UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

REFATORAÇÃO DE PROGRAMAS FORTRAN DE ALTO DESEMPENHO

DISSERTAÇÃO DE MESTRADO

Bruno Batista Boniati

Santa Maria, RS, Brasil

2009

REFATORAÇÃO DE PROGRAMAS FORTRAN DE ALTO DESEMPENHO

por

Bruno Batista Boniati

Dissertação apresentada ao Programa de Pós-Graduação em Informática
da Universidade Federal de Santa Maria (UFSM, RS), como requisito
parcial para a obtenção do grau de
Mestre em Computação

Orientador: Prof. Dr. Benhur de Oliveira Stein (UFSM)

Co-orientador: Profª Drª Andrea Schwertner Charão (UFSM)

Santa Maria, RS, Brasil

2009

Universidade Federal de Santa Maria
Centro de Tecnologia
Programa de Pós-Graduação em Informática

A Comissão Examinadora, abaixo assinada,
aprova a Dissertação de Mestrado

REFATORAÇÃO DE PROGRAMAS FORTRAN DE ALTO DESEMPENHO

elaborada por
Bruno Batista Boniati

como requisito parcial para obtenção do grau de
Mestre em Computação

COMISSÃO EXAMINADORA:

Profª Drª Andrea Schwertner Charão (UFSM)
(Presidente/Co-orientador)

Prof. Dr. Jairo Panetta (INPE/Petrobras/ITA)

Profª Drª Alice de Jesus Kozakevicius (UFSM)

Santa Maria, 17 de Agosto de 2009.

AGRADECIMENTOS

A minha esposa Silvia Daiana pelo ombro amigo e pelo incentivo.

Ao pequeno Mathias, meu filho, a quem privei da minha presença algumas vezes.

Aos professores Andrea e Benhur pela confiança, orientação e amizade.

Aos colegas da UNIJUÍ que seguraram as pontas nas minhas ausências.

Aos colegas do mestrado e do laboratório de sistemas de computação pela amizade e pela rica troca de experiências.

Ao professor Jairo Panetta por suas contribuições ao trabalho.

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.” — MARTIN FOWLER

RESUMO

Dissertação de Mestrado
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria

REFATORAÇÃO DE PROGRAMAS FORTRAN DE ALTO DESEMPENHO
Autor: Bruno Batista Boniati
Orientador: Prof. Dr. Benhur de Oliveira Stein (UFSM)
Co-orientador: Profª Drª Andrea Schwertner Charão (UFSM)
Local e data da defesa: Santa Maria, 17 de Agosto de 2009.

Refatoração é uma técnica de engenharia de software que consiste em alterar a estrutura interna de uma aplicação sem que tais alterações interfiram nos resultados produzidos pela mesma. Trata-se de uma tarefa permanentemente presente no ciclo de vida de uma aplicação e está diretamente associada às características não funcionais do software, como legibilidade e desempenho. 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 de alto desempenho, a refatoração de código é pouco explorada, principalmente em função de que boa parte do código legado de programas de alto desempenho está escrita em linguagens anteriores ao paradigma da orientação a objetos. A linguagem Fortran (FORmula TRANslator), largamente utilizada em aplicações de alto desempenho, possui poucas e limitadas ferramentas para refatoração de código. Neste contexto, este trabalho explora essa deficiência através da automatização de técnicas de refatoração, utilizando-se do framework da ferramenta Photran (um plugin para edição de código Fortran integrado ao IDE Eclipse). Partindo-se da identificação de oportunidades de refatoração para código 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 impacto no desempenho das mesmas.

Palavras-chave: Alto desempenho, Fortran, refatoração.

ABSTRACT

Master's Dissertation
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria

REFACTORING OF HIGH PERFORMANCE FORTRAN PROGRAMS

Author: Bruno Batista Boniati
Advisor: Prof. Dr. Benhur de Oliveira Stein (UFSM)
Coadvisor: Prof[a] Dr[a] Andrea Schwertner Charão (UFSM)

Refactoring is a software engineering technique that aims at improving the internal structure of an application, in such a way that the changes do not interfere in the results produced by the software. This technique is permanently employed in the software life cycle and refers to non-functional characteristics as legibility and performance. Most refactoring techniques currently apply to object-oriented systems and are widely available in integrated development environments for this programming paradigm. In high performance computing applications, code refactoring is a little-explored technique, as a great amount of legacy high performance code was written before the widespread use of object-oriented languages. The Fortran language (FORmula TRANslator) is heavily used in high performance applications, but has a few and limited tools for code refactoring. In such context, our work explores this open research area through the development of new techniques for refactoring Fortran source code. Our development starts from identifying refactoring opportunities for Fortran code and extends the Photran tool, which is an Eclipse IDE plugin for Fortran programming. The new automated refactoring techniques are applied to third-party code written in Fortran, in order to evaluate their impact on the software performance.

Keywords: high performance, Fortran, refactoring.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AST	Abstract Syntax Tree
BLAS	Basic Linear Algebra Subprograms
BNF	Backus-Naur Form
BRAMS	Brazilian Regional Atmospheric Modeling System
CDT	C, C++ Development Tools
CPTEC	Centro de Previsão de Tempo e Estudos Climáticos
CVS	Concurrent Version System
EBNF	Extended Backus-Naur Form
HPC	High Performance Computing
IBM	International Business Machines
IDE	Integrated Development Environment
IMSL	International Mathematics and Statistics Library
LAPACK	Linear Algebra PACKage
MPI	Message Passing Interface
NAG	Numerical Algorithms Group
OTI	Object Technology International
PTP	Parallel Tools Platform
SAG	Software Architecture Group
SDK	Software Development Kit
TLB	Translation Lookaside Buffer
UI	User Interface
VPG	Virtual Program Graph

SUMÁRIO

1 INTRODUÇÃO

1.1 Contexto e Motivação

Evolução é uma propriedade natural no processo de desenvolvimento de *software*. Em geral, um *software* nasce a partir de um bom projeto e posteriormente é codificado. Durante o ciclo de vida normalmente há necessidade de o *software* evoluir, seja para adicionar um novo requisito, ou para alterar funcionalidades existentes.

O problema é que nem sempre o *software* está preparado para receber novos requisitos ou suportar adaptações nas funcionalidades existentes. Dependendo da forma como as alterações são feitas, acabam por degradar a qualidade e enfraquecer a relação do *software* com o projeto original. Ao longo do processo de desenvolvimento de *software*, é comum que mudanças em requisitos e/ou a não compreensão dos mesmos pelos programadores favoreçam a presença de código mal escrito. Este código pode ser transformado e otimizado de forma a tornar-se mais legível e eficiente.

Refatoração (do inglês *refactoring*) é uma técnica que consiste na aplicação de melhorias na estrutura interna de programas de computador sem que isso afete os resultados produzidos pela mesma (FOWLER et al., 1999; OPDYKE, 1992). Tais técnicas são empregadas com o intuito de evitar a degradação do *software* durante seu ciclo de vida. Refatorar o código não é uma prática nova, pois aplicar melhorias e reestruturações em programas de computador é um processo implícito e contínuo à tarefa de desenvolver *software* (ARNOLD, 1986; TOURWÉ; MENS, 2004).

O termo refatoração está diretamente ligado ao paradigma da orientação a objetos e está presente de forma automatizada em várias ferramentas alinhadas com este paradigma. Na computação de alto desempenho (*High Performance Computing* - HPC) a refatoração de código é uma técnica pouco explorada, principalmente em função de que boa parte do código legado de tais aplicações está escrita em linguagens anteriores ao paradigma da

orientação a objetos (JOHNSON et al., 2006).

Este é o caso de Fortran, uma linguagem estruturada cuja gramática original já tem mais de cinquenta anos e que até os dias de hoje é amplamente utilizada em aplicações científicas, que em geral exigem alto desempenho computacional. A lacuna existente entre a grande quantidade de código legado escrito em Fortran e o reduzido número de ferramentas integradas de desenvolvimento (*Integrated Development Environment* - IDE) que automatizem técnicas de refatoração para Fortran é a principal motivação deste trabalho.

1.2 Objetivos e Contribuição

A catalogação de técnicas de refatoração é uma atividade que em geral pode ser desenvolvida de forma genérica. É possível (em linguagem natural) descrever os objetivos, pré-requisitos, limitações e a própria mecânica de funcionamento de uma técnica de refatoração sem se preocupar exaustivamente com o paradigma ou a linguagem de programação na qual a aplicação a ser refatorada foi escrita (DE, 2004).

Por outro lado, a automatização (implementação) de uma técnica de refatoração deve obrigatoriamente se preocupar com detalhes sintáticos e semânticos da linguagem de programação na qual a aplicação foi escrita. Da mesma forma, características específicas do paradigma de programação devem ser consideradas (GARRIDO; JOHNSON, 2002; ROBERTS; BRANT; JOHNSON, 1997). Refatorar com apoio de uma ferramenta, embora não se aplique a toda e qualquer refatoração, tem algumas vantagens, entre elas a possibilidade de se desfazer uma ação e a dispensa de testes. A utilização de ferramentas automatizadas para refatorar reduz o risco de erros e inconsistências, além de reduzir também o trabalho e consequentemente o custo de desenvolvimento (FOWLER et al., 1999).

A refatoração automatizada por ferramentas torna a ação de refatorar cada vez menos separada da atividade de programar. Refatorar de forma manual é uma atividade cara, que por vezes, justificada pelo seu alto custo, não é feita. O objetivo de se automatizar técnicas de refatoração é de reduzir o custo do processo de reestruturação de código ao ponto em que refatorar seja tão natural quanto usar uma opção para aumentar ou diminuir o tamanho da fonte no editor de código.

A contribuição deste trabalho objetiva identificar, automatizar e aplicar técnicas de refatoração em aplicações de alto desempenho escritas em linguagem Fortran, em especial

refatorações que representem melhores construções com vistas ao ganho de desempenho em relação a suas construções originais. O trabalho explora melhorias não funcionais ligadas ao desempenho (“melhores práticas”) e a evolução de aplicações Fortran.

Para alcançar estes objetivos, o trabalho faz uso do *framework* do Photran (DRAGANCHIRILA, 2004; EIPE, 2004; OVERBEY et al., 2005), um *plugin* integrado ao IDE do Eclipse que atua sobre código Fortran e que disponibiliza uma infra-estrutura básica para *parsing* de código Fortran e manipulação da árvore sintática do código fonte. As técnicas identificadas são automatizadas e integradas à ferramenta Photran. A fim de avaliar a utilização das técnicas implementadas e da ferramenta, assim como o impacto de melhorias não funcionais, uma aplicação de alto-desempenho é utilizada como estudo de caso e os resultados obtidos são tabulados e apresentados ao final do trabalho.

1.3 Organização do Texto

Este trabalho está organizado da seguinte forma: o capítulo 2 apresenta uma revisão bibliográfica acerca do tema central do trabalho: refatoração de programas Fortran. Neste capítulo são abordados conceitos e objetivos de técnicas de refatoração, assim como aspectos ligados à sua utilização em paradigmas não orientados a objetos. Ainda no capítulo 2, o trabalho apresenta os recursos e características da ferramenta Photran, bem como os requisitos para desenvolvimento e integração de ações de refatoração à mesma.

O capítulo 3 aprofunda aspectos mais técnicos ligados à ferramenta Photran e detalha o processo de implementação de algumas técnicas que foram identificadas durante a realização do trabalho. Para cada técnica descreve-se sua motivação, seus requisitos, sua mecânica de funcionamento, assim como seu projeto de implementação.

No capítulo 4 é realizado um estudo de caso com a utilização das técnicas de refatoração automatizadas, em uma aplicação real. Neste capítulo são identificadas oportunidades de refatoração do código fonte da referida aplicação e são avaliados os benefícios e limitações detectados. Por fim, no capítulo 5, apresentam-se as considerações finais referentes ao trabalho, e discutem-se algumas ideias para continuidade.

2 REFATORAÇÃO

2.1 Conceitos

Fowler et al. (1999) definem refatoração como uma alteração feita na estrutura interna do *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 o *software*, aplicando-lhe uma série de modificações sem alterar o seu comportamento observável. Em particular, os resultados produzidos após a refatoração devem ser idênticos àqueles do *software* original.

O termo refatorar faz parte de um domínio de pesquisa mais amplo, relacionado à reestruturação de *software* (ARNOLD, 1986; GRISWOLD; NOTKIN, 1993). Comumente, é empregado para caracterizar reestruturações realizadas em *software* desenvolvido sob o paradigma da orientação a objetos. A ideia chave é redistribuir classes, variáveis e métodos através da hierarquia/estrutura do *software*, de forma a facilitar futuras adaptações e extensões (TOURWÉ; MENS, 2004). Em geral, são alterações simples que atuam sobre características não funcionais do *software*, como por exemplo, extensibilidade, modularidade, reusabilidade, complexidade e eficiência.

Refatorar o *software* não é uma prática nova, afinal aplicar melhorias e reestruturações ao código fonte é um processo implícito e contínuo à tarefa de desenvolver *software* (FOWLER et al., 1999). Muito do que se faz atualmente é agrupar, catalogar e documentar técnicas de refatoração de forma que as mesmas possam ser aplicadas de forma sistemática, melhoradas e compartilhadas.

2.1.1 Histórico

O primeiro trabalho extensivo e acadêmico sobre o tema refatoração foi a tese de doutorado de William F. Opdyke (OPDYKE, 1992), orientada pelo professor Ralph Johnson

da Universidade de Illinois em Urbana-Champaign (EUA). Willian Opdyke trabalhava em um setor da Bell Labs que desenvolvia o *software* de *switches* eletrônicos (equipamentos com uma série de restrições de confiabilidade e desempenho). Tais equipamentos possuíam uma vida útil bastante considerável e a maior parte do esforço de desenvolvimento do *software* era gasto na manutenção de tais sistemas ao longo do tempo. Com o objetivo de pesquisar e desenvolver técnicas para tornar o processo de evolução do *software* mais simples e menos custoso, nasceu o primeiro trabalho sobre o tema.

Posteriormente, no mesmo grupo de pesquisa, John Brant e Don Roberts, também orientados pelo professor Ralph Johnson, desenvolveram a primeira ferramenta para automatização de técnicas de refatoração. Tratava-se do *Refactoring Browser* (ROBERTS; BRANT; JOHNSON, 1996, 1997), uma ferramenta para código Smalltalk que introduzia pela primeira vez a automação de refatorações primitivas, simplificando e incentivando o uso da técnica.

Muito do que se tem feito desde então é a catalogação e documentação de refatorações, assim como a automatização de tais técnicas em ferramentas ou IDEs. Trabalhos mais recentes sobre o tema abordam problemáticas específicas de algumas características de linguagens de programação em particular, como os trabalhos que tratam de desafios na automatização de ações de refatoração para código da linguagem C (GARRIDO; JOHNSON, 2002, 2003) e que abordam o assunto no contexto da computação de alto desempenho (HPC) (OVERBEY et al., 2005; RIEGER et al., 2007; OVERBEY; NEGARA; JOHNSON, 2009).

2.1.2 Benefícios

Fowler et al. (1999) definem quatro razões principais para empregar técnicas de refatoração em projetos de *software*: melhorar o projeto, simplificá-lo, detectar falhas e agilizar o desenvolvimento. A refatoração pode ser utilizada para melhorar o projeto do *software* auxiliando na reestruturação do código fonte. Em geral, alterações de curto prazo ou sem o total entendimento acabam por desestuturar o *software* em relação ao seu propósito/projeto original. Essa situação tem um efeito cumulativo, ou seja, quanto mais difícil for a compreensão do projeto do *software* a partir do seu código fonte, mais difícil acaba sendo preservar o projeto, e o *software* rapidamente se desestrutura.

Neste sentido, reduzir a quantidade de código duplicado, embora não afete o com-

portamento do *software*, simplifica a tarefa de manutenção do mesmo. Quanto maior o código a ser entendido, maiores as chances de uma manutenção degradar a qualidade do *software*. É comum encontrar situações corriqueiras em que uma alteração no projeto do *software* não produz o resultado esperado, em função de que no mesmo projeto há mais locais onde a alteração também precisaria ser feita (e não o foi).

Tornar o *software* mais simples e mais fácil de entender é a segunda razão apontada por Fowler et al. (1999) para incentivar a utilização de refatorações. A tarefa de programar é um "momento íntimo" entre o programador e o computador, onde o primeiro diz exatamente o que o segundo deve fazer (e este o faz). Porém, por vezes o programador esquece que há uma "terceira pessoa" neste processo, que em muitos casos pode ser ele próprio. Durante o processo evolutivo, o *software* precisará de manutenção, e alguém (o próprio programador ou outra pessoa) precisará entender o código fonte para que possa alterá-lo ou corrigí-lo.

