingNode scope : scopes) {
        System.out.println("Scope:␣" + scope.getClass().getName());
        List<Definition> definitions = scope.getAllDefinitions();
        for (Definition def : definitions) {
            if (def.isLocalVariable()) {
                Set<PhotranTokenRef> references = def.findAllReferences(true);
                // Se a variável não tiver sido referenciada ao longo do código,
                // então ela nunca foi usada, e deverá, portanto, ser removida.
                if (references.isEmpty()) {
                    hasChanged = true;
```

```
                        System.out.println("The␣variable␣[" + def.getDeclaredName() + "]␣was␣not␣
                            used␣and␣will␣be␣removed.");
                        ASTTypeDeclarationStmtNode declarationNode =
                            getTypeDeclarationStmtNode(def.getTokenRef().findToken().getParent());
                        if (declarationNode.getEntityDeclList().size() == 1) {
                            declarationNode.replaceWith("\n");
                        } else {
                            IASTListNode<ASTEntityDeclNode> statementsInNode =
                                declarationNode.getEntityDeclList();
                            for (ASTEntityDeclNode statement : statementsInNode) {
                                ASTObjectNameNode objectName = statement.getObjectName();
                                String statementName = objectName.getObjectName().getText();
                                if (statementName.equals(def.getDeclaredName())) {
                                    if (!statementsInNode.remove(statement)) {
                                        fail("Sorry,␣could␣not␣complete␣the␣operation.");
                                    }
                                    break;
                                }
                            }
                            declarationNode.setEntityDeclList(statementsInNode);
                        }
                    }
                }
            }
        }
        if (hasChanged) {
            addChangeFromModifiedAST(file, pm);
            status.addInfo("After␣clicking␣'Continue',␣do␣the␣same␣refactoring␣again␣to␣
                make␣sure␣that␣all␣unused␣variables␣are␣removed␣from␣file␣" + file.getName
                () + "!");
        } else {
            status.addInfo("All␣unused␣variables␣have␣been␣removed␣from␣file␣" + file.
                getName() + "!");
        }
    }

    @Override
    protected void doCreateChange(IProgressMonitor pm) throws CoreException,
        OperationCanceledException {
        // A mudança é feita no método makeChangesTo(...).
    }
}
```

A.3 *Data To Parameter*

A.3.1 Interface com o usuário (*DataToParameterAction.java*)

Nesse código fonte estão contidas duas classes, sendo que uma é responsável por receber a chamada do usuário e associar a ação de refatoração com seu respectivo assistente

e a outra é o próprio assistente (*wizard*) da refatoração.

```
package org.eclipse.photran.internal.refactoring.ui;

import java.util.ArrayList;

import org.eclipse.core.resources.IFile;
import org.eclipse.ltk.ui.refactoring.UserInputWizardPage;
import org.eclipse.photran.internal.core.refactoring.DataToParameterRefactoring;
import org.eclipse.photran.internal.core.refactoring.infrastructure.AbstractFortranRefactoring;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.ui.IEditorActionDelegate;
import org.eclipse.ui.IWorkbenchWindowActionDelegate;

public class DataToParameterAction extends AbstractFortranRefactoringActionDelegate
    implements IWorkbenchWindowActionDelegate, IEditorActionDelegate {

  public DataToParameterAction() {
    super(DataToParameterRefactoring.class, FortranDataToParameterRefactoringWizard.
        class);
  }

  @Override
  protected AbstractFortranRefactoring getRefactoring(ArrayList<IFile> files) {
    return new DataToParameterRefactoring(files);
  }

  public static class FortranDataToParameterRefactoringWizard extends
      AbstractFortranRefactoringWizard {

    protected DataToParameterRefactoring dataToParameterRefactoring;

    public FortranDataToParameterRefactoringWizard(DataToParameterRefactoring r) {
      super(r);
      this.dataToParameterRefactoring = r;
    }

    @Override
    protected void doAddUserInputPages() {
      addPage(new UserInputWizardPage(dataToParameterRefactoring.getName()) {

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

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

          Label lbl = new Label(top, SWT.NONE);
```

```
                lbl.setText("Click␣OK␣to␣change␣the␣attributes␣Data␣by␣attributes␣
                    Parameter␣in␣the␣selected␣files.␣To␣see␣what␣changes␣will␣be␣made,␣
                    click␣Preview.");
            }
        });
    }
  }
}
```

A.3.2 Arquivo de ação da refatoração (*DataToParameterRefactoring.java*)

Nesse código fonte está a principal classe a ser codificada, que consiste na ação propriamente dita da refatoração.