Ações de refatoração podem auxiliar a tornar o código mais legível, estruturando-o de forma que comunique melhor seu propósito. Um código reestruturado oportuniza ao programador conhecer e entender aspectos que antes não ficavam claros, mesmo que o *software* estivesse funcionando corretamente. Nessa mesma linha, uma terceira razão para ser refatorar um *software* é de oportunizar a identificação de falhas. Ao reestruturar o código e torná-lo mais simples, algumas construções passíveis de falhas e que antes passavam despercebidas acabam ficando claras ao programador, que vê na refatoração uma oportunidade de melhorar o projeto do *software*.

Velocidade de programação é o quarto fator apontado para incentivar o uso de refatoração. Isso não é uma constatação óbvia como as anteriores, mas, se há um bom projeto, um código simples e bem estruturado e livre de falhas, leva-se menos tempo para entender, reestruturar e consertar falhas. Apoiado fortemente em técnicas de refatoração, Beck e Fowler (2001) propõem a metodologia de desenvolvimento ágil conhecida como programação extrema (*eXtreme Programming* - XP). Nessa metodologia, o processo de design evolutivo se baseia em refatorações constantes de uma base de código simples.

Evolução de sistemas legados também representa uma boa motivação para o emprego de técnicas de refatoração. Na medida em que novas técnicas, práticas e ferramentas são incorporadas ao processo de desenvolvimento de *software*, é necessário e algumas vezes inevitável que o código legado tenha que evoluir para contemplar os novos recursos. A

evolução do *software* significa em muitos casos substituir construções anteriomente empregadas por novas (geralmente com melhor poder de expressão). Um exemplo disso seria substituir um trecho do código com desvios rotulados (ex. *GO TO*) por comandos de decisão ou laços de repetição. Em algumas linguagens de programação existem construções que se tornam obsoletas e são descontinuadas ou desaconselhadas (*deprecated*), sendo que técnicas de refatoração podem auxiliar na detecção e substituição de tais construções.

De forma semelhante à evolução do *software*, pode-se citar benefícios do emprego de técnicas de refatoração quando há mudança na arquitetura de execução do *software*. Existem situações nas quais determinadas construções podem se comportar melhor (ou não) dependendo da arquitetura onde o *software* é executado. Nestes casos, o emprego de técnicas de refatoração permite explorar situações nas quais a reestruturação do *software* o permite migrar de arquitetura de execução. Há situações em que, sem as reestruturações, o *software* sequer executa na nova arquitetura.

2.1.3 Recomendações

Uma das principais questões acerca da refatoração é qual o tempo certo para utilizá-la. A resposta depende de alguns fatores como o estado do código base, o estágio do ciclo de vida da aplicação ou ainda as necessidades imediatas do projeto. Algumas características podem ser citadas como indicadores de necessidade de refatoração (FOWLER et al., 1999): métodos ou classes muito extensos, métodos com excessivos parâmetros, falta de clareza ou legibilidade do código.

Fowler et al. (FOWLER et al., 1999) descrevem algumas situações para as quais é altamente recomendável a utilização de refatoração:

- **ao adicionar um novo recurso**: neste caso a refatoração deve ser utilizada inclusive para preparar o código para a mudança;
- **na correção de um erro (*bug*)**: se há conhecimento suficiente do algoritmo para corrigí-lo, também o há para simplificá-lo;
- **na 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. É comum que o programador desa-

credite que simples alterações venham a agregar qualquer melhoria ao *software*. O ganho se dá de forma cumulativa, ou seja, quando um grande conjunto de pequenas melhorias e boas práticas são aplicadas ao *software*. Existem refatorações primitivas e refatorações compostas, essas últimas sendo definidas como a aplicação de uma sequência ordenada das primeiras.

É desejável que a especificação de técnicas ou sua automatização sempre preserve a característica da simplicidade e atomicidade, de forma a não comprometer o comportamento observável do *software*. Também é possível que existam interrelações entre técnicas de refatoração, por exemplo: para que determinada técnica seja aplicada é pré-requisito que outra(s) técnica(s) seja(m) previamente aplicada(s).

Uma importante recomendação que se deve considerar antes de refatorar o *software* é a existência de um sólido conjunto de testes (DEURSEN et al., 2001). Preferencialemente, os testes precisam ser automatizados. Recomenda-se que toda classe/programa tenha seus próprios conjuntos de testes implementados e utilizados para testar seu funcionamento (HUNT; THOMAS, 2003). 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.

2.1.4 Pesquisas e Desafios

Em relação à pesquisa acerca de refatoração, Mens et al. (2003) classificam o tema em quatro grandes campos. O primeiro deles é o formalismo. Neste campo de pesquisa, a aplicação de técnicas de refatoração é estudada em um alto nível de abstração. Temas como divisão de programas (*programa slicing*) (ETTINGER, 2007), transformações em representações gráficas (*graph transformations*) (MENS; TAENTZER; RUNGE, 2007) e métricas de *software* (BOIS, 2006) (como medir os benefícios do uso de refatoração) são alguns assuntos que possuem propostas e questões em aberto.

Um segundo campo de pesquisa diz respeito às técnicas de refatoração. Neste caso técnicas são catalogadas e documentadas de forma que orientem sua aplicação sistemática. Geralmente consideram aspectos específicos do paradigma de programação no qual atuam. Um pouco mais específicas são as pesquisas no campo das linguagens de programação. Neste caso, além de considerar aspectos ligados ao paradigma de programação, faz-se necessário considerar construções da linguagem de programação no qual a técnica

será utilizada.

Suporte de ferramentas é uma área que desperta bastante interesse e com questões práticas em aberto (WATSON; DEBARDELEBEN, 2006; ROBERTS; BRANT; JOHNSON, 1996; OVERBEY; JOHNSON, 2008; GRAF; ZGRAGGEN; SOMMERLAD, 2007). Esse tema exige um considerável conhecimento da pesquisa desenvolvida nos demais campos (formalismos, técnicas e linguagens) e é neste campo que os resultados são mais visíveis e boa parte do que se produz é colocado em prática. Neste campo de pesquisa se estuda como ferramentas de apoio ao desenvolvimento podem automatizar técnicas de refatoração, atuando especificamente sobre um paradigma e uma linguagem de programação.

Também é válido considerar que tanto na pesquisa de técnicas, linguagens ou ferramentas, o uso de refatoração pode se aplicar a outros artefatos do *software* que não o código simplesmente. Alguns trabalhos abordam o uso de refatorações em diagramas UML (ASTELS, 2002; SUNYE et al., 2001) assim como a automatização de refatorações integradas a editores UML (BOGER; STURM; FRAGEMANN, 2003). Na área de banco de dados, há trabalhos que discutem a refatoração no contexto da evolução de esquemas relacionais (BOEHM et al., 2007; AMBLER; SADALAGE, 2006). Um dos grandes desafios é definir mecanismos para manter a consistência entre diferentes artefatos (implementação e modelo) de *software* com o uso de refatoração.

Paradigmas de programação não orientados a objetos têm motivado e desafiado pesquisadores em torno do tema da refatoração. De forma geral, o conceito principal se aplica igualmente independente do paradigma, mas a construção de ferramentas e a própria sistematização de técnicas possui características específicas. Existem trabalhos recentes que estudam o impacto e a forma de refatorar programas funcionais (LI, 1992; LI; THOMPSON, 2008) e lógicos (SEREBRENIK; SCHRIJVERS; DEMOEN, 2008). Destacam-se também os estudos na tentativa de evoluir aplicações legadas (LIU; BATORY; LENGAUER, 2006).

Alto desempenho e refatoração constituem-se de uma área de pesquisa relativamente recente. Observa-se a publicação de estudos de caso relacionados a técnicas de refatoração para evolução de aplicações ligadas ao alto desempenho (OVERBEY; NEGARA; JOHNSON, 2009; SANDERS; DEUMENS; LOTRICH, 2008), uso de técnicas para melhorar o desempenho de aplicações (RIEGER et al., 2007) e em especial a automatização de técnicas em ferramentas ou *frameworks* específicos (OVERBEY et al., 2005; WAT-

SON; DEBARDELEBEN, 2006). Neste contexto, há uma carência de ferramentas que agreguem produtividade ao desenvolvimento de aplicações de alto desempenho e que estejam integradas a um ambiente de desenvolvimento.

2.2 Refatoração e Fortran

Fortran, acrônimo para FORmula TRANslation, é uma linguagem de programação imperativa, isto é, baseada em um controle sequencial do fluxo de execução alicerçado por construções de desvio como procedimentos e funções. Embora preservem suas características imperativas, versões atuais da linguagem permitem várias características de programação orientada a objetos.

Desempenho é a palavra-chave que explica a perpetuação histórica de Fortran. A linguagem sofreu várias revisões ao longo do tempo, ganhou bibliotecas muito ricas e altamente otimizadas. Uma das suas principais vantagens, considerando a computação de alto desempenho, é a existência de operações para tratamento de dados multidimensionais que normalmente não são encontradas de forma nativa em outras linguagens de programação (KOFFMANN; FRIEDMAN, 2006). Pelo fato de que o domínio de aplicação de Fortran é bastante específico e ligado à pesquisa científica e não ao mercado, há uma lacuna entre a quantidade de código legado escrito em Fortran e técnicas/ferramentas de refatoração existentes para a linguagem Fortran.

2.2.1 A linguagem Fortran

A primeira versão de Fortran foi apresentada em 1957 por uma equipe dirigida por John Backus (NYHOFF; LEESTMA, 1997). Foi a linguagem de programação pioneira em apresentar uma semântica de alto nível e também a utilização de um compilador. Fortran libertou programadores da difícil e demorada tarefa de programar computadores em baixo nível ou utilizando linguagem *assembly* (DE, 2004). A pesquisa de Backus que levou ao nascimento do Fortran objetivava responder ao seguinte questionamento "(...) pode uma máquina traduzir uma linguagem matemática abrangente em um conjunto razoável de instruções, a um baixo custo, e resolver totalmente uma questão?" (BACKUS, 1954).

A principal aplicação de Fortran desde seu advento é a computação científica. Este foco de atuação fez com que a linguagem tenha se perpetuado ao longo do tempo e,

mesmo depois de mais de cinquenta anos de sua primeira versão, ainda seja altamente utilizada em aplicações científicas de alto custo computacional.

Fortran tem sido utilizada para escrever diversos programas e rotinas em bibliotecas padronizadas, como BLAS (*Basic Linear Algebra Subprograms*), LAPACK (*Linear Algebra PACKage*), IMSL (*International Mathematics and Statistics Library*), NAG (*Numerical Algorithms Group*) e pode ser considerada a linguagem predominante em áreas de aplicação como matemática, física, engenharia e análises científicas (DE, 2004; KOFFMANN; FRIEDMAN, 2006; NYHOFF; LEESTMA, 1997). Também oferece suporte à utilização de bibliotecas para programação concorrente e distribuída como MPI (RASMUSSEN; SQUYRES, 2005) e OpenMP (HERMANNS, 2002), por exemplo.

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. Outro exemplo é o *benchmark* LINPACK (DONGARRA; LUSZCZEK; PETITET, 2003), utilizado para comparar o desempenho dos maiores computadores do mundo, auxiliando na construção da lista TOP 500 (TOP500, 2009).

Todo o pioneirismo e a idade da linguagem Fortran lhe trazem o ônus de manter viva a semântica e a compatibilidade com versões anteriores. É possível encontrar em uma mesma aplicação Fortran código delimitado (que respeita o espaço destinado ao espaço de perfuração dos antigos cartões perfurados), direcionamento do fluxo de instruções com construções rotuladas e, mais recentemente, código com características de orientação a objetos. Em Fortran não há o conceito de palavras reservadas, de forma que é possível encontrar em aplicações o uso de variáveis com nomes de comandos como *IF*, *WHILE* ou *DO*.

Fortran é uma linguagem fortemente tipada, o que significa que cada elemento (entidade de informação) tem um e um só tipo bem definido. O alfabeto Fortran (base de sua linguagem) é constituído pelos seguintes conjuntos de caracteres:

- letras maiúsculas (A-Z) ou minúsculas (a-z) (indistinguíveis entre si, totalizando 26);

- os algarismos 0...9, perfazendo 10 caracteres;
- o espaço em branco, a vírgula (,), o ponto final (.) e o sinal de igual (=);
- os operadores de soma (+), subtração (-), multiplicação e exponenciação (*) e divisão (/);
- os parênteses ();
- os símbolos ‘ (para delimitar caracteres), ! (para comentários), : (definição de tipo) e & (continuação de linha);
- e por fim, versões mais recentes permitem ainda a utilização dos sinais de maior (>) e menor (<) e percentagem (%), este último para acesso a campos de dados estruturados.

A organização típica de um programa Fortran é composta por um cabeçalho único de identificação (*PROGRAM*) onde são inseridos os comandos da linguagem até que o bloco seja finalizado (*END*). Um programa pode ser sub-dividido em blocos ou segmentos de código, sendo que as principais divisões podem ser:

- ***SUBROUTINE***: procedimento invocado dentro de uma declaração *CALL* e que recebe valores de entrada e devolve os resultados através de uma lista de argumentos;
- ***FUNCTION***: procedimento cujo resultado é um único valor (podendo ser multidimensional) que pode ser combinado com variáveis e constantes a fim de formar expressões;
- ***MODULE***: seção de um programa Fortran (a partir da versão 90) que pode ser compilada e reutilizada por outros programas como uma subrotina externa. Geralmente é usada para empacotar conjuntos de procedimentos e funções de forma a favorecer o reuso;
- ***COMMON*** e ***BLOCK DATA***: estruturas *COMMON* podem ser utilizadas para definir e utilizar um conjunto de variáveis através do programa principal e nas subrotinas/funções que os declaram. Um *BLOCK DATA* é segmento de código chamado para inicializar variáveis declaradas em um *COMMON*. Somente pode existir um único *BLOCK DATA* em um programa Fortran.

Fortran disponibiliza construções para blocos de desvio condicional (*IF..ENDIF*) e de repetição (*DO..ENDDO*), além do direcionamento do fluxo de instruções com construções rotuladas (*GO TO*). Permite a definição de tipos de estruturas de dados compostos (*TYPE*) e também a utilização de ponteiros (*POINTER*).

Há ainda duas revisões recentes na linguagem, Fortran 2003 (METCALF; REID; COHEN, 2004) e Fortran 2008 (ADAMS et al., 2008), para as quais ainda não existe nenhum compilador com todos os seus recursos implementados. Pode-se observar, em ambas as revisões, que cada vez mais são exploradas novas construções ligadas ao desempenho como a utilização de laços concorrentes (*DO CONCURRENT*) e co-matrizes (*co-arrays*), além de construções ligadas ao paradigma da orientação a objetos (abstração, encapsulamento, polimorfismo e herança).

A manipulação de dados multidimensionais é certamente um dos maiores atrativos da linguagem Fortran para a construção de aplicações de alto desempenho. A partir da versão 90, a linguagem disponibiliza algumas construções para utilização de matrizes e vetores de uma forma bastante simples, o que favorece a legibilidade do código. Considere o exemplo de código à esquerda na figura 2.1, um típico bloco aninhado de laços de repetição para copiar as informações da matriz B para a matriz A.

A mesma operação pode ser feita conforme o código colocado na primeira linha à direita da figura 2.1. Operações matemáticas sobre dados multidimensionais, atribuições ou inicializações são bastante frequentes em aplicações de alto desempenho. Em função disso, a sintaxe de Fortran oferece construções bastante simplificadas para a manipulação de matrizes e vetores. Ainda na figura 2.1, no lado direito, é possível visualizar (a partir da segunda linha) algumas dessas operações (soma, produto e divisão).

```
do I = 1, N              |   A = B     ! Atribuição
  do J = 1, N            |   A = B * 2 ! Multiplicação por escalar
    A(I,J) = B(I,J)      |   A = B + B ! Soma de elementos
  end do                 |   B = A / B ! Divisão
end do                   |
```

Figura 2.1: Formas alternativas de utilização de código Fortran para manipulação de *arrays*

2.2.2 Refatoração e o Paradigma Estruturado

Reestruturação de código é um termo facilmente encontrado para referir-se a transformações internas de programas estruturados (MENS et al., 2003). Embora o advento do termo refatoração tenha origem na orientação a objetos, é possível utilizar seu conceito para o paradigma estruturado com algumas limitações.

As técnicas agrupadas por Fowler et al. (1999), relacionadas à composição de métodos, organização de dados e estruturas de controle, aplicam-se quase que na sua totalidade ao paradigma estruturado. Um exemplo disso é a técnica de refatoração *"Extract Method"*. Nela, um bloco de código selecionado é transformado em um novo método e em seu lugar uma chamada ao novo método é adicionada. Com pequenas diferenças, sua mecânica de funcionamento pode ser bastante semelhante se aplicada ao código estruturado, por exemplo, extraindo um bloco de código para um procedimento ou função.

A técnica conhecida por *"Rename"*, que é utilizada para renomear uma entidade do código fonte (variável, procedimento), também pode ser igualmente aplicada em um programa estruturado sem alteração em sua mecânica de funcionamento. Neste caso há inclusive uma simplificação, uma vez que não há uma hierarquia de classes para ser alterada.