```
package org.eclipse.photran.internal.core.refactoring;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

import org.eclipse.core.resources.IFile;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.OperationCanceledException;
import org.eclipse.ltk.core.refactoring.RefactoringStatus;
import org.eclipse.photran.core.IFortranAST;
import org.eclipse.photran.internal.core.analysis.binding.ScopingNode;
import org.eclipse.photran.internal.core.parser.ASTAssignmentStmtNode;
import org.eclipse.photran.internal.core.parser.ASTDataStmtNode;
import org.eclipse.photran.internal.core.parser.ASTDataStmtSetNode;
import org.eclipse.photran.internal.core.parser.ASTDataStmtValueNode;
import org.eclipse.photran.internal.core.parser.ASTDatalistNode;
import org.eclipse.photran.internal.core.parser.ASTDerivedTypeDefNode;
import org.eclipse.photran.internal.core.parser.ASTExecutableProgramNode;
import org.eclipse.photran.internal.core.parser.IDataStmtObject;
import org.eclipse.photran.internal.core.parser.Parser.IASTListNode;
import org.eclipse.photran.internal.core.parser.Parser.IASTNode;
import org.eclipse.photran.internal.core.refactoring.infrastructure.
    MultipleFileFortranRefactoring;
import org.eclipse.photran.internal.core.refactoring.infrastructure.Reindenter;
import org.eclipse.photran.internal.core.refactoring.infrastructure.SourcePrinter;

/**
 * Refatoração para transformar declarações do tipo Data em declarações do tipo Parameter.
 *
 * @author Gustavo Rissetti
 **/

public class DataToParameterRefactoring extends MultipleFileFortranRefactoring {

    public DataToParameterRefactoring(ArrayList<IFile> myFiles) {
```

```
    super(myFiles);
}

@Override
public String getName() {
    return "Data␣To␣Parameter";
}

@Override
protected void doCheckInitialConditions(RefactoringStatus status, IProgressMonitor pm)
      throws PreconditionFailure {
    ensureProjectHasRefactoringEnabled(status);
    removeFixedFormFilesFrom(this.selectedFiles, status);
}

@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();
    }
}

private void makeChangesTo(IFile file, IFortranAST ast, RefactoringStatus status,
      IProgressMonitor pm) throws PreconditionFailure {
    List<String> astAssignmentStmtNames = new LinkedList<String>();
    List<IASTNode> parameter_nodes = new LinkedList<IASTNode>();
    List<IASTNode> nodes_to_delete = new LinkedList<IASTNode>();
    List<ASTDataStmtValueNode> values_to_delete = new LinkedList<
        ASTDataStmtValueNode>();
    List<IDataStmtObject> statements_to_delete = new LinkedList<IDataStmtObject>();
    List<ScopingNode> scopes = ast.getRoot().getAllContainedScopes();
    boolean hasChanged = false;
    for (ScopingNode scope : scopes) {
      astAssignmentStmtNames.clear();
      parameter_nodes.clear();
      nodes_to_delete.clear();
      values_to_delete.clear();
      statements_to_delete.clear();
      if (!(scope instanceof ASTExecutableProgramNode) && !(scope instanceof
          ASTDerivedTypeDefNode)) {
```

```
IASTListNode<IASTNode> body = (IASTListNode<IASTNode>) scope.getBody();
// Para poder transformar uma declaração do tipo: data nome / valor /
// em uma declaração do tipo: parameter ( nome = valor ), a variável
// [nome] não pode ser alterada no programa, podendo ser apenas
// referenciada. Assim, inicialmente, faz−se uma verificação de todas
// as variáveis que tiveram seu valor alterado, para ter certeza na
// hora de fazer a troca, de que nenhuma das variáveis do tipo Data
// alteradas seja transformada em Parameter. Caso uma das variáveis
// do tipo Data tiver sido alterada, ela permanece sendo do tipo Data.
for (IASTNode node : body) {
    if (node instanceof ASTAssignmentStmtNode) {
        String name = ((ASTAssignmentStmtNode) node).getLhsVariable().getName()
                .getText();
        astAssignmentStmtNames.add(name);
    }
}
for (IASTNode node : body) {
    if (node instanceof ASTDataStmtNode) {
        IASTListNode<ASTDatalistNode> data_list = ((ASTDataStmtNode) node).
                getDatalist();
        int data_list_size = data_list.size();
        for (ASTDatalistNode data_node : data_list) {
            ASTDataStmtSetNode stmt_set_node = data_node.getDataStmtSet();
            IASTListNode<IDataStmtObject> statement_list = stmt_set_node.
                    getDataStmtObjectList();
            IASTListNode<ASTDataStmtValueNode> value_list = stmt_set_node.
                    getDataStmtValueList();
            int statement_count = 0;
            values_to_delete.clear();
            statements_to_delete.clear();
            for (statement_count = 0; statement_count < statement_list.size();
                    statement_count++) {
                String parameter_name = statement_list.get(statement_count).toString().
                        trim();
                // Novo nó que conterá a declaração do tipo Parameter.
                IASTNode parameter = null;
                // Se a variável não foi alterada, então ela será
                // transformada em Parameter.
                if (!astAssignmentStmtNames.contains(parameter_name)) {
                    hasChanged = true;
                    // Pega o código do nó.
                    String source = SourcePrinter.getSourceCodeFromASTNode(node);
                    String[] source_split = source.split("\n");
                    // Nova declaração.
                    StringBuffer parameter_statement = new StringBuffer("parameter␣(␣
                            ");
                    parameter_statement.append(parameter_name + "␣=␣");
                    String value = value_list.get(statement_count).getConstant().toString
                            ().trim();
                    parameter_statement.append(value + "␣)");
                    // Pega os comentários do final da linha da declaração.
```