Linguagens não orientadas a objetos são mais difíceis de refatorar, pois fluxos de controle e de dados são fortemente interligados (DE, 2004). Em geral, reestruturações em código não orientado 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. Na seção seguinte aborda-se o uso e as motivações de uma ferramenta que atua sobre código Fortran.

2.3 Photran

O Photran é um ambiente integrado de desenvolvimento (IDE) para código Fortran baseado na plataforma Eclipse (BEATON; RIVIERES, 2006). Seu desenvolvimento inicial é fruto dos trabalhos de Vaishali De (2004), Julia Dtragan-Chirila (2004) e Rohit Eipe (2004), desenvolvidos no grupo de pesquisas em arquiteturas de software (SAG, *Software Architecture Group*) coordenados pelo professor Ralph Johnson na Universidade Illinois em Urbana-Champaign.

Assim como outras ferramentas baseadas no Eclipse, o Photran é desenvolvido em linguagem Java e compõe-se de *plugins* e recursos (*features*). *Plugins* adicionam funci-

onalidades ao Eclipse. Recursos são unidades de desenvolvimento (diversos *plugins* são empacotados em recursos e um recurso é distribuído aos usuários). Arquiteturalmente, o projeto Photran divide-se em subprojetos em que cada um se constitui de um *plugin* ou um recurso.

Atualmente, o Photran é um projeto oficial da Eclipse Foundation e faz parte de um macro projeto denominado PTP (*Parallel Tools Platform*). O objetivo do projeto PTP é explorar a deficiência de ferramentas para a computação científica (WATSON; DEBARDELEBEN, 2006) e de alto desempenho (VANTER; POST; ZOSEL, 2005), organizando em torno do IDE do Eclipse ferramentas para desenvolvimento, depuração, distribuição e monitoramento de tais aplicações.

O Photran baseia-se em dois atributos de qualidade: a usabilidade e a manutenabilidade. Primeiramente o Photran é um editor de código Fortran, e como tal precisa ter uma interface gráfica fácil de manusear, intuitiva, configurável, enfim, que torne mais produtivo o trabalho do programador. Da mesma forma, e ligado mais diretamente à automatização de técnicas de refatoração, o Photran deve ser extensível e integrado, permitindo lhe adicionar novos recursos sem comprometer seu funcionamento.

2.3.1 Recursos do IDE Photran

Photran pode ser considerado antes de tudo uma ferramenta de desenvolvimento para código Fortran que contempla o ciclo de editar, compilar, executar e depurar. O foco do Photran é a codificação e para tanto disponibiliza assistentes, atalhos e recursos de apoio a essa atividade.

Os componentes visuais da interface com o usuário (UI, *User Interface*) do Photran foram adaptados a partir de outro *plugin* para Eclipse, o CDT (*C,C++ Development Tools*) (LEE; SCUSE, 2004; GRAF; ZGRAGGEN; SOMMERLAD, 2007). O CTD é o primeiro *plugin* a habilitar o uso do Eclipse para trabalhar com linguagem C/C++. O Photran estende os pontos de extensão do CTD implementando suas interfaces. Um ponto de extensão consiste em um local bem definido (no Eclipse) onde outros *plugins* contribuem com suas funcionalidades. Isso favorece o reuso dos elementos da interface gráfica, assistentes, analisadores de erros, entre outros. A figura 2.2 apresenta um *screenshot* do Photran sendo executado em ambiente Linux.

Um dos pontos de extensão mais importantes do Photran é o editor de código. A

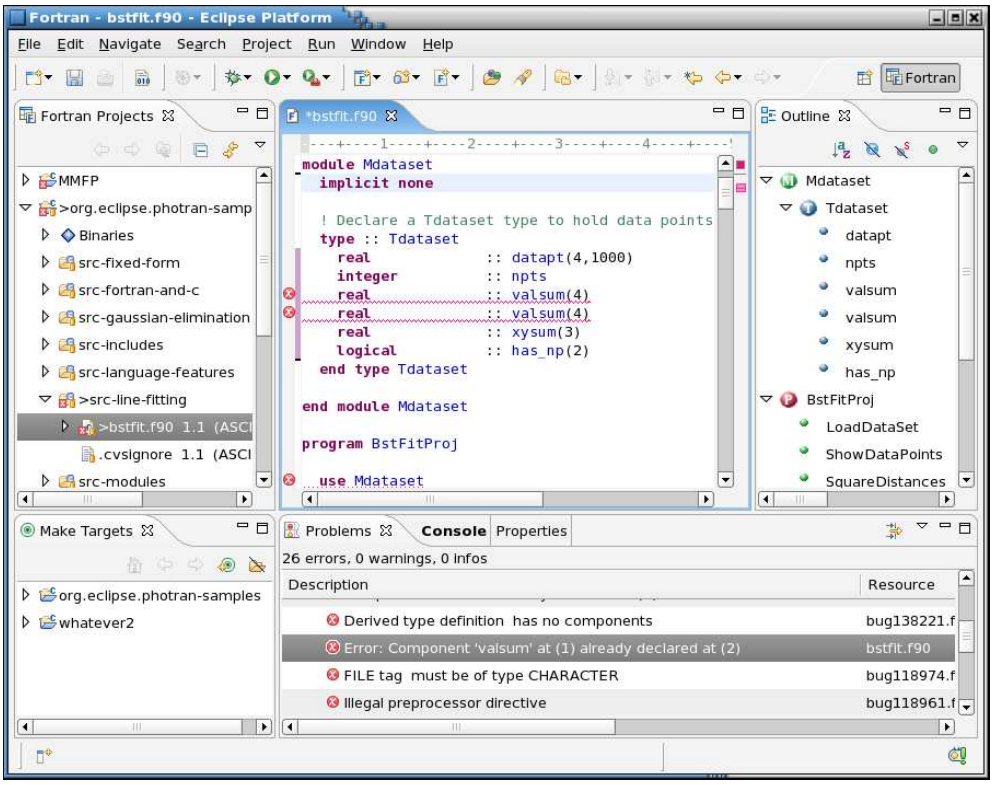

Figura 2.2: *Screenshot* do Photran em ambiente Linux

linguagem Fortran permite dois tipos de edição de código: formato fixo (versão 77 e anteriores) e formato livre (versão 90 e posteriores). O primeiro caso é uma herança da época em que o código era impresso em cartões perfurados e precisava respeitar uma borda inicial formada por 6 colunas, reservadas para *labels* de desvio de processamento (colunas 1-5) e o caractere de continuação de linha (coluna 6). O código fonte do formato fixo começa a ser considerado apenas na posição 7 e suporta até 72 caracteres (também em função de limitações impostas pelos cartões perfurados).

O formato livre permite começar a escrever o código em qualquer lugar (não há uma coluna específica para iniciar o programa). Em função das diferenças significativas no processamento e visualização de uma e outra forma de escrita, o Photran disponibiliza duas classes para implementar os editores visuais: *FixedFormFortranEditor* e *FreeFormFortranEditor*. Essas duas classes são sub-classes de *AbstractFortranEditor* que é implementado pelo Photran e agrega métodos e funcionalidades comuns a ambos e que por sua vez estende as características do editor padrão do Eclipse (*TextEditor*). A figura 2.3 apresenta a hierarquia de classes e o formato dos editores de código fixo e livre.

O editor de código também é responsável por acionar as classes que fazem o reconhecimento da sintaxe Fortran, formatando visualmente diferentes elementos do código

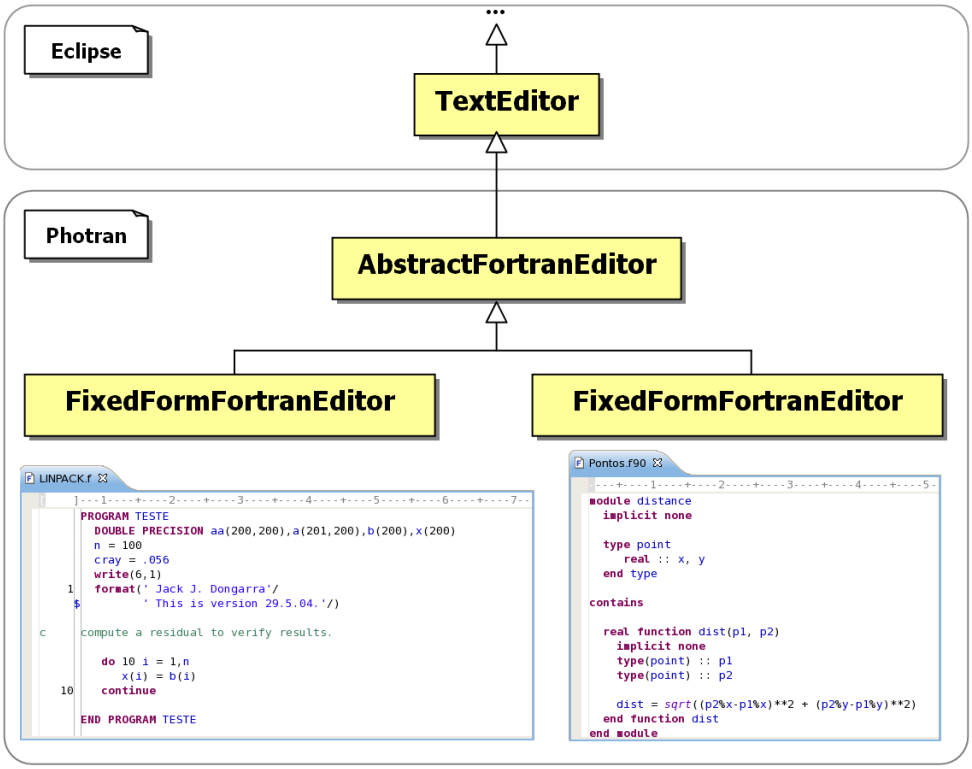

Figura 2.3: Hierarquia de classes dos editores de código do Photran

(caracteres, variáveis, números, comandos, etc.). O Photran permite a utilização de alguns assistentes que completam o código na medida em que ele vai sendo digitado. Há também alguns recursos bastante úteis para comentar e descomentar um bloco de código selecionado ou ainda avançar ou recuar um bloco de código.

Também são disponibilizados assistentes para criação de projetos Fortran. Da mesma forma, ao adicionar um novo arquivo de código fonte ao projeto, um assistente pode ser utilizado para verificar se o nome do novo arquivo é válido para a linguagem Fortran, considerando para isso a extensão do nome do arquivo que é fornecido.

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). Ao clicar sobre um elemento da *Outline View*, o Photran posiciona o cursor sobre o elemento de código mais significativo do editor que o representa.

Em relação ao processo de compilação, o Photran oferece recursos para que os projetos sejam gerenciados de automaticamente (neste caso, quando novos arquivos são adicionados ao projeto, o arquivo *makefile* que faz a configuração da compilação é alterado

para considerar também o novo arquivo). Também é possível desabilitar este recurso. A compilação é configurada por meio das opções do projeto e permite utilizar diferentes compiladores. Da mesma forma, o Photran fornece alguns interpretadores de erros, que a partir da saída (erros) gerada pelo compilador conseguem interpretar a mensagem e apontar, no editor de código, a linha onde o erro ocorreu.

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

2.3.2 Refatorações do IDE Photran

Em sua versão atual (4 *beta* 5) o Photran disponibiliza duas técnicas de refatoração: *Rename* e *Introduce Implicit None*.

2.3.2.1 Rename

É sempre recomendável que métodos, variáveis e tipos sejam nomeados de forma que comuniquem facilmente sua intenção e revelem seu propósito. Um código fonte de boa qualidade é aquele que pode ser entendido facilmente (dispensando até mesmo comentários), ou seja, nomear adequadamente é uma prática importante e indispensável.

A refatoração *rename* poderia ser considerada uma ação semelhante a um "localizar e substituir". Através dela, uma palavra é localizada e modificada em todo o projeto, considerando e observando corretamente o escopo do termo a ser refatorado. É neste sentido que a refatoração *rename* difere bastante de uma ação "localizar e substituir", uma vez que uma expressão literal e um nome de variável tem escopos bem distintos dentro do código fonte. Este tipo de refatoração se aplica em variáveis locais, subprogramas, tipos derivados, módulos, *namelists*, *block data* e *common blocks*.

2.3.2.2 Introduce Implicit None

O Fortran não obriga a declaração explícita de tipos para as variáveis utilizadas no programa, exceto em alguns casos especiais como variáveis lógicas, cadeias de caracteres e vetores de um modo geral. No entanto, a declaração explícita de variáveis constitui uma

boa prática de programação.

A instrução *IMPLICT NONE* obriga que todas variáveis de um programa ou subprograma sejam declaradas explicitamente pelo programador. Se o comando *IMPLICIT NONE* não for utilizado e as variáveis não forem declaradas explicitamente, o Fortran utiliza-se da seguinte regra para determinar o tipo de dado de uma variável: se a primeira letra da variável começar por I, J, K, L, M ou N será definida como inteiro, qualquer outra letra será do tipo real.

A refatoração *Introduce Implicit None* adiciona o comando *IMPLICIT NONE* e as declarações explícitas para todas as variáveis que não tiverem declaração. Tal técnica se aplica a programas, subprogramas e módulos.

2.3.3 Requisitos para Automatização de Ações de Refatoração

Do ponto de vista prático, uma ferramenta de refatoração deve ser veloz, deve estar integrada a um ambiente de desenvolvimento e deve dispor de formas para desistir ou anular uma refatoração executada ou simulada. Uma ferramenta que deseja dispor de mecanismos para automatização de técnicas de refatoração precisa minimamente, segundo Roberts (1999), contemplar:

- **Representação abstrata do programa**: A reestruturação/manipulação no código fonte de uma aplicação pela ferramenta de refatoração dificilmente é feita em baixo nível (alterando estruturas textuais). Faz-se necessário que a ferramenta ofereça a representação abstrata do programa por meio de uma árvore sintática.

- **Banco de dados do programa**: A ferramenta precisa guardar de forma atualizada um conjunto de informações ligadas à representação abstrata do programa, mas não hierarquizadas. Deve dispor de um repositório pesquisável, fornecendo um conjunto de operações que subsidiam decisões em tempo de execução da técnica de refatoração.

- **Acurácia**: é desejável que ao final da refatoração o programa refatorado preserve o mesmo comportamento que tinha antes de sofrer a refatoração (CORNELIO, 2004). A ferramenta deve dispor de recursos para minimamente detectar a introdução de um erro (e permitir desfazer as alterações). Também é desejável que as técnicas implementadas conheçam suas limitações de forma a não permitir que as

mesmas, se acontecerem, afetem negativamente o código fonte da aplicação.

2.3.4 AST e VPG

Uma árvore sintática abstrata (*Abstract Syntax Tree*, AST) é uma estrutura para representação do código do programa. Compõe-se de uma raiz da qual são derivados vários nós que por sua vez compõe-se de outros nós. O último nível dessa árvore geralmente representa os comandos ou operadores da linguagem de programação. Cada nó é classificado de acordo com seu funcionamento e suas ações. Por exemplo, um nó do tipo declaração provavelmente derivará outros nós que representarão, por exemplo, o nome da variável declarada e o tipo da mesma (eventualmente ainda um valor padrão de inicialização).

A construção de uma AST requer a existência de um analisador sintático específico para a linguagem de programação com a qual se deseja trabalhar. A análise sintática (*parsing*) é uma etapa do processo de compilação e pode ser definida com um algoritmo que, a partir de uma sentença, constrói uma árvore gramatical (decompondo a estrutura do código fonte a partir de uma gramática formal) (DELAMARO, 2004). 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 peça chave para a automatização de ações de refatoração (OVERBEY; JOHNSON, 2009). Ela possibilita ao desenvolvedor navegar pela estrutura do código fonte, detectando correlações entre seus nós e identificando oportunidades de refatoração. A ação de refatoração em geral introduz modificações na estrutura da AST (incluindo, excluindo ou alterando o posicionamento dos nós). A figura 2.4 ilustra um código em linguagem Fortran e sua respectiva AST (a figura foi produzida utilizando-se da *OutLine View* do Photran).

Analisadores sintáticos são complexos de serem construídos e, dependendo da estrutura da linguagem de programação, podem demandar um considerável tempo de codificação e testes. Geradores de analisadores sintáticos são aplicações que a partir de uma

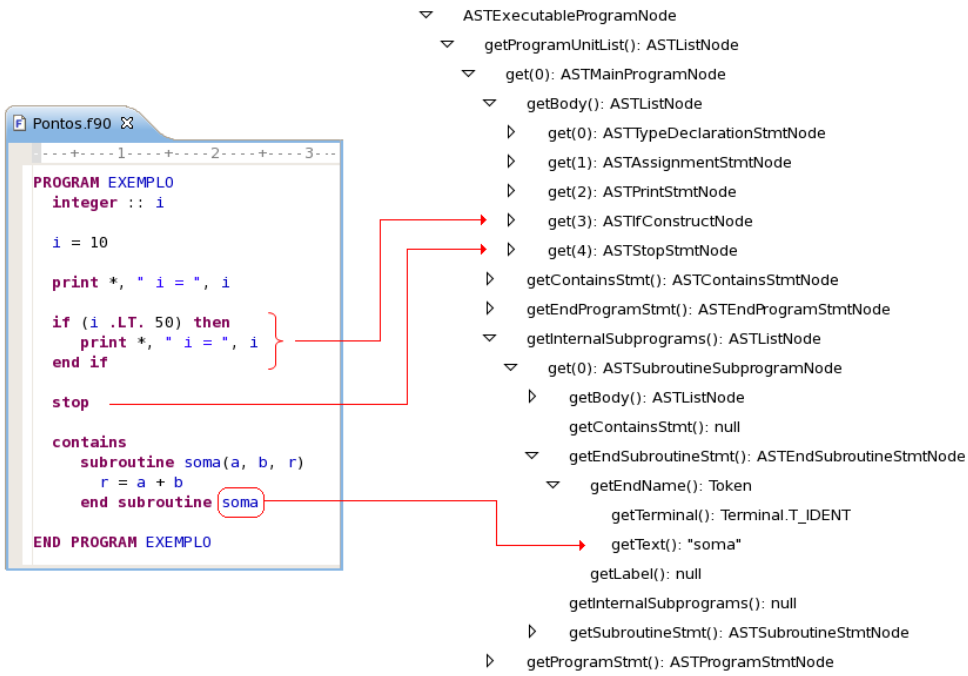

Figura 2.4: Código Fortran e sua AST - *Abstract Syntax Tree*

gramática (representando a estrutura de uma linguagem de programação) geram o código fonte base de um algoritmo de análise sintática (um *parser*) (DELAMARO, 2004). Recebem como entrada um arquivo especificando a gramática da linguagem e geram como saída um código fonte em determinada linguagem, contendo rotinas que realizam o reconhecimento de sentenças segundo a gramática utilizada. São exemplos de ferramentas para geração de analisadores sintáticos JavaCC (COPELAND, 2007), ANTLR (PARR, 2007) e Ludwig (OVERBEY; JOHNSON, 2008, 2009).

No caso do Photran, seu parser foi produzido utilizando-se do Ludwig. O projeto do Ludwig nasce da idéia de produzir analisadores sintáticos e árvores sintáticas abstratas com foco na simplificação e rapidez do processo de automatização de ações de refatoração. 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 (*Rewritable AST*), 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 ações de refatoração.

Além da representação abstrada da árvore sintática que é obitda pelo *parser* do Pho-

tran, uma outra estrutura muito importante também é alimentada durante este processo: o *Virtual Program Graph* (VPG). O VPG agrega à árvore sintática abstrata outras ligações entre seus nodos que não representam necessariamente a hierarquia da árvore. Algumas dessas ligações adicionais podem ser, por exemplo, um vínculo entre a utilização de determinada variável (em uma expressão) e sua respectiva declaração, ou ainda um comando e seu escopo (bloco lógico de atuação). É por meio da VPG que se pode obter um conjunto bastante rico de informações para subsidiar a manipulação dos nodos de uma AST.

Pode-se entender o VPG como um conjunto de arestas que ligam os nós de uma AST de forma não hierarquizada. Na figura 2.5, é possível observar por meio de um pseudo-código a AST do mesmo. As linhas que ligam a utilização da variável *i* à sua declaração e essa ao escopo (*PROGRAMA*) na qual a mesma tem validade, são exemplos de informações obtidas por meio de consultas ao VPG.

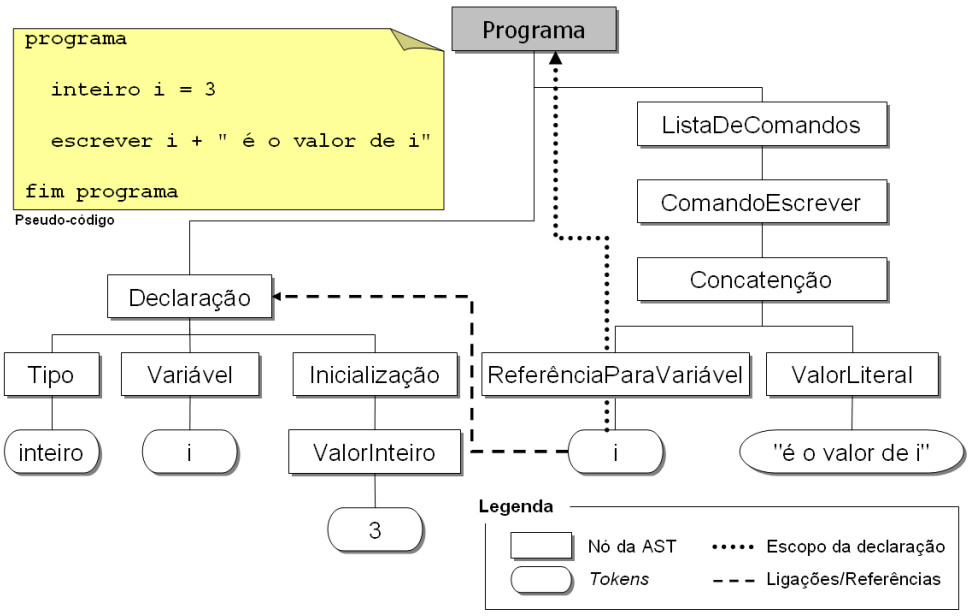

Figura 2.5: Pseudo-código e seu VPG - *Virtual Program Graph*

O capítulo 3 fornece mais detalhes sobre como as técnicas de refatoração são implementadas no Photran, quais são os requisitos e como as implementações podem ser integradas à ferramenta. Na seção 3.3 são discutidas as técnicas implementadas por este trabalho.

3 DESENVOLVIMENTO

3.1 Identificação das Técnicas de Refatoração

A primeira etapa do desenvolvimento e automatização de refatorações consiste da identificação e descrição das técnicas escolhidas para tal. Durante a realização do presente trabalho foram selecionadas 4 técnicas cuja descrição e motivação serão descritas nessa seção. Estas técnicas têm diferentes objetivos:

- Evolução da linguagem (refatorar um código escrito para determinada versão de Fortran substituindo-o por construções de uma versão mais recente);
- Design de código (refatorações que afetam apenas aspectos ligados à organização do código fonte, o que pode facilitar a manutenção de códigos muito complexos);
- Explorar construções de código que oferecem melhor desempenho;

A primeira técnica de refatoração selecionada para ser automatizada afeta o design de código e favorece a evolução da linguagem. Trata-se da substituição de operadores obsoletos (nomeada na ferramenta Photran de *Replace Obsolete Operators*). Dentre as várias evoluções sintáticas pelas quais Fortran tem passado ao longo do tempo, uma delas é a forma de representar os operadores relacionais. Na tabela 3.1 pode-se visualizar as antigas e novas construções dos operadores relacionais e seu funcionamento. Em geral, as construções mais recentes têm melhor poder de expressão e são mais facilmente entendidas (principalmente por programadores menos experientes em Fortran).

A utilização do novo formato de operadores relacionais foi introduzida à sintaxe da linguagem Fortran a partir da versão 90. A substituição de operadores obsoletos por novas construções simplifica o entendimento do código, tornando-o mais legível e auxilia a evolução do código para uma sintaxe mais natural e recente. Não há nenhuma relação

Tabela 3.1: Antigas e novas construções de operadores relacionais de Fortran

Comparação	Operador Obsoleto	Novo Operador
Menor que (*less than*)	.LT.	<
Menor ou igual a (*less or equal*)	.LE.	<=
Maior que (*great than*)	.GT.	>
Maior ou igual a (*great or equal*)	.GE.	>=
Igual a (*equal*)	.EQ.	==
Diferente de (*not equal*)	.NE.	/=

direta entre a utilização dessa técnica de refatoração com o ganho de desempenho, porém conforme já fora mencionado na seção 2.1.2, um código mais limpo e legível oportuniza a identificação de melhores construções ou reestruturações.

A figura 3.1 ilustra a ação *Replace Obsolete Operators* sendo utilizada pelo Photran. Na região mais à esquerda da janela construída pelo Eclipse, o programador visualiza o código original e na região à direita o código refatorado. Nessa última, as alterações realizadas estão em destaque. Essa janela permite ao programador aceitar ou não o resultado da refatoração. Em não aceitando, nenhuma alteração é realizada.

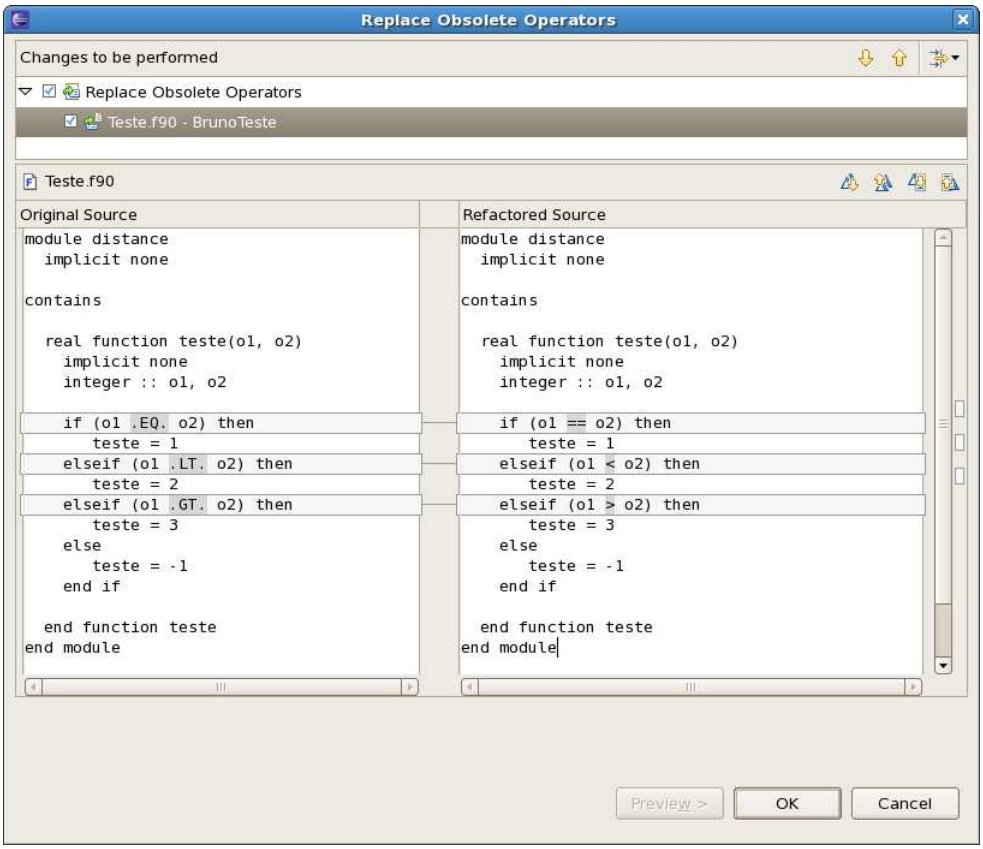

Figura 3.1: Refatoração utilizando a técnica *Replace Obsolete Operators*

A segunda técnica de refatoração selecionada para ser automatizada, assim como a primeira, também afeta o design de código e favorece a evolução da linguagem. Trata-se da inclusão do atributo *INTENT* (nomeada na ferramenta Photran de *Introduce INTENT*). 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, sendo portanto definido antes da chamada ao subprograma e não sendo utilizado para retornar resultados (não pode ser alterado dentro do subprograma). Uma tentativa de alterar um argumento deste tipo pelo subprograma é um erro que pode ser detectado se sua intenção for declarada.

- ***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. Um argumento de subprograma com *INTENT(OUT)* não deve ser referenciado antes que o subprograma o tenha inicializado. Um argumento *INTENT(OUT)* precisa ser necessariamente uma variável (pois é nessa variável que o valor de retorno será atribuído). Expressões não podem ser usadas como parâmetros para argumentos do tipo *INTENT(OUT)*.

- ***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. Pode ser referenciado antes de ser definido. Da mesma forma que *INTENT(OUT)*, um argumento recebido do tipo *INTENT(INOUT)* deve ser um objeto de dados (uma variável).

Embora opcional, o uso de *INTENT* é altamente recomendado por dois motivos. O primeiro é que permite ao programador conhecer a intenção dos argumentos de um subprograma apenas pela sua interface/declaração, sem a necessidade de ler todo seu código (que pode ser bastante extenso). Isso favorece a legibilidade do código. O segundo motivo diz respeito a oportunizar 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. Uma vez que o compilador é informado

acerca de como os argumentos serão utilizados (e consegue validar isso), pode optar por diferentes estratégias de armazenamento quando o programa for executado.

Em alguns casos, a melhor estratégia poderá ser a de fazer uma cópia do valor e fornecer a cópia ao argumento. Outra forma seria realizar uma passagem de parâmetro por referência (fornecendo diretamente o endereço de memória da variável correspondente ao argumento). Neste caso são economizadas algumas instruções para fazer a cópia.

A principal motivação da técnica aqui nomeada de *Introduce Intent* 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.

A figura 3.2 demonstra a refatoração *Introduce Intent* aplicada a um típico código de subprograma Fortran. No lado esquerdo da figura temos o código original, uma função *area*, que recebe três parâmetros: *h*, *b* e *r* (não declarados com o atributo *INTENT*). No lado direito da figura (código refatorado) é possível observar que a refatoração detectou o tipo de *INTENT* de cada parâmetro e realizaou a alteração.

A terceira técnica de refatoração selecionada por este trabalho para ser automatizada, afeta exclusivamente o design de código e é utilizada para empacotar um bloco de código selecionado em um subprograma. Extrair subrotina (ou extrair método) é um exemplo clássico de refatoração aplicável com poucas modificações a qualquer paradigma de programação. Consiste basicamente em escolher um fragmento de código, transformar o fragmento no corpo de um novo subprograma (ou método) e substituir o fragmento original por uma chamada ao novo subprograma.

Fowler et al. (1999) dedicam um capítulo para abordar um conjunto e técnicas ligadas à composição de métodos. Em geral, o problema está relacionado a códigos longos em demasia que são divididos em porções menores e devidamente empacotadas em subprogramas. A principal problemática de longos trechos de código é a excessiva informação que fica ocultada pela complexidade da lógica associada (LAKHOTIA; DEPREZ, 1998).

A maior complexidade da técnica de extrair subprograma (nomeada de *Extract Subroutine*) são as variáveis existentes no corpo do código a ser extraído. É preciso uma análise inicial para identificar se a variável é utilizada exclusivamente no trecho de có-

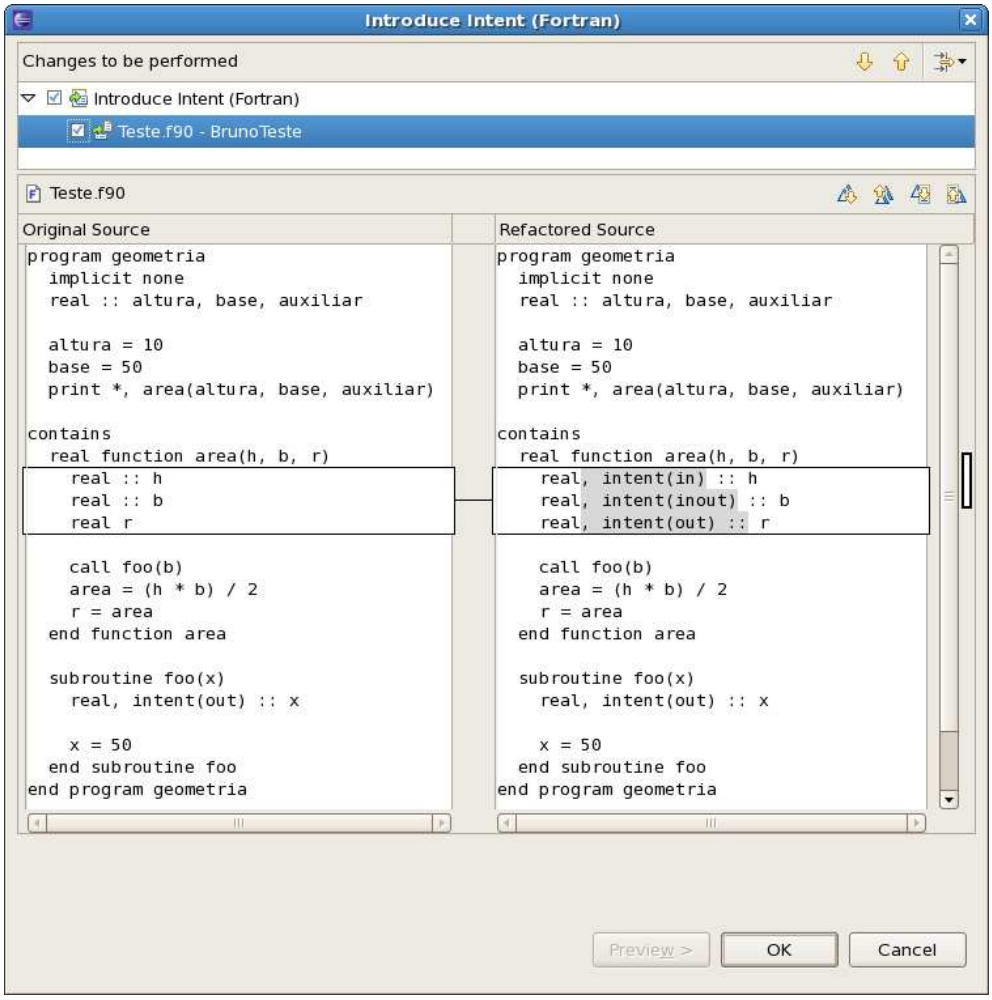

Figura 3.2: Refatoração utilizando a técnica *Introduce Intent*

digo que está sendo extraído ou se é referenciada além dele. No primeiro caso (mais simples) a variável é tratada como local e precisa apenas ser declarada na nova subrotina que será criada.