```
            String comments_end_of_line = source_split[source_split.length -
                1];
            boolean has_comment = false;
            int index_comment = 0;
            for (index_comment = 0; index_comment < comments_end_of_line.
                length(); index_comment++) {
                if (comments_end_of_line.charAt(index_comment) == '!') {
                    has_comment = true;
                    break;
                }
            }
            if (has_comment) {
                parameter_statement.append(" " + comments_end_of_line.
                    substring(index_comment));
            }
            // É criado o novo nó.
            parameter = parseLiteralStatement(parameter_statement.toString());
            // Referência do local onde o novo nó deve ser inserido na AST.
            parameter_nodes.add(node);
            // Novo nó a ser inserido na AST.
            parameter_nodes.add(parameter);
            // Deve ser removido um valor e uma declaração das listas originais.
            ASTDataStmtValueNode value_to_remove = value_list.get(
                statement_count);
            IDataStmtObject statement_to_remove = statement_list.get(
                statement_count);
            values_to_delete.add(value_to_remove);
            statements_to_delete.add(statement_to_remove);
        }
    }
    // Remoção das entradas da AST.
    if (statement_list.size() == statements_to_delete.size()) {
        data_node.removeFromTree();
        data_list_size--;
        // Se um nó tiver todos os seus dados removidos, é necessário recuperar
        // os comentários que estavam antes do mesmo e recolocá-los em seu
        // devido lugar.
        if (data_list_size == 0) {
            IASTNode parameter = null;
            String source = SourcePrinter.getSourceCodeFromASTNode(node);
            String[] source_split = source.split("\n");
            String comments_before_line = new String("");
            for (int i = 0; i < source_split.length - 1; i++) {
                comments_before_line += source_split[i] + "\n";
            }
            IASTNode last_parameter_node = parameter_nodes.get(
                parameter_nodes.size() - 1);
            String last_parameter_node_source = SourcePrinter.
                getSourceCodeFromASTNode(last_parameter_node);
            // Adicionados os comentários.
```

```
                    String last_parameter_with_comments = comments_before_line +
                        last_parameter_node_source;
                    // Gerado o novo nó.
                    parameter = parseLiteralStatement(last_parameter_with_comments);
                    // É substituido da lista o último nó pelo novo nó com os
                    // comentários e o valor do antigo nó.
                    parameter_nodes.remove(parameter_nodes.size() - 1);
                    parameter_nodes.add(parameter);
                    // É adicionado na lista de nós a serem removidos o nó vazio.
                    nodes_to_delete.add(node);
                }
            } else {
                statement_list.removeAll(statements_to_delete);
                value_list.removeAll(values_to_delete);
            }
        }
    }
}
// Insere todos os nós do tipo Parameter criados.
for (int i = 0; i < parameter_nodes.size(); i += 2) {
    body.insertAfter(parameter_nodes.get(i), parameter_nodes.get(i + 1));
    Reindenter.reindent(parameter_nodes.get(i + 1), ast);
}
// Deleta os nós do tipo Data que ficaram vazios.
for (int i = 0; i < nodes_to_delete.size(); i++) {
    ASTDataStmtNode delete = (ASTDataStmtNode) nodes_to_delete.get(i);
    if (body.contains(delete)) {
        delete.removeFromTree();
    }
}
// Caso alguma declaração tenha ficado como: data ,var/valor/
// é necessário retirar a vírgula que sobrou depois do termo Data.
IASTNode coma = null;
for (IASTNode node : body) {
    if (node instanceof ASTDataStmtNode) {
        coma = node;
        String source_coma = SourcePrinter.getSourceCodeFromASTNode(coma);
        String[] source_coma_split = source_coma.split("\n");
        // Acha a declaração Data.
        String statement = source_coma_split[source_coma_split.length - 1].trim();
        String data = statement.substring(0, 4);
        String list_data = statement.substring(4);
        list_data = list_data.trim();
        // Se iniciar com uma ",", então tem que retirá-la.
        if (list_data.startsWith(",")) {
            // Retira a vírgula que está sobrando.
            list_data = list_data.substring(1);
            list_data = list_data.trim();
            String new_source = new String("");
            for (int i = 0; i < source_coma_split.length - 1; i++) {
                new_source += source_coma_split[i] + "\n";
```