Uma variável que é acessada fora do bloco que está sendo extraída precisa ser transformada em um parâmetro. O bloco de código extraído é transformado em um subprograma e em seu lugar é adicionado uma chamada ao novo subprograma (informando os parâmetros necessários). Essa técnica é bastante útil no trabalho de tornar um código mais simples e legível. Porém, extrair de forma manual um subprograma (em um grande trecho de código) requer bastante análise por parte do desenvolvedor, o que justifica a necessidade de automatização de tal tarefa.

A figura 3.3 ilustra a refatoração *Extract Subroutine* sendo utilizada para extrair do código (à esquerda) os comandos para construção de um menu. Um subprograma *menu* é criado e adicionado ao código (logo após a instrução *CONTAINS*) e no lugar do código

original é feita a chamada *CALL menu(numero)*. Neste caso, o código extraído faz uso de uma variável (*numero*) que foi transformada em um parâmetro e que utiliza o atributo *INTENT(INOUT)*

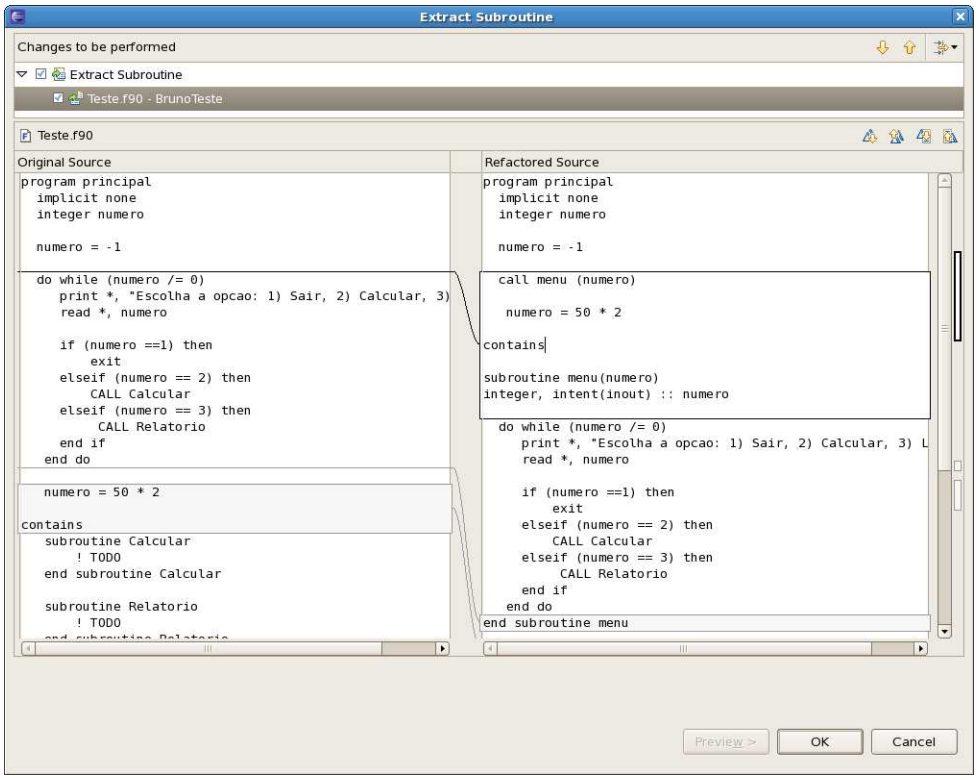

Figura 3.3: Refatoração utilizando a técnica *Extract Subroutine*

A última técnica de refatoração selecionada para ser automatizada foi escolhida em função de sua relação direta ao ganho de desempenho. Trata-se do desenrolamento de laço de repetição (*Loop Unrolling*) (HENNESSY; PATTERSON, 2002), que consiste em potencializar o uso de instruções no corpo do laço de repetição objetivando diminuir suas interações.

Em algumas construções de repetição utilizadas na programação estruturada, pode existir um conjunto tão pequeno de instruções a serem executadas que boa parte da computação acaba sendo realizada em função do controle do laço de repetição. Neste caso, a verificação da condição de saída e o incremento da variável de controle aumentam consideravelmente o número de instruções utilizadas pelo processador.

O desenrolamento de laço diminui a quantidade de comparações que são executadas como condição de parada e também o número de incrementos realizados na variável de controle do laço de repetição. Em sendo adicionadas mais instruções similares dentro do mesmo laço, possibilita-se que a variável de controle seja alterada em um passo maior, e

a comparação passa a ser feita uma vez para cada grupo de instruções.

O procedimento básico para desenrolar um laço consiste em replicar o corpo do laço pelo fator de desenrolamento (um número inteiro) e atualizar a construção do laço para que o incremento do mesmo seja feito em função deste mesmo fator (DONGARRA; HINDS, 1979). Havendo instruções que referenciam a variável de controle do laço, essas, ao serem replicadas precisam ter a referência também incrementada. Em alguns casos, é necessário adicionar uma instrução extra no início ou fim do laço de repetição para o caso do fator de desenrolamento não ser múltiplo exato do número de iterações do laço.

Conforme a implementação mostrada na figura 3.4, o laço original manipula um vetor indexado pela variável de controle *i*, que vai de 1 a 100 e executa uma linha interna uma vez para cada posição do vetor. No laço desenrolado (ao lado), 5 execuções internas são realizadas e o incremento da variável de controle também acompanha este valor. Neste exemplo, onde o laço de repetição foi desenrolado para um fator de 5, o número de comparações feitas é reduzido de 100 para 20.

```
! Laço original                !Laço desenrolado em 5 níveis

do i=1, 100                    do i=1, 100, 5
  c(i) = a(i) + b(i)             c(i) = a(i) + b(i)
end do                           c(i+1) = a(i+1) + b(i+1)
                                 c(i+2) = a(i+2) + b(i+2)
                                 c(i+3) = a(i+3) + b(i+3)
                                 c(i+4) = a(i+4) + b(i+4)
                               end do
```

Figura 3.4: Código comparativo - laço original e laço desenrolado

O desenrolamento de um laço tem influência direta no tempo de execução de um programa, uma vez que diminui o *overhead* do controle do laço (condição de saída e incremento) e também oportuniza melhores condições para execução de paralelismo de instruções. Por outro lado, laços de repetição com grandes conjuntos de instruções e desenrolados para um número muito grande de níveis introduzem alguns efeitos colaterais à aplicação, como por exemplo, aumento no número de instruções, o que prejudica o uso da memória *cache* e da TLB (*Translation Lookaside Buffer*). Laços menores e desenrolados para um fator aceitável de níveis, por outro lado, favorecem o uso da memória *cache* e da TLB. O tamanho aceitável de níveis está diretamente ligada à arquitetura de execução.

Um outro efeito colateral do uso de laços de repetição desenrolados é o fato de que, enquanto potencializam o desempenho, podem degradar a legibilidade do código. Desenrolar um laço de repetição manualmente é uma tarefa bastante propensa à adição de erros involuntários (dado a complexidade do código gerado). Em função disso, a automatização de tal técnica é muito importante (OVERBEY et al., 2005; JOHNSON et al., 2006).

É comum que essa técnica seja implementada por compiladores, reestruturando o código em baixo nível (AHO et al., 2006). Contudo, nem sempre o número de iterações de um laço pode ser detectado pelo compilador (BACON; GRAHAM; SHARP, 1994) e podem existir situações nas quais as otimizações do compilador não possam ser utilizadas (em função de efeitos colaterais em outras partes do código).

Normalmente, os gargalos de desempenho estão localizados em partes específicas de uma aplicação (MCCONNELL, 1993). Ou seja, as otimizações terão impacto significativo apenas em algumas partes do código. Em boa parte dos casos, laços de repetição representam estes gargalos. Considerando essa idéia, uma aplicação pode ser construída com um foco maior na sua legibilidade e uma vez que a mesma é considerada funcional, as refatorações ligadas ao desempenho são utilizadas para, de forma automatizada, reestruturarem o código e assim alcançar maiores requisitos de desempenho. Tais reestruturações podem ser realizadas inclusive em uma versão do código preparada para uma arquitetura específica (mantendo-se a versão original). O uso de técnicas de refatoração ligadas ao desempenho (entre elas *loop unrolling*) é pontencializado com a automatização das mesmas.

3.2 Infraestrutura Disponível

A construção de ações de refatoração para a ferramenta Photran é feita em 3 passos: pré-validação, manipulação e pós-validação 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). A utilização deste tipo de ferramenta abstrai do programador a ação direta sobre o código fonte, respeitando eventuais dependências e interligações entre os nós.

Para implementar e integrar uma ação de refatoração ao Photran, faz-se necessário

estender o comportamento de algumas classes (CHEN; OVERBEY, 2008). 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 que a chamada do usuário seja feita a partir do menu principal) e *IEditorActionDelegate* (permitindo que a chamada do usuário seja feita a partir 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. Essa classe é estendida de *AbstractFortranRefactoringWizard* e basicamente se responsabiliza pela construção gráfica da janela do assistente. Seu método construtor recebe como parâmetro um objeto para a terceira classe necessária: a ação de refatoração.

A principal classe a ser codificada é a ação da refatoração. Essa classe estende a superclasse *FortranRefactoring* (classe específica do Photran) que por sua vez estende a classe *Refactoring* (que faz parte do *framework* do Eclipse). Nessa classe, quatro métodos precisam ser codificados:

- ***getName()***: como seu próprio nome já indica, é responsável por fornecer o título da refatoração para que venha a ser utilizado em janelas e caixas de diálogo (do Eclipse).

- ***doCheckInitialConditions()*** e ***doCheckFinalConditions()***: são métodos chamados antes de iniciar a ação e após sua conclusão, respectivamente. Podem ser utilizados para a realização de pré e pós-validações. Ambos podem 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()*** este é o método principal, é nele que são implementadas as ações que modificam a AST do código fonte. Ao término de tais modificações, este método chama dois outros métodos: *addChangeFromModifiedAST()* (para adicionar as modificações feitas à AST utilizada pelo Photran) e *releaseAllASTs()* (para forçar o Photran atualizar os controles visuais da AST assim como sua validação).

Uma vez que as classes estão implementadas, é necessário alterar o arquivo *manifest*

(*plugin.xml*) para indicar ao Eclipse que existem novos pontos de extensão que precisam ser disponibilizados. O apêndice A descreve os pontos de extensão que precisam ser descritos no arquivo *manifest* e apresenta um exemplo do mesmo.

3.3 Automatização de Refatorações

Essa seção apresenta as implementações que foram feitas para diferentes técnicas de refatoração aplicáveis a código Fortran. As técnicas utilizadas possuem diferentes objetivos, entre eles: evolução do código (substituição de construções obsoletas), design 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). Na seção 3.1 as técnicas, sua motivação e utilidade foram apresentadas. A seguir são discutidos aspectos técnicos ligados à mecânica (breve descrição das etapas necessárias à refatoração) e à implementação (forma como automatização foi feita e como a infraestrutura do Photran foi utilizada).

3.3.1 Substituir Operadores Obsoletos (*Replace Obsolete Operators*)

A substituição de operadores obsoletos consiste na alteração de símbolos obsoletos da linguagem, neste caso operadores relacionais, por suas construções mais recentes. É uma técnica ligada à evolução e design do código fonte.

3.3.1.1 Mecânica

A mecânica de funcionamento da substituição de operadores obsoletos consiste em percorrer a árvore sintática em busca de nós que representem operadores. Ao identificar tais nós, deve-se verificar se são operadores relacionais e qual é o texto de seu *token* (*token* são nós folha da AST, representam comandos ou nomes, por exemplo, variáveis, cadeias de caracteres, valores, etc.). Ao identificar uma construção obsoleta, o texto do *token* correspondente deve ser alterado para o novo formato do operador relacional. Uma vez que somente nós da AST do tipo operadores são alterados, não há risco da aplicação apresentar mau comportamento após a refatoração.

3.3.1.2 Projeto e Implementação

O *framework* do Photran dispõe de uma classe denominada *ASTOperatorNode* que representa um nó do tipo “operador”. Essa classe disponibiliza métodos para validar o

tipo de operador que o objeto em questão representa (ex.: *hasLtOp()* retorna verdadeiro se o operador em questão é um “menor que”).

Conforme discutido anteriormente, é necessário visitar todos os nós da árvore sintática que são operadores e verificar o tipo de operação que realizam. Ao detectar operadores relacionais em antigas construções, o objeto nó terá o texto do seu único *token* (objetos da classe *ASTOperatorNode* possuem um único *token* que representa o texto do operador) alterado para o texto equivalente na nova construção, conforme demonstrado na tabela 3.1. Na figura 3.5 é possível visualizar o código fonte do método que visita os nós da árvore sintática e realiza as alterações necessárias.

```
public void visitASTNode(IASTNode node)
{
    if (node instanceof ASTOperatorNode)
        replaceOperatorIn((ASTOperatorNode)node);

    traverseChildren(node);
}

private void replaceOperatorIn(ASTOperatorNode op)
{
    if (op.hasLtOp()) {
        setText(op, "<");
    } else if (op.hasLeOp()) {
        setText(op, "<=");
    } else if (op.hasEqOp()) {
        setText(op, "==");
    } else if (op.hasNeOp()) {
        setText(op, "/=");
    } else if (op.hasGtOp()) {
        setText(op, ">");
    } else if (op.hasGeOp()) {
        setText(op, ">=");
    }
}

private void setText(ASTOperatorNode op, String newText)
{
    op.findFirstToken().setText(newText);
    changedAST = true;
}
```

Figura 3.5: Seção de código da refatoração *Replace Obsolete Operators*

3.3.2 Introduzir atributo *INTENT* (*Introduce INTENT*)

A introdução do atributo *INTENT* 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. Um argumento pode ter basicamente duas formas de utilização: ser referenciado ou ser modificado.

Algumas formas de se modificar o conteúdo de um argumento podem ser (ISO, 1997): um comando de atribuição, utilização do parâmetro como índice em um laço de repetição indexado, através de entrada de dados (*READ*) ou então por meio de chamadas de subprogramas em que o atributo *INTENT(OUT)* ou *INTENT(INOUT)* é utilizado. Há outras formas de atribuição, menos frequentes no código, que não serão consideradas neste estudo (ISO, 1997).

Se o argumento não sofre nenhum tipo de modificação (considerando as formas citadas anteriormente) mas é referenciado, por exemplo, em uma expressão ou comando de consulta ao seu valor, então pode-se inferir que o argumento é apenas referenciado. Em função da utilização que um argumento tem, podemos 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 técnica *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 (deve declarar o comando *IMPLICIT NONE* exigindo que todas as declarações explicitem o tipo de dado). Este requisito garante que todos os argumentos estejam declarados (com seu tipo bem definido) evitando que essa técnica precise se preocupar também com a definição dos tipos (tarefa que já é contemplada pela técnica *Introduce Implict None*, detalhada na subseção 2.3.2).

3.3.2.1 Projeto e Implementação

A implementação da técnica *Introduce INTENT* requer inicialmente um conjunto de validações, dentre elas: verificar se o escopo selecionado pelo programador representa um subprograma, se possui parâmetros e se utiliza da declaração explícita de tipo. Para tanto, foi definido um atributo privado (*subprogram*) na classe que implementa a refatoração

(*IntroIntentRefactoring*) para guardar o nó da AST que representa o subprograma selecionado pelo programador e que será manipulado. Este atributo é declarado para um tipo especial de classe implementada pelo *framework* do Photran que representa um escopo no código, a classe *ScopingNode*.

Um objeto da classe *ScopingNode* representa uma unidade do código Fortran que agrupa um conjunto de outras unidades. São exemplos de unidades Fortran descendentes de *ScopingNode*: *ASTMainProgramNode* (*PROGRAM*, corpo do programa principal), *ASTFunctionSubProgramNode* (*FUNCION*, corpo de uma função), *ASTSubroutineSubprogramNode* (*SUBROUTINE*, corpo de um procedimento), *ASTModuleNode* (*MODULE*, corpo de um módulo), etc. A classe *ScopingNode* oferece um conjunto de métodos relacionados à unidade do código que representa, como por exemplo, obter a lista de variáveis declaradas dentro do escopo.

Para validar se o escopo selecionado pelo programador é um subprograma, são utilizados dois métodos: *isSubroutine()*, que verifica se a classe do escopo selecionado é uma instância de *ASTSubroutineSubprogramNode*, e *isFunction()* que verifica se a classe do escopo selecionado é uma instância de *ASTFunctionSubprogramNode*. Se o resultado de um dos dois métodos for verdadeiro, então a seleção feita pelo programador é válida.

O passo seguinte, considerando que o conteúdo selecionado pelo programador é um subprograma, é verificar se o mesmo exige a indicação explícita de tipo nas variáveis e parâmetros. Isso irá garantir que todo parâmetro utilizado esteja declarado de forma que a ação *Introduce Intent* não precise se preocupar com a necessidade de realizar primeiramente a declaração das variáveis. O método *isImplicitNone()* da classe *ScopingNode* realiza essa verificação.

A última validação é descobrir, dentre as declarações do escopo, quais representam argumentos do subprograma. Uma classe especial, *Definition*, é provida pelo VPG do Photran. Um objeto *Definition* contém informações sobre a declaração de uma variável. É possível, por exemplo, a partir de um *token* que representa a utilização de uma variável, obter sua declaração. Pode-se construir uma lista com todas as definições do escopo que representam argumentos. Essa lista servirá de base para o processamento que irá definir para cada declaração se ela é alterada, referenciada ou alterada e referenciada. O código da figura 3.6 demonstra como a lista de definições pode ser obtida a partir do escopo e de que forma seu conteúdo pode ser pesquisado.

```
List<Definition> definicoes = subprogram.getAllDefinitions();
List<Definition> defQueSaoArgumentos = new LinkedList<Definition>();

for (Definition def : definicoes){
    if (def.isSubprogramArgument()) {
         defQueSaoArgumentos.add(def);
    }
}
```

Figura 3.6: Seção de código da refatoração *Introduce INTENT*

Uma vez que as declarações dos argumentos são conhecidas é preciso saber, para cada uma delas, de que forma seu conteúdo é acessado/alterado. Para tanto, são definidos dois métodos recursivos cujo objetivo é percorrer os nós da AST (dentro do escopo do subprograma) à procura de construções que alteram ou que apenas referenciam as declarações.

Os métodos *hasAssigment()* e *isReferenced()* recebem como parâmetro uma definição de argumento e um nó do subprograma. De forma recursiva, cada nó recebido é pesquisado à procura de classes que caracterizem alterações ou referências ao código (*ASTAssignmentStmtNode* e *ASTReadStmtNode*, por exemplo). Ao encontrar nós que satisfaçam aos critérios, a recursividade é encerrada e o resultado positivo é retornado. Ao final deste processo é possível inferir, para cada declaração de parâmetro, se o mesmo é modificado, referenciado ou ambos (dentro do escopo do subprograma refatorado).

Antes de concluir a refatoração é necessário verificar se os argumentos do subprograma em questão não são submetidos a outros subprogramas e, se o são, é preciso então descobrir e verificar de que forma estes argumentos são tratados. Independente do que foi descoberto no primeiro passo, essa nova informação precisa ser considerada em conjunto. Por exemplo, suponhamos que um determinado argumento é apenas referenciado no corpo do procedimento, porém é submetido a uma função onde é utilizado o atributo *INTENT(OUT)*. Neste caso, o argumento será refatorado para considerar *INTENT(INOUT)*.

A tarefa de descobrir se existem chamadas a outros subprogramas e se nessas chamadas são passados argumentos do subprograma principal precisa de duas informações: as interfaces dos subprogramas utilizadas na aplicação e as interfaces das chamadas a subprogramas feitas a partir do subprograma que está sendo refatorado. Uma classe auxiliar foi construída de forma a guardar essas informações: *SubProgramInformation*.

A classe *SubProgramInformation* foi especificada com o objetivo de guardar (para

posterior consulta) o nome do subprograma (subrotina ou função), o nome dos parâmetros e o tipo de *INTENT* de cada um. Duas listas de *SubProgramInformation* são construídas. A primeira guarda todos os subprogramas, parâmetros e tipos de *INTENT* que são chamados a partir do subprograma refatorado. A segunda guarda todas as interfaces disponíveis na aplicação (inclusive em outros arquivos do projeto, além daquele que está sendo refatorado). Essas listas são construídas ainda na fase de pré-validação (no método *doCheckInitialConditions()*).

Uma vez que as duas listas foram alimentadas, parte-se da lista de subprogramas chamados e pesquisa-se a definição de argumentos na lista de interfaces da aplicação. Essa informação é usada em conjunto com o resultado dos métodos *hasAssigment()* e *isReferenced()* para inferir o tipo de *INTENT* que cada parâmetro irá receber.

De forma a efetivar a ação de refatoração, o atributo *INTENT* e o tipo devidamente identificado (*IN*, *OUT* ou *INOUT*) são incluídos logo após a definição do tipo da variável. Um novo nó é construído utilizando-se do método auxiliar *parseLiteralStatement()* (da classe *FortranRefactoring*), que a partir de valor literal (*string*) produz um nó para a AST. Este nó substitui a declaração existente com o método da AST *replaceWith()*.

Um detalhe a ser observado é a forma como o parâmetro foi declarado. Na figura 3.7 são demonstradas 3 formas de se declarar um mesmo atributo (todas elas válidas para o Fortran). No primeiro caso (*soma*), antes de introduzir o *INTENT* é preciso adicionar o sinal de "::" (duplo dois pontos) entre o tipo e o nome da variável. No segundo caso (*minimo* e *maximo*) há uma declaração múltipla. Essa situação não é suportada na versão atual (da ação de refatoração) e em existindo, o programador é alertado de que múltiplas declarações não são suportadas. Por último, (*media*) é uma declaração onde não é necessário fazer nenhuma alteração a não ser incluir o parâmetro *INTENT*.

```
SUBROUTINE FOO(soma, media, minimo, maximo)
   real soma
   real :: minimo, maximo
   real :: media

   !Implementação

END SUBROUTINE FOO
```

Figura 3.7: Exemplo de declaração de parâmetros

3.3.3 Extrair Subrotina (*Extract Subroutine*)

Para extrair um subprograma, o programador deverá selecionar no editor de código um conjunto de linhas e será interrogado por um assistente acerca do nome do novo subprograma. A ferramenta deve verificar se o nome fornecido não está sendo utilizado e, se estiver, o programador deve ser alertado para escolher outro nome.

O código a ser extraído deve respeitar os limites dos blocos de código, ou seja, a aplicação não pode permitir que, por exemplo, um comando *DO* (abertura de um laço de repetição) seja separado do seu finalizador *END DO*. Neste sentido, os desvios rotulados *GO TO* constituem um problema, uma vez que podem direcionar o fluxo de execução para qualquer região acima ou abaixo do código da aplicação. Havendo qualquer rótulo ou instrução de desvio rotulada no trecho de código escolhido, a ferramenta não deve permitir a extração.

Declarações de variáveis somente podem ser extraídas se as únicas referências a elas estiverem também contidas no bloco de código a ser extraído. Variáveis que são referenciadas unicamente no bloco extraído podem ser transformadas em variáveis locais e a declaração externa pode ser removida. Variáveis que são referenciadas fora do bloco extraído precisam ser transformadas em argumentos que devem ser fornecidos na chamada ao subprograma.

Deve ser considerada a existência de uma cláusula *CONTAINS* que no código de Fortran delimita o local onde os subprogramas são declarados. Na existência da cláusula *CONTAINS*, a nova subrotina é adicionada imediatamente após ela. Em não existindo uma cláusula *CONTAINS*, então é necessário primeiramente adicioná-la.

3.3.3.1 Mecânica

Inicialmente, deve-se verificar dentro do conjunto de comandos selecionados para extração se existe a utilização de variáveis. Se não existirem variáveis, a refatoração deve simplesmente declarar um novo subprograma (*SUBROUTINE*) contendo o código selecionado pelo programador. No local onde antes existia o código que foi selecionado deve ser incluíndo uma chamada ao subprograma (no caso do Fortran, através do comando *CALL*).

Se o código selecionado para extração utilizar variáveis, é necessário observar se essas variáveis também são manipuladas ou referenciadas em áreas de código externas ao bloco

selecionado (antes ou depois). Em não existindo referências externas (fora do bloco extraído) às variáveis utilizadas, então elas simplesmente precisarão ser declaradas no subprograma que será criado. Opcionalmente, a declaração de variáveis que são utilizadas exclusivamente no bloco a ser extraído pode ser eliminada.

Existindo referências externas para as variáveis utilizadas no bloco a ser extraído, faz-se necessário transformar tais variáveis em parâmetros. Neste caso, a declaração da subrotina será acompanhada de uma lista de parâmetros e os mesmos deverão ser declarados no cabeçalho da subrotina (antes do corpo de código que foi extraído). Recomenda-se o uso do atributo *INTENT* (discutido na seção anterior), em função dos benefícios já mencionados.

A chamada ao novo subprograma deverá ser acompanhada da relação de variáveis que precisam ser passadas como parâmetro. É importante observar que a ordem dos parâmetros na chamada ao subprograma deve ser a mesma na qual o subprograma espera recebê-los. Ao final, o bloco que deu origem à extração de código pode ser removido, sendo substituído pela chamada ao novo subprograma.

3.3.3.2 Projeto e Implementação

A automatização da técnica *Extract Subroutine* exige inicialmente a construção de um assistente (*wizard*) para interrogar o programador acerca do nome que ele deseja atribuir à nova subrotina. Para tanto, implementa-se o método *doAddUserInputPages()* da classe que estende a superclasse *AbstractFortranRefactoringWizard*. Este método utiliza-se de recursos visuais para construir uma interface gráfica padrão. Neste caso, a única informação a ser solicitada é o nome da nova subrotina. A figura 3.8 demonstra a interface da janela do assistente.

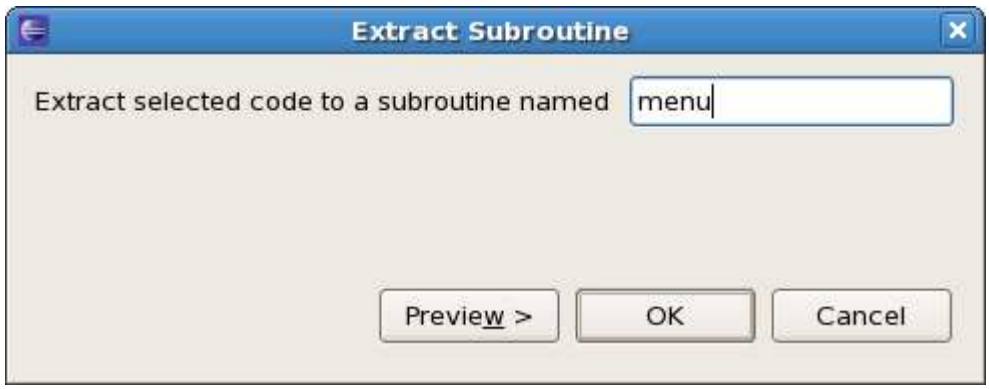

Figura 3.8: Tela do assistente da refatoação *Extract Subroutine*

A janela do assistente somente é invocada pelo Photran após a fase de validação inicial, que neste caso deve ter duas preocupações básicas: certificar-se de que o programador selecionou uma região de código e verificar se a mesma é válida. O método *findEnclosingStatementSequence()* (da classe *FortranRefactoring*) é responsável por retornar uma lista de nós representando a região selecionada. A validação da lista de nós é feita considerando a existência de declarações cujas variáveis são utilizadas externamente ao bloco selecionado (o que não é permitido) e a existência de construções como *DO..END DO* (que devem ser iniciados e finalizados dentro do bloco selecionado). Desvios de fluxo rotulado (*GO TO*) nessa implementação também não são permitidos.

Ainda na fase de validação, é preciso identificar a existência da cláusula *CONTAINS*. Para que um programa Fortran conheça seus subprogramas, eles devem ser declarados logo após essa cláusula. A existência do comando *CONTAINS* não é obrigatória (é possível que o programa não tenha nenhum subprograma declarado), mas deve ser verificada, pois é após ela que a nova subrotina será declarada. A inexistência da cláusula *CONTAINS* indica que a mesma precisa ser declarada. Neste caso, é preciso guardar o nó representando o final do programa (*END PROGRAM*), pois a cláusula *CONTAINS* e a nova subrotina serão incluídos imediatamente antes dele.

Deve-se ressaltar que a janela do assistente é invocada pelo Photran após a fase de validação inicial. Em função disso, a verificação do nome fornecido pelo programador deverá ser validada pelo próprio assistente através da chamada de um método público da ação de refatoração que verificará a disponibilidade de utilização do nome fornecido. Caso o nome já seja utilizado, a própria janela do assistente pode avisar o programador solicitando a indicação de outro nome.

Uma vez que o código a ser extraído foi validado, é necessário construir uma lista de variáveis que são acessadas pelo bloco selecionado. Por meio de uma pesquisa recursiva em todos os nós selecionados são descobertas as variáveis utilizadas (uma variável é um nó pertencente à classe *ASTNameNode* ou *ASTDataRefNode*). Para cada variável descoberta e adicionada na lista guarda-se em uma outra lista um indicativo (verdadeiro ou falso) de que a variável é ou não referenciada fora do bloco selecionado.

Para descobrir se uma variável é utilizada/referenciada fora do bloco selecionado, o método *isVariableReferencedIntoScope()* foi desenvolvido. Basicamente, ele percorre recursivamente todos os nós do escopo (programa ou subprograma) no qual o bloco a

ser extraído se encontra, exceto as linhas selecionadas para extração. O método recebe o nó que representa uma variável referenciada dentro da seleção feita pelo programador e retorna verdadeiro caso ela tamém seja utilizada fora da seleção, ou falso caso não seja.

Em função do processamento da lista de variáveis e de sua indicação de utilização ou não utilização fora do bloco a ser extraído, é possível inferir quais variáveis serão consideradas parâmetros (aquelas cujo indicador de utilização for verdadeiro) e quais serão apenas declaradas como variáveis locais (demais variáveis). No caso das variáveis que serão consideradas parâmetros, é preciso submetê-las aos mesmos procedimentos discutidos na subseção 3.3.2, relativos à adição do atributo *INTENT*.

Para definir o *INTENT* de uma variável, deve-se verificar se a mesma é referenciada e se é alterada. A combinação dessas informações irá indicar o tipo de *INTENT* que será utilizado para a mesma. Variáveis apenas referenciadas são consideradas *INTENT(IN)*, variáveis que são apenas alteradas são consideradas *INTENT(OUT)* e variáveis que são alteradas e referenciadas são consideradas *INTENT(INOUT)*. Além disso, é preciso verificar se o código extraído não realiza chamadas a outros subprogramas (subrotinas ou funções), sendo que neste caso o valor dos atributos *INTENT* destes outros subprogramas também precisam ser considerados.

Para finalizar, é necessário criar o nó correspondente à nova subrotina. Para tanto, o nome fornecido pelo programador (e já validado) e a lista de variáveis que representam os parâmetros são concatenados ao comando *SUBROUTINE* para formar a base da nova subrotina. Na sequência, todas as variáveis utilizadas no interior da subrotina são declaradas (utilizando o mesmo formato da sua declaração original), aquelas que são parâmetros recebem o atributo *INTENT* correspondente. Por fim, o bloco de comandos extraídos também é concatenado à nova subrotina e ao comando *END SUBROUTINE*.

O resultado da concatenção anterior é transformado em um nó (*ASTSubroutineSubprogramNode*) através do método *parseLiteralStatement()* e é adicionado logo após o nó *ASTContainsStmtNode* (se o comando *CONTAINS* existir) ou então antes do *ASTEndProgramStmtNode* (*END PROGRAM*), neste caso concatenando-se ainda o comando *CONTAINS*. O último procedimento a ser feito é substituir o primeiro nó da seleção feita pelo programador por uma chamada *CALL* (concatenada ao nome fornecido e a lista de parâmetros). Os demais nós são removidos através do método *removeFromTree()*.

Para reorganizar (endentar) o código, pode-se utilizar o método estático *reindent()* da

classe auxiliar *Reindenter*. Baseado no nó que foi adicionado e o objeto que representa a AST (ambos passados por parâmetro), o método *reident()* reorganiza a representação textual da AST (na região modificada).