```
                        }
                        new_source += data + "␣" + list_data;
                        // Cria o novo nó e substitui pelo antigo.
                        IASTNode without_coma = parseLiteralStatement(new_source);
                        coma.replaceWith(without_coma);
                        Reindenter.reindent(without_coma, ast);
                    }
                }
            }
        }
        // Adiciona as mudanças na AST.
        if (hasChanged) {
            addChangeFromModifiedAST(file, pm);
        }
    }

    @Override
    protected void doCreateChange(IProgressMonitor pm) throws CoreException,
            OperationCanceledException {
        // A mudança é feita no método makeChangesTo(...).
    }
}
```

A.4 *Introduce INTENT*

Essa refatoração foi desenvolvida durante o mestrado de Bruno Batista Boniati (BONIATI, 2009). Ela foi desenvolvida em uma versão anterior do *plugin*, e na última atualização, com as mudanças estruturais do mesmo, a refatoração parou de funcionar. Foi feita então uma atualização da refatoração para que ela funcionasse na nova versão do Photran, fazendo-se também pequenas melhorias no código.

A.4.1 Interface com o usuário (*IntroduceIntentAction.java*)

Nesse código fonte estão contidas duas classes, sendo que uma é responsável por receber a chamada do usuário e associar a ação de refatoração com seu respectivo assistente e a outra é o próprio assistente (*wizard*) da refatoração.

```
package org.eclipse.photran.internal.refactoring.ui;

import java.util.ArrayList;

import org.eclipse.core.resources.IFile;
import org.eclipse.ltk.ui.refactoring.UserInputWizardPage;
import org.eclipse.photran.internal.core.refactoring.IntroduceIntentRefactoring;
import org.eclipse.photran.internal.core.refactoring.infrastructure.AbstractFortranRefactoring;
import org.eclipse.swt.SWT;
```

```
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.ui.IEditorActionDelegate;
import org.eclipse.ui.IWorkbenchWindowActionDelegate;

public class IntroduceIntentAction extends AbstractFortranRefactoringActionDelegate
    implements IWorkbenchWindowActionDelegate, IEditorActionDelegate {

  public IntroduceIntentAction() {
    super(IntroduceIntentRefactoring.class, FortranIntroduceIntentRefactoringWizard.class);
  }

  public static class FortranIntroduceIntentRefactoringWizard extends
      AbstractFortranRefactoringWizard {

    protected IntroduceIntentRefactoring introduceIntentRefactoring;

    public FortranIntroduceIntentRefactoringWizard(IntroduceIntentRefactoring r) {
      super(r);
      this.introduceIntentRefactoring = r;
    }

    @Override
    protected void doAddUserInputPages() {
      addPage(new UserInputWizardPage(introduceIntentRefactoring.getName()) {

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

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

          Label lbl = new Label(top, SWT.NONE);
          lbl.setText("Click␣OK␣to␣introduce␣intents␣in␣the␣selected␣subroutine/
              function.␣To␣see␣what␣changes␣will␣be␣made,␣click␣Preview.");
        }
      });
    }
  }

  @Override
  protected AbstractFortranRefactoring getRefactoring(ArrayList<IFile> files) {
    return new IntroduceIntentRefactoring(getFortranEditor().getIFile(), getFortranEditor().
        getSelection());
  }
}
```

A.4.2 Arquivo de ação da refatoração (*IntroduceIntentRefactoring.java*)

Nesse código fonte está a principal classe a ser codificada, que consiste na ação propriamente dita da refatoração.

```
package org.eclipse.photran.internal.core.refactoring;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

import org.eclipse.core.resources.IFile;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.OperationCanceledException;
import org.eclipse.jface.text.ITextSelection;
import org.eclipse.ltk.core.refactoring.RefactoringStatus;
import org.eclipse.photran.core.vpg.PhotranTokenRef;
import org.eclipse.photran.internal.core.analysis.binding.Definition;
import org.eclipse.photran.internal.core.analysis.binding.ScopingNode;
import org.eclipse.photran.internal.core.lexer.Token;
import org.eclipse.photran.internal.core.parser.ASTAssignmentStmtNode;
import org.eclipse.photran.internal.core.parser.ASTAttrSpecNode;
import org.eclipse.photran.internal.core.parser.ASTCallStmtNode;
import org.eclipse.photran.internal.core.parser.ASTDataRefNode;
import org.eclipse.photran.internal.core.parser.ASTFunctionSubprogramNode;
import org.eclipse.photran.internal.core.parser.ASTNameNode;
import org.eclipse.photran.internal.core.parser.ASTObjectNameNode;
import org.eclipse.photran.internal.core.parser.ASTReadStmtNode;
import org.eclipse.photran.internal.core.parser.ASTSubroutineArgNode;
import org.eclipse.photran.internal.core.parser.ASTSubroutineSubprogramNode;
import org.eclipse.photran.internal.core.parser.ASTTypeDeclarationStmtNode;
import org.eclipse.photran.internal.core.parser.ASTVarOrFnRefNode;
import org.eclipse.photran.internal.core.parser.Parser.IASTListNode;
import org.eclipse.photran.internal.core.parser.Parser.IASTNode;
import org.eclipse.photran.internal.core.refactoring.infrastructure.SingleFileFortranRefactoring;
import org.eclipse.photran.internal.core.refactoring.infrastructure.SourcePrinter;

/**
 * Refatoração para introduzir os Intents em Subrotinas e Funções.
 *
 * @author Bruno Batista Boniati
 * @author Gustavo Rissetti
 **/

public class IntroduceIntentRefactoring extends SingleFileFortranRefactoring {

    private ScopingNode subprogram = null;
    List<Definition> argumentDefinitions;
    List<SubProgramInformation> subProgramInterfaces;
    List<SubProgramInformation> callProgram;
```

```
@Override
public String getName() {
    return "Introduce␣Intent";
}

public boolean subprogramExists(String name, String file) {
    boolean r = false;
    name = name.toUpperCase();
    file = file.toUpperCase();
    for (int i = 0; i < subProgramInterfaces.size(); i++) {
      String n_aux = subProgramInterfaces.get(i).getSubProgramName();
      String f_aux = subProgramInterfaces.get(i).getFileName();
      if ((n_aux.equalsIgnoreCase(name)) && (f_aux.equalsIgnoreCase(file))) {
          r = true;
          break;
      }
    }
    return r;
}

private ASTTypeDeclarationStmtNode getTypeDeclarationStmtNode(IASTNode o) {
    if (o == null) {
      return null;
    }
    if (o instanceof ASTTypeDeclarationStmtNode) {
      return (ASTTypeDeclarationStmtNode) o;
    }
    return getTypeDeclarationStmtNode(o.getParent());
}

private void processInterfaces(IFile file) throws PreconditionFailure {
    List<ScopingNode> scopes = vpg.acquirePermanentAST(file).getRoot().
        getAllContainedScopes();
    for (ScopingNode scopingNode : scopes) {
      List<Definition> definitions = scopingNode.getAllDefinitions();
      for (Definition def : definitions) {
          if (def.isSubprogramArgument()) {
              PhotranTokenRef tr = def.getTokenRef();
              Token t = tr.findToken();
              ScopingNode s = t.getEnclosingScope();
              if (s != null) {
                String subprogram = s.getRepresentativeToken().getText();
                String fileName = def.getTokenRef().getFilename();
                if (!subprogramExists(subprogram, fileName)) {
                    SubProgramInformation spi = new SubProgramInformation(subprogram,
                        fileName);
                    List<Definition> def_scope = s.getAllDefinitions();
                    for (Definition d : def_scope) {
                        if (d.isSubprogramArgument()) {
```

```
                        ASTTypeDeclarationStmtNode a = getTypeDeclarationStmtNode(d.
                            getTokenRef().findToken().getParent());
                        // Só atua se a declaração for simples.
                        if (a.getEntityDeclList().size() == 1) {
                            ASTAttrSpecNode at = getASTAttrSpecNode(a);
                            spi.addParam(d.getDeclaredName(), at);
                        } else {
                            // Se alguma declaração for composta, é necessário padronizá-la
                            // usando a refatoração Standardize Statements.
                            fail("All␣Statements␣must␣be␣simple␣(integer␣::␣a).␣To␣
                                Standardize␣all␣\n" +
                                 "statements␣in␣file,␣use␣Standardize␣Statements␣
                                     Refactoring␣first!");
                        }
                    }
                }
                subProgramInterfaces.add(spi);
            }
        }
    }
  }
}