3.3.4 Desenrolar Laço de Repetição (*Loop Unrolling*)

O desenrolamento de um laço de repetição pode ser aplicado a qualquer tipo de estrutura de repetição (com pré-validação, com pós-validação ou indexada). Neste trabalho, contudo, considera-se apenas laços de repetição indexados, ou seja, laços que são controlados por um índice (variável) que é incrementado em uma unidade a cada iteração.

O programador deverá fornecer um laço de repetição através de uma seleção de código. Essa seleção, para ser válida deverá começar com uma instrução *DO* e ser encerrada por uma instrução *END DO*, formando um bloco de código único. Para ser válido, o laço selecionado deverá ser do tipo indexado, ou seja, deverá conter uma variável de controle com uma expressão de inicialização e um limite.

O limite pode ser um valor inteiro, uma variável e até mesmo uma expressão. O uso de expressões, embora seja permitido, não é recomendado, uma vez que irá forçar o processador resolver a expressão a cada iteração, desperdiçando processamento. Se o limite for uma expressão o programador poderia simplesmente retirá-la da estrutura de controle do laço de repetição e utilizar uma variável temporária para computar seu resultado (antes do laço). Essa variável de controle pode então ser utilizada como limite do laço de repetição (essa é uma boa prática que melhora o desempenho do código).

O número de níveis (fator de desenrolamento do laço) e o nome de uma variável auxiliar também deverão ser informados pelo programador. A variável auxiliar será usada para computar se o limite do laço é múltiplo do fator informado. Neste caso, deve ser permitido que a variável informada já exista, pois mais de um laço pode estar sendo desenrolado na mesma aplicação e é desejável reaproveitar a variável temporária.

3.3.4.1 Mecânica

Para demonstrar a mecânica da refatoração *Loop Unrolling* será utilizado um laço de repetição simples (e não desenrolado) representado na figura 3.9. O laço possui um índice de controle representado pela variável *i* que é iniciado por um valor armazenado na variável *inicio* e executa o número de vezes definidas na variável *final*. O corpo do laço é composto por dois comandos, um deles mais simples (acumula 5 ao valor existente na

variável *cont* e não utiliza a variável de controle do laço *i*) e outro comando que indexa um vetor (*a*), utilizando-se da variável de controle do laço (*i*).

```
do i = inicio, final
   cont = cont + 5
   a(i) = a(i) + cont
end do
```

Figura 3.9: Exemplo de laço de repetição (não desenrolado)

Inicialmente é preciso certificar-se de que o número de vezes que o programador deseja desenrolar o laço (variável *fator*) seja múltiplo do valor final do laço (variável *final*). Caso não seja múltiplo, é preciso executar o corpo do laço ao menos uma vez e iniciar a execução do laço desenrolado na segunda iteração. Isso pode ser feito introduzindo uma estrutura condicional (*IF THEN*) antes do laço desenrolado.

A estrutura condicional adicionada deverá verificar por meio do comando *MOD* (resto da divisão) se o limite do laço é múltiplo do número de níveis do desenrolamento. Se não for, então o corpo do laço original deverá ser executado integralmente. Além disso, é necessário inicializar uma variável (indicada pelo programador, e aqui representada por *istart*) para indicar se as iterações do laço deverão começar a partir da segunda, ou seja, a inicialização do laço desenrolado poderá ser acrescida de uma unidade em função o comando executado pela estrutura condicional citada anteriormente. Essa variável deve conter o valor zero antes do teste condicional. A estrutura condicional adicionada pode ser visualizada pela figura 3.10.

Uma vez definido o início das iterações do novo laço de repetição, a instrução de declaração do laço de repetição deverá ser alterada em relação ao laço original com a inclusão do parâmetro de incremento, que será exatamente igual ao *fator* informado. Para que o laço seja desenrolado corretamente, o incremento do laço original não poderá ser diferente de 1.

Por fim, o último passo é repetir o corpo do laço original considerando o valor de *fator*, ou seja, se o programador escolheu desenrolar o laço em 3 níveis (*fator = 3*), o corpo do laço será replicado 3 vezes. É importante observar que qualquer instrução que vier a ser repetida e fizer uso da variável incrementada pelo laço (*i*) deverá, em cada repetição, ter seu valor incrementado. A exceção fica por conta da primeira vez, na qual a construção original permanecerá inalterada. A figura 3.10 demonstra um laço de repetição que foi

desenrolado em 3 níveis, de acordo com a mecânica proposta para essa refatoração.

```
istart = 0 !variável indicada pelo programador

! Verifica se o nº de níveis é múltiplo do limite do laço
if (mod(final,niveis) /= 0) then

   cont = cont + 5                !Instrução original
   a(inicio) = a(inicio) + cont   !Instrução original alterada

   !Variável auxiliar indicando que o laço deve
   !começar a partir da 2ª iteração
   istart = 1
end if

! O início é definido em função do valor de istart
do i = inicio + istart, n, 3
   cont = cont + 5         !Instrução original
   a(i) = a(i) + cont      !Instrução original

   cont = cont + 5         !Instrução original
   a(i+1) = a(i+1) + cont  !Instrução original alterada

   cont = cont + 5         !Instrução original
   a(i+2) = a(i+2) + cont  !Instrução original alterada
end do
```

Figura 3.10: Laço de repetição desenrolado em 3 níveis

3.3.4.2 Projeto e Implementação

A primeira validação a ser feita na implementação da refatoração *Loop Unrolling* é validar se a seleção de código feita pelo programador representa de fato um laço de repetição. A versão atual da AST do Photran não implementa o par *DO .. END DO* como sendo um único nó, o que faz com que o código da refatoração *Loop Unrolling* tenha que ser consistido por um algoritmo próprio.

Para que a seleção seja válida ela deve começar com uma instrução *DO* (classe *ASTDoConstructNode*) e ser encerrada por uma instrução *END DO* (classe *ASTEndDoStmtNode*), sendo que entre estes dois nós não poderá existir um número ímpar de outras instruções, sejam *DO* ou *END DO*. Conforme já fora mencionado anteriormente, outros blocos de construção de laços como *DO WHILE* não são considerados nessa versão da implementação.

Além de validar o tipo de construção do laço é necessário identificar a presença da variável de controle e das expressões de inicialização e finalização do laço. A existên-

cia dessas informações indicará que o laço de fato é indexado e não uma outra construção utilizando-se do comando *DO* (como *DO-CYCLE-EXIT-END DO* ou *DO-EXIT-END DO*), que também são possíveis (e utilizam-se da mesma classe para representá-los (*ASTDoConstructNode*) na AST do Photran.

Todas as informações necessárias para implementar a mecânica dessa refatoração precisam ser coletadas e armazenadas em atributos da classe, entre elas: o nó que representa a variável de controle, as expressões de inicialização e de finalização, o bloco de comandos do interior do laço e os nós que representam os comandos *DO* e *END DO* (que juntamente com o interior do laço original ao final serão removidos da AST).

Uma vez que se tem conhecimento de todas as informações necessárias e também o número de níveis e o nome da variável auxiliar, o desenrolamento do laço pode ser feito de forma bastante simples seguindo a mecânica que foi explicada anteriormente. Inicialmente um nó *ASTIfConstructNode* representando a estrutura de decisão inicial é construído. Ele irá compor o comando inicial do conjunto de instruções que irão representar o laço desenrolado. O método *getIfConstructNode()* foi definido para retornar um nó *ASTIfConstructNode* contendo o teste condicional para validar se o número de níveis informado é múltiplo da expressão de limite do laço original. Na figura 3.11 demonstra-se como o método *getIfConstructNode()* foi implementado, considerando os atributos coletados e o bloco de comandos do antigo laço que serão utilizados.

O passo seguinte é definir um bloco *DO..END DO* contendo no nó *ASTDoConstructNode* a nova expressão de inicialização (a antiga mais o valor contido na variável informada pelo programador) e ao final, a indicação de incremento do laço, que ao invés de 1 (ou vazio) do laço original agora será o valor informado para o fator de desenrolamento (número de níveis). Por fim, repete-se a lista de comandos do laço original para tantas vezes quanto determinado pela variável *fator*. A cada repetição dos blocos de comando, as expressões que fizerem uso da variável de controle são incrementadas conforme o número da repetição, lembrando que a primeira inclusão do bloco de comando não altera a variável de controle, a segunda inclusão altera em uma unidade, e assim sucessivamente.

Para finalizar, os nós representando a inicialização da variável de controle, a estrutura condicional e o bloco do laço de repetição são agrupados e adicionados no lugar da instrução *DO* (no laço original). O bloco do antigo laço e as instruções abrangidas entre ele e o comando de finalização (*END DO*) são removidos. Na figura 3.12 podemos observar

```
public ASTIfConstructNode getIfConstructNode(){
    String s = "";

    s = CRLF +
        "if (mod(" + limitControl.getText() + "," +
         String.valueOf(getProfundity()) + ") /= 0) then" +
         CRLF;

    for (int i=0; i<assignmentsList.size(); i++){
       s = s.concat(SourcePrinter.getSourceCodeFromASTNode(
                           assignmentsList.get(i))) + CRLF ;
    }

    s = s.concat(getVariableName() + " = 1 " + CRLF +
        "end if " + CRLF);

    return (ASTIfConstructNode)parseLiteralStatement(s);
}
```

Figura 3.11: Implementação do método *getIfConstructNode()*

a refatoração sendo automatizada pela ferramenta Photran. Utilizou-se o mesmo exemplo demonstrado anteriormente (na figura 3.9) para simplificar o entendimento.

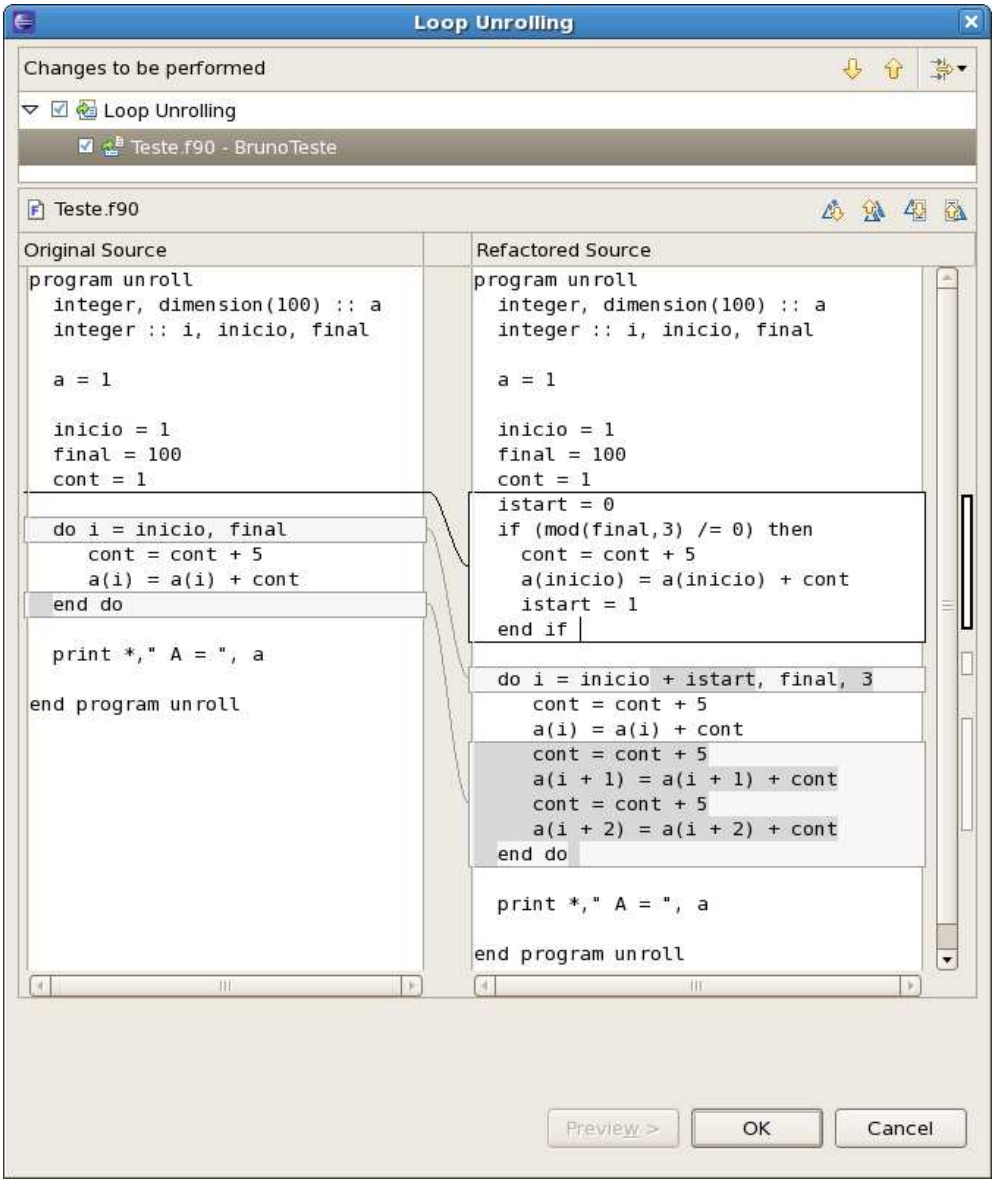

Figura 3.12: Refatoração utilizando a técnica *Loop Unrolling*

4 APLICAÇÃO E AVALIAÇÃO

4.1 Metodologia

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 (*LμMet*) vinculado à Universidade Federal de Santa Maria. O objetivo deste estudo de caso é avaliar de que forma a utilização de técnicas de refatoração podem influenciar no desempenho (de forma positiva ou negativa) da referida aplicação. As técnicas utilizadas cujos resultados serão avaliados são: *Introduce INTENT*, *Extract Subroutine* e *Loop Unrolling*, as mesmas que foram descritas, automatizadas e integradas ao IDE Photran, conforme detalhado no capítulo 3.

A aplicação alvo foi escrita em linguagem Fortran 77 e utiliza algumas construções desaconselhadas por versões mais recentes da linguagem Fortran, como por exemplo a utilização de desvios rotulados (*GO TO*) e declaração implícita de variáveis. Antes de iniciar a avaliação, o código da aplicação alvo foi preparado para receber as refatorações. Duas ações foram realizadas neste sentido: os laços de repetição a serem refatorados e que utilizavam-se de desvios rotulados foram transformados em laços do tipo *DO..END DO* (isso foi feito manualmente). A segunda adaptação no código foi a eliminação de tipos implícitos, utilizando-se para isso a refatoração *Introduce Implicit None*, disponível no IDE Photran.

Partindo-se do código original acrescido das evoluções do código citadas anteriormente, a avaliação foi organizada da seguinte forma:

- **Etapa 1**: mensurar o tempo de execução, número e o tamanho dos arquivos resultantes da execução da aplicação compilada a partir da versão original do código fonte (apenas com os laços de repetição padronizados e a introdução de declarações explícitas);

- **Etapa 2**: aplicar ao código fonte as refatorações que melhoram seu design e favorecem sua legibilidade (*Introduce INTENT* e *Extract Subroutine*) e ao término compilar a nova versão do código fonte e executar a aplicação resultante para mensurar seu novo tempo de execução, número e o tamanho dos arquivos de resultado;

- **Etapa 3** detectar gargalos no código fonte gerados em função de laço(s) de repetição muito interno(s) e aplicar o desenrolamento do(s) mesmo(s). Ao final, compilar, executar e mensurar o tempo de execução da nova versão da aplicação assim como a quantidade e o tamanho dos arquivos resultantes.

Todas as etapas foram executadas em um computador *AMD Athlon XP 2 Ghz*, com 1 *GByte* de memória. Os códigos-fonte foram compilados utilizando-se de dois compiladores diferentes: Intel Fortran (versão 8) e GCC Fortran (versão 4.1). Para isolar o efeito de eventuais otimizações dos compiladores a aplicação foi compilada com esse recurso desabilitado (opção de compilação: -O0).

Os testes foram realizados no sistema operacional *Linux Fedora*, versão do *kernel*: 2.6.22.14-72.fc6 utilizando-se do modo de operação *single user* (*runlevel* = 1). Para mensurar o tempo de execução do processo utiliza-se o comando *time*, que ao final apresenta um relatório estatístico do processo indicando seu tempo de execução.

4.2 Estudo de Caso: Aplicação para análise de dados micrometeorológicos

As técnicas de refatoração foram utilizadas no código fonte de uma aplicação utilizada para a análise de dados micrometeorológicos captados por sensores especiais de estações meteorológicas da região de Santa Maria - RS. O código fonte original da aplicação foi escrito em Fortran 77 pelo Laboratório de Micrometeorologia (*LμMet*) vinculado à Universidade Federal de Santa Maria. 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 (em média com espessura de 1 km). 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 (numeroso) 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 de grande importância para os experimentos do *LμMet*, mas dado seu tempo de vida e algumas construções obsoletas utilizadas no seu desenvolvimento apresenta uma alta complexidade de compreensão e manutenção. Essa situação representa um problema quando novas funcionalidades ou análises precisam ser adicionadas à aplicação. O código fonte original da aplicação (com a substituição dos laços rotulados e introdução explícita de declarações) possui em torno de 1130 linhas, sendo composto por um programa principal e 10 subprogramas.