private ASTAttrSpecNode getASTAttrSpecNode(IASTNode o) {
    ASTAttrSpecNode r = null;
    if (o instanceof ASTAttrSpecNode) {
      r = (ASTAttrSpecNode) o;
    } else {
      for (IASTNode child : o.getChildren()) {
          if (r == null) {
              r = getASTAttrSpecNode(child);
          }
      }
    }
    return r;
}

public IntroduceIntentRefactoring(IFile file, ITextSelection selection) {
    super(file, selection);
    subProgramInterfaces = new ArrayList<SubProgramInformation>();
    callProgram = new ArrayList<SubProgramInformation>();
}

@Override
protected void doCheckInitialConditions(RefactoringStatus status, IProgressMonitor pm)
     throws PreconditionFailure {
    ensureProjectHasRefactoringEnabled(status);
    processInterfaces(fileInEditor);
    IASTNode n = findEnclosingNode(astOfFileInEditor, selectedRegionInEditor);
    if (n != null) {
```

```
        if ((n instanceof ASTSubroutineSubprogramNode) | (n instanceof
             ASTFunctionSubprogramNode)) {
            subprogram = (ScopingNode) n;
        }
    }
    if (subprogram == null) {
        fail("You␣can␣select␣a␣Subroutine␣or␣Subprogram");
    } else {
        if (!subprogram.isImplicitNone()) {
            fail("The␣subprogram␣selected␣must␣be␣'Implict␣None'!␣Please␣use␣the␣'
                 Introduce␣Implict␣None␣Refactoring'␣first!");
        }
        List<Definition> definitions = subprogram.getAllDefinitions();
        argumentDefinitions = new LinkedList<Definition>();
        for (Definition def : definitions) {
            if (def.isSubprogramArgument()) {
                argumentDefinitions.add(def);
            }
        }
        for (int i = 0; i < subprogram.getBody().size(); i++) {
            processCallsAndFunctions(subprogram.getBody().get(i));
        }
        processCallsAndFunctionsWithInterfaces();
    }
}

private void processCallsAndFunctions(IASTNode node) {
    // Funções.
    if (node instanceof ASTVarOrFnRefNode) {
        if (((ASTVarOrFnRefNode) node).getPrimarySectionSubscriptList() != null) {
            SubProgramInformation spi = new SubProgramInformation(((ASTVarOrFnRefNode
                 ) node).getName().getName().getText(), fileInEditor.getName());
            for (int i = 0; i < ((ASTVarOrFnRefNode) node).getPrimarySectionSubscriptList().
                 size(); i++) {
                if (((ASTVarOrFnRefNode) node).getPrimarySectionSubscriptList().get(i).
                     getExpr() instanceof ASTVarOrFnRefNode) {
                    spi.addParam(((ASTVarOrFnRefNode) ((ASTVarOrFnRefNode) node).
                         getPrimarySectionSubscriptList().get(i).getExpr()).getName().getName().
                         getText(), null);
                }
            }
            callProgram.add(spi);
        }
    } else {
        // Subrotinas.
        if (node instanceof ASTCallStmtNode) {
            SubProgramInformation spi = new SubProgramInformation(((ASTCallStmtNode)
                 node).getSubroutineName().getText(), fileInEditor.getName());
            IASTListNode<ASTSubroutineArgNode> argumentos = ((ASTCallStmtNode) node
                 ).getArgList();
            for (ASTSubroutineArgNode arg : argumentos) {
```

```
                ASTVarOrFnRefNode exp = (ASTVarOrFnRefNode) arg.getExpr();
                spi.addParam(exp.getName().getName().getText(), null);
            }
            callProgram.add(spi);
        }
    }
    for (IASTNode child : node.getChildren()) {
        processCallsAndFunctions(child);
    }
}

// Este método vai percorrer todas as chamadas a métodos dentro da subrotina e verificar nos
// escopos de função como são os intents.
private void processCallsAndFunctionsWithInterfaces() {
    for (int i = 0; i < callProgram.size(); i++) {
        for (int j = 0; j < subProgramInterfaces.size(); j++) {
            if ((callProgram.get(i).getSubProgramName().equalsIgnoreCase(
                    subProgramInterfaces.get(j).getSubProgramName())) &&
                    (callProgram.get(i).getParamsSize() == subProgramInterfaces.get(j).
                            getParamsSize())) {
                for (int p = 0; p < subProgramInterfaces.get(j).getParamsSize(); p++) {
                    switch (subProgramInterfaces.get(j).getIntentParam(p)) {
                        case SubProgramInformation.intentINOUT:
                            callProgram.get(i).setIntentInOut(p);
                            break;
                        case SubProgramInformation.intentIN:
                            callProgram.get(i).setIntentIn(p);
                            break;
                        case SubProgramInformation.intentOUT:
                            callProgram.get(i).setIntentOut(p);
                            break;
                    }
                }
                break;
            }
        }
    }
}

@Override
protected void doCheckFinalConditions(RefactoringStatus status, IProgressMonitor pm)
        throws PreconditionFailure {
    // Não faz nada...
}

@Override
protected void doCreateChange(IProgressMonitor pm) throws CoreException,
        OperationCanceledException {
    if (argumentDefinitions != null) {
        for (Definition def : argumentDefinitions) {
            if (def.getTokenRef().findToken() != null) {
```

```
                ASTTypeDeclarationStmtNode a = getTypeDeclarationStmtNode(def.
                    getTokenRef().findToken().getParent());
                if (a.getEntityDeclList().size() == 1) {
                  if (!isIntent(a)) {
                      String name = getVariableName(a);
                      if (name != null) {
                          String s = SourcePrinter.getSourceCodeFromASTNode(a);
                          String twoPoints = "";
                          String intentType = "";
                          String attr = "";
                          if (hasReference(name)) {
                            intentType = "in";
                          }
                          if (hasAssigment(name)) {
                            intentType = intentType.concat("out");
                          }
                          if (a.getAttrSpecSeq() != null) {
                            attr = a.getAttrSpecSeq().toString();
                          }
                          int p = getVariablePosition(s, attr);
                          if (!hasTwoPoints(s.substring(p))) {
                            twoPoints = "::␣";
                          }
                          String s2 = s.substring(0, p);
                          String s3 = s.substring(p + 1, s.length());
                          if (intentType.length() > 0) {
                            intentType = ",␣intent(".concat(intentType).concat(")␣");
                          }
                          s = s2.concat(intentType).concat(twoPoints).concat(s3);
                          a.replaceWith(parseLiteralStatement(s));
                      }
                  }
                }
            }
        }
    }
    addChangeFromModifiedAST(fileInEditor, pm);
    vpg.releaseAST(fileInEditor);
}

// Indica que a variável passada por parâmetro sofre alteração no corpo do procedimento.
private boolean hasAssigment(String name) {
    boolean r = false;
    for (int i = 0; i < callProgram.size(); i++) {
      for (int j = 0; j < ((SubProgramInformation) callProgram.get(i)).getParamsSize(); j++)
          {
          if (name.equalsIgnoreCase(((SubProgramInformation) callProgram.get(i)).
              getParamName(j))) {
            if (((SubProgramInformation) callProgram.get(i)).paramIsIntentOut(j) || ((
                SubProgramInformation) callProgram.get(i)).paramIsIntentInOut(j)) {
              r = true;
```

```
                    break;
                }
            }
        }
    }
    if (!r) {
        for (int i = 0; i < subprogram.getBody().size(); i++) {
            r = hasAssigment(subprogram.getBody().get(i), name);
            if (r) {
                break;
            }
        }
    }
    return r;
}

private boolean hasRead(IASTNode node, String v) {
    boolean r = false;
    if (node instanceof ASTDataRefNode) {
        r = (((ASTDataRefNode) node).getName().getText().equalsIgnoreCase(v));
    } else {
        for (IASTNode child : node.getChildren()) {
            if (!r) {
                r = hasRead(child, v);
            } else {
                break;
            }
        }
    }
    return r;
}

private boolean hasAssigment(IASTNode node, String variavel) {
    boolean r = false;
    if (node instanceof ASTAssignmentStmtNode) {
        r = (((ASTAssignmentStmtNode) node).getLhsVariable().getName().getText().
            equalsIgnoreCase(variavel));
    } else {
        if (node instanceof ASTReadStmtNode) {
            r = hasRead(((ASTReadStmtNode) node).getInputItemList(), variavel);
        }
    }
    if (!r) {
        for (IASTNode child : node.getChildren()) {
            if (!r) {
                r = hasAssigment(child, variavel);
            } else {
                break;
            }
        }
    }
```