O tempo de execução da aplicação alvo executada em um computador *AMD Athlon XP 2 Ghz*, com 1 *GByte* de memória é de 1 hora, 2 minutos e 20 segundos utilizando-se o compilador GCC e 59 minutos e 19 segundos utilizando-se o compilador Intel. Na tabela 4.1 pode-se observar o tempo de execução (nos diferentes compiladores) e também o tamanho do arquivo executável bem como o número e tamanho dos dados de saída produzidos pela aplicação.

Inicialmente, as 10 subrotinas existentes no programa foram refatoradas de forma a incluir os atributos *INTENT* correspondentes em cada um de seus parâmetros. Para tanto, a ação de refatoração *Introduce Intent*, integrada ao IDE Photran, foi utilizada. O código da figura 4.1 demonstra o código da subrotina *SPECDIS* após a refatoração. O código original difere daquele visualizado na figura 4.1 apenas pela ausência do atributo *INTENT*. Neste caso, dos 4 argumentos da subrotina, três deles (*N*, *K* e *VEM*) foram considerados pela ação de refatoração como sendo *INTENT(IN)*, ou seja atributos apenas referenciados no corpo da subrotina. O argumento *DSPEC* por sua vez foi considerado *INTENT(OUT)*, ou seja, apenas alterado pela subrotina.

Ainda na mesma etapa de avaliação, foram identificadas no código fonte, através de blocos de comentários, outras 17 regiões do código em condições de serem extraídas e transformadas em novos subprogramas. Para tanto, utilizou-se a ação de refatoração *Extract Subroutine*. A figura 4.2 demonstra um trecho de código do programa principal antes da refatoração e a figura 4.3 demonstra o mesmo trecho após a refatoração, bem como o código fonte do novo subprograma originado. A refatoração *Replace Obsolete Operators* detalhada na seção 3.3 também foi aplicada ao código fonte de forma a substituir o

```
SUBROUTINE SPECDIS(VEM,DSPEC,N,K)
    IMPLICIT NONE
    INTEGER, INTENT(IN)    :: N
    INTEGER, INTENT(IN)    :: K
    COMPLEX*8, INTENT(IN)  :: VEM(N)
    REAL*8, INTENT(OUT)    :: DSPEC(K+1)
    REAL*8                 :: SPEC(N),
    REAL*8                 :: FREQ, DELTAN
    INTEGER                :: I
    FREQ=16.
    DO I=1,N
        SPEC(I)=(VEM(I)*(CONJG(VEM(I))))/(N*N)
    END DO
    DELTAN=(N+0.0)/FREQ
    DO I=1, K
        DSPEC(I)=2*SPEC(I)*DELTAN
    END DO
    DSPEC(K+1)=SPEC(K+1)*DELTAN
    RETURN
END SUBROUTINE SPECDIS
```

Figura 4.1: Código da subrotina *DSPEC* após a refatoração

uso dos operadores *.GT.* e *.LE.* por suas construções equivalentes (> e <=), de forma a contribuir ainda mais com a legibilidade do código.

A avaliação do desempenho do código após a refatoração pode ser observada na tabela 4.1. Em ambos os compiladores o tempo de execução da nova versão da aplicação sofre um pequeno acréscimo. A versão compilada com o compilador GCC levou 70 segundos a mais para executar (executou 1,87% mais lentamente que a aplicação original). No segundo caso, a aplicação compilada com o compilador Intel sofre um acréscimo menor, apenas 26 segundos (o que equivale a 0,73% do tempo da aplicação original).

A terceira parte da avaliação consiste em aplicar a refatoração *Loop Unrolling* sobre o código já refatorado pela etapa dois e mensurar o eventual ganho de desempenho. Em função de que tal refatoração introduz ao código uma dificuldade adicional de compreensão, não é recomendado que a mesma seja utilizada em demasia, mas sim em locais no código que representam gargalos de execução.

Uma alusão a "lei 90/10" ou *The Matthew Effect* (MERTON, 1968) pode ser utilizada neste experimento. Pode-se considerar que, em geral, as aplicações gastam 10% do tempo para executar 90% de todo o código e os outros 90% do tempo para executar apenas 10% do código. Ou seja, grande parte do ganho de desempenho de uma aplicação pode

```
...
!CONDIÇÕES DE ESTABILIDADE********
azL=abs(z/L)
if (azL.le.0.025) zL="01"
if (azL.gt.0.025.and.azL.le.0.05)  zL="02"
if (azL.gt.0.05.and.azL.le.0.075)  zL="03"
if (azL.gt.0.075.and.azL.le.0.1)  zL="04"
if (azL.gt.0.1.and.azl.le.0.15)  zL="05"
if (azL.gt.0.15.and.azl.le.0.2)  zL="06"
if (azL.gt.0.2.and.azl.le.0.3)  zL="07"
if (azL.gt.0.3.and.azl.le.0.4)  zL="08"
if (azL.gt.0.4.and.azl.le.0.5)  zL="09"
if (azL.gt.0.5.and.azl.le.0.6)  zL="10"
if (azL.gt.0.6.and.azl.le.0.7)  zL="11"
if (azL.gt.0.7.and.azl.le.0.8)  zL="12"
if (azL.gt.0.9.and.azl.le.1.0)  zL="13"
if (azL.gt.1.0)  zL="14"
...
```

Figura 4.2: Código original (antes da extração de subrotina)

ser explorada em regiões bem específicas do código. Isso se dá em função de que boa parte do código fonte, principalmente de aplicações de alto desempenho, é composta por inicializações, cálculos, leituras de dados de entrada (o que correspondem aos 90% do código) e laços de repetição, em especial os aninhados, que quanto mais internos mais tempo de execução consomem (que representam apenas 10% do código).

O código da aplicação alvo foi estudado de forma a identificar laços de repetição que poderiam ser desenrolados. O estudo identificou 43 laços em condições de serem refatorados. Contudo, apenas um deles foi escolhido, em função de apresentar o maior número de iterações (aproximadamente 1.073.741.824 no pior caso). O laço a ser desenrolado é o mais interno (indexado pela variável i) que pode ser visualizado na figura 4.4. A constante *N*, limite para os laços de repetição (interno e externo) é definida no início do programa principal com o valor de 32768.

O laço de repetição identificado contém 3 linhas de código e foi desenrolado em 32 níveis utilizando-se a técnica *Loop Unrolling* automatizada e integrada à ferramenta Photran. De forma a separar do código original o código refatorado, este último também foi transformado em uma subrotina. É importante ressaltar que o código fonte da aplicação que foi refatorada continha também as melhorias adicionadas pelas refatorações da etapa 2, ou seja, a inclusão do atributos *INTENT* e a extração das novas subrotinas.

```
...
!CONDIÇÕES DE ESTABILIDADE********
azL=abs(z/L)

CALL CONDICOES_ESTABILIDADE(azL, zL)
...


...
SUBROUTINE CONDICOES_ESTABILIDADE(azL, zL)
      implicit none
      REAL, INTENT(IN)              :: azL
      CHARACTER(LEN=2), INTENT(OUT) :: zL

      if (azL <= 0.025) zL="01"
      if (azL > 0.025.and.azL <= 0.05)  zL="02"
      if (azL > 0.05.and.azL <= 0.075)  zL="03"
      if (azL > 0.075.and.azL <= 0.1)  zL="04"
      if (azL > 0.1.and.azl <= 0.15)  zL="05"
      if (azL > 0.15.and.azl <= 0.2)  zL="06"
      if (azL > 0.2.and.azl <= 0.3)  zL="07"
      if (azL > 0.3.and.azl <= 0.4)  zL="08"
      if (azL > 0.4.and.azl <= 0.5)  zL="09"
      if (azL > 0.5.and.azl <= 0.6)  zL="10"
      if (azL > 0.6.and.azl <= 0.7)  zL="11"
      if (azL > 0.7.and.azl <= 0.8)  zL="12"
      if (azL > 0.9.and.azl <= 1.0)  zL="13"
      if (azL > 1.0)  zL="14"

END SUBROUTINE CONDICOES_ESTABILIDADE
```

Figura 4.3: Exemplo de extração de subrotina e remoção de operadores obsoletos

```
...
do j=1,n

   auto(j)=0.
   corr(j)=0.

   ! Este é o laço escolhido para ser desenrolado
   do i=1,n
       correlacao(j,i)=turb(i)*turb(j+i-1)
       auto(j)=auto(j)+correlacao(j,i)
       if((i+j).gt.n) EXIT
   end do

end do
...
```

Figura 4.4: Laço de repetição candidato a refatoração

Após a refatoração, o tempo de execução para ambas as versões dos executáveis (compiladas com GCC e Intel) foi reduzido. No primeiro caso, a aplicação compilada com o GCC teve um tempo de execução 4,25% menor considerando a versão produzida na segunda etapa da avaliação e 2,46% menor considerando o tempo de execução da versão original (sem as refatorações aplicadas). A execução da aplicação refatorada produzida com o compilador Intel resultou em 4,25% de redução do tempo considerando a versão da segunda avaliação e 3,65% de redução considerando a aplicação original.

Os resultados produzidos (em quantidade e tamanho de arquivos) são idênticos em todas as versões executadas. O resultado completo do teste pode ser visualizado na tabela 4.1. A figura 4.5 ilustra um gráfico demonstrando os tempos de execução de todas as versões da aplicação.

Tabela 4.1: Avaliação dos tempos de execução

Ver. do Código	Linhas	Exec.(bytes)	Tempo	Nº Arq.Result.	Tam.Result.(Mb)
Original (GCC)	1130	31849	01:02:20	331	70,54
Original (Intel)	1130	629130	00:59:19	331	70,54
Etapa 2 (GCC) [1]	1420	34933	01:03:30	331	70,54
Etapa 2 (Intel) [1]	1420	633149	00:59:45	331	70,54
Etapa 3 (GCC) [2]	1560	39029	01:00:48	331	70,54
Etapa 3 (Intel) [2]	1560	637021	00:57:09	331	70,54

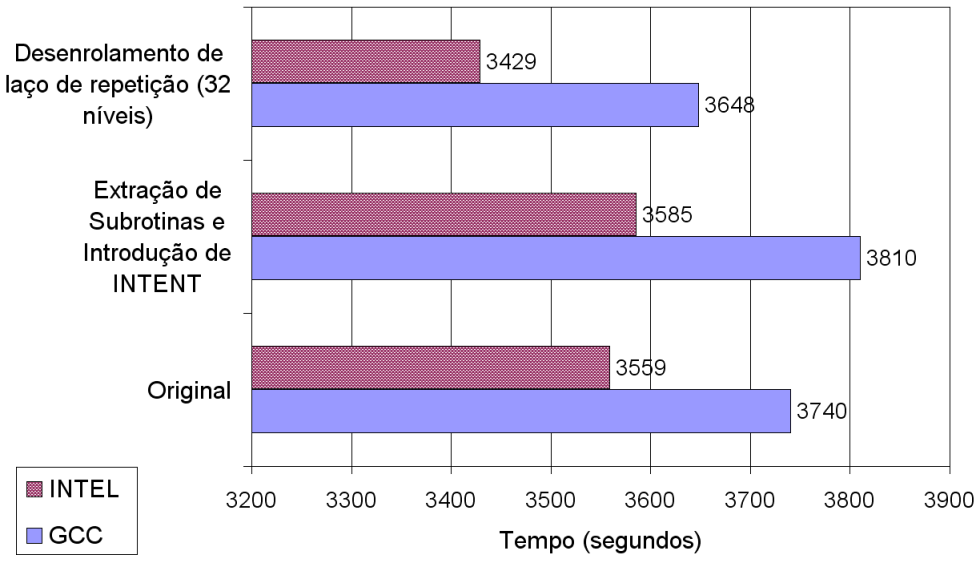

Figura 4.5: Gráfico comparativo dos tempos de execução

[1] Extração de subrotinas e introdução de *INTENT*

[2] Todas as melhorias da etapa 1 e também o desenrolamento de laço de repetição (em 32 níveis)

4.3 Conclusão da Avaliação

A avaliação dos tempos e resultados obtidos após a execução de todas as etapas permite realizar duas conclusões significativas:

- A primeira é que um código mais legível, que expressa melhor seu objetivo, que utiliza-se de melhores construções não necessariamente significa um código que perde desempenho. O teste executado demonstra que o código com melhores características de legibilidade possui um desempenho médio 1,3% inferior (o que não chega a representar 1 minuto no tempo total de execução da aplicação utilizada no estudo). Entende-se que essa é uma degradação mínima considerando os benefícios que um código melhor organizado pode apresentar ao longo do seu tempo de vida. Este tempo pode vir a ser compensado com a aplicação de melhorias específicas ligadas ao desempenho ou ainda outras, que somente se tornarão possíveis de serem identificas a partir de uma melhor organização do codigo-fonte.

- A segunda conclusão diz respeito ao fato de que é possível aplicar refatorações com vistas ao ganho de desempenho. O experimento realizado com a refatoração *Loop Unrolling* sobre apenas um laço de repetição (dos 43 identificados) permitiu reduzir o tempo de execução em média em 4,3%. É importante ressaltar que este ganho de desempenho é obtido considerando inclusive as demais refatorações que melhoram o design de código.

Além das conclusões relativas ao tempo de execução é preciso registrar que a utilização de técnicas automatizadas agilizam e dão maior segurança ao programador. O desenrolamento de laço de repetição executado durante a terceira etapa da avaliação dificilmente teria sido viabilizado de forma manual. O uso da ferramenta também agrega a funcionalidade do programador pré-visualizar o código refatorado e decidir ou não por sua utilização.

5 CONCLUSÃO

A programação de alto desempenho é uma área da ciência da computação que se preocupa com o emprego de metodologias e técnicas para a execução eficiente de aplicações. Isso abrange 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*.

Neste sentido, técnicas de refatoração ligadas à computação de alto desempenho têm sido exploradas, na tentativa de prover melhores práticas e evoluções ao *software* existente. A degradação natural pela qual o código fonte de aplicações sofre com o tempo, pode ser amenizada aplicando-se técnicas de refatoração.

O processo de refatoração em si pode ser executado de forma manual, mas o ganho de qualidade em escala se dá quando técnicas são automatizadas e integradas à 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 consequentemente o custo de desenvolvimento.

O presente trabalho explora a utilização de técnicas de refatoração sobre aplicações de alto desempenho escritas em linguagem Fortran, objetivando melhorar o design de código e detectar oportunidades de ganho de desempenho. As técnicas estudadas são 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.

As principais contribuições do trabalho são a identificação e automatização de técnicas de refatoração para linguagem Fortram. Quatro técnicas são automatizadas e integradas à ferramenta Photran: substituição de operadores obsoletos, introdução do parâmetro

INTENT, extração de subprograma e desenrolamento de laço de repetição.

Para avaliar o impacto da utilização de técnicas de refatoração em aplicações de alto desempenho escritas em Fortran, realizou-se um estudo de caso. O caso compreende o código fonte de uma aplicação para análise de dados micrometeorológicos escrita em linguagem Fortran 77. O tempo de execução da referida aplicação é mensurado antes e depois da aplicação de sucessivas refatorações.

Os resultados da avaliação demonstram que a utilização de técnicas de refatoração ligadas à evolução e ao design de código, como *Introduce INTENT* e *Extract Subroutine*, por exemplo, não possuem influência negativa sobre o desempenho da aplicação. Ou seja, ao passo que tornam o código mais legível e melhor organizado, não introduzem gargalos que reduzem de forma perceptível o desempenho da aplicação.

Da mesma forma, a técnica *Loop Unrolling*, desenvolvida especialmente em função do ganho de desempenho, se mostrou eficaz e comprovou o ganho de desempenho depois de utilizada. Neste caso é importante destacar que o uso deve ser moderado em função de que ao passo que o desempenho é beneficiado a legibilidade do código pode ser prejudicada.

A utilização do Photran como ferramenta de apoio à codificação e automação 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 enorme quantidade de código Fortran (em especial de aplicações científicas e de alto desempenho) e o limitado número de ferramentas de apoio ao desenvolvimento com técnicas de refatoração integradas.

Em relação à ferramenta Photran ainda há inúmeras questões em aberto que vão desde otimizações nos mecanismos que analisam o código fonte, novos recursos para a ferramenta e principalmente a automatização de outras técnicas de refatoração. 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.

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.]: Springer Publishing Company, Incorporated, 2008.

AHO, A. V.; LAM, M. S.; SETHI, R.; ULLMAN, J. D. **Compilers**: principles, techniques, and tools. 2.ed. [S.l.]: Addison Wesley, 2006.

AMBLER, S. W.; SADALAGE, P. J. **Refactoring Databases**: evolutionary database design. [S.l.]: Addison