```
    return r;
}

// Indica que a variável passada por parâmetro é referenciada no corpo do procedimento.
private boolean hasReference(String name) {
    boolean r = false;
    for (int i = 0; i < subprogram.getBody().size(); i++) {
      r = isReferenced(subprogram.getBody().get(i), name);
      if (r) {
          break;
      }
    }
    return r;
}

private boolean isReferenced(IASTNode node, String name) {
    boolean r = false;
    if (node instanceof ASTVarOrFnRefNode) {
      r = existsReferenceForVariable(node, name);
    } else {
      for (IASTNode child : node.getChildren()) {
          if (!r) {
              r = isReferenced(child, name);
          } else {
              break;
          }
      }
    }
    return r;
}

private boolean existsReferenceForVariable(IASTNode node, String name) {
    boolean r = false;
    if (node instanceof ASTNameNode) {
      if (((ASTNameNode) node).getName().getText().equalsIgnoreCase(name)) {
          r = true;
      }
    } else {
      for (IASTNode child : node.getChildren()) {
          if (!r) {
              r = existsReferenceForVariable(child, name);
          } else {
              break;
          }
      }
    }
    return r;
}

private boolean hasTwoPoints(String s) {
    for (int i = 0; i < s.length() − 1; i++) {
```

```
        char p1 = s.charAt(i);
        char p2 = s.charAt(i + 1);
        if (p1 == '!' || p2 == '!') {
            return false;
        } else if (p1 == ':' && p2 == ':') {
            return true;
        }
    }
    return false;
}

int getVariablePosition(String s, String attr) {
    String[] coments = s.split("\n");
    int size = coments.length - 1;
    int i, j;
    boolean start = false;
    int attr_length = attr.length() - 1;
    for (int k = 0; k < coments.length - 1; k++) {
        size += coments[k].length();
    }
    String statement = coments[coments.length - 1];
    i = 0;
    for (j = 0; j < statement.length(); j++) {
        if ((statement.charAt(j) == '␣') & (start)) {
            i = j;
            break;
        } else {
            if (statement.charAt(j) != '␣') {
                start = true;
            }
        }
    }
    if (attr_length < 0) {
        attr_length = 0;
    }
    return i + size + attr_length;
}

private String getVariableName(IASTNode node) {
    String name = null;
    if (node instanceof ASTObjectNameNode) {
        name = ((ASTObjectNameNode) node).getObjectName().getText();
    } else {
        for (IASTNode child : node.getChildren()) {
            if (name == null) {
                name = getVariableName(child);
            } else {
                break;
            }
        }
    }
```

```
    return name;
}

private boolean isIntent(IASTNode node) {
    boolean r = false;
    if (node instanceof ASTAttrSpecNode) {
      r = ((ASTAttrSpecNode) node).isIntent();
    } else {
      for (IASTNode child : node.getChildren()) {
          if (!r) {
              r = isIntent(child);
          }
      }
    }
    return r;
}

public class SubProgramInformation {
    public static final int intentOFF = -1;
    public static final int intentINOUT = 0;
    public static final int intentIN = 1;
    public static final int intentOUT = 2;
    private String subprogramName;
    private String fileName;
    private List<String> parameters;
    private List<Integer> intentsInfo;

    public SubProgramInformation(String n, String f) {
      subprogramName = n.toUpperCase();
      fileName = f.toUpperCase();
      parameters = new ArrayList<String>();
      intentsInfo = new ArrayList<Integer>();
    }

    public String getSubProgramName() {
      return subprogramName;
    }

    public String getFileName() {
      return fileName;
    }

    public void addParam(String n, ASTAttrSpecNode att) {
      parameters.add(n.toUpperCase());
      if (att != null) {
          if (att.isIntent()) {
              if (att.getIntentSpec().isIntentIn()) {
                intentsInfo.add(new Integer(intentIN));
              } else {
                if (att.getIntentSpec().isIntentOut()) {
                    intentsInfo.add(new Integer(intentOUT));
```

```
                } else {
                    intentsInfo.add(new Integer(intentINOUT));
                }
            }
        } else {
            intentsInfo.add(new Integer(intentINOUT));
        }
    } else {
        intentsInfo.add(new Integer(intentOFF));
    }
}

public int getParamsSize() {
    return parameters.size();
}

public String getParamName(int index) {
    return parameters.get(index);
}

public int getIntentParam(int index) {
    return intentsInfo.get(index).intValue();
}

public void setIntentIn(int index) {
    intentsInfo.set(index, new Integer(intentIN));
}

public void setIntentOut(int index) {
    intentsInfo.set(index, new Integer(intentOUT));
}

public void setIntentInOut(int index) {
    intentsInfo.set(index, new Integer(intentINOUT));
}

public boolean paramIsIntentIn(int index) {
    return intentsInfo.get(index).intValue() == intentIN;
}

public boolean paramIsIntentOut(int index) {
    return intentsInfo.get(index).intValue() == intentOUT;
}

public boolean paramIsIntentInOut(int index) {
    return intentsInfo.get(index).intValue() == intentINOUT;
}
}
}
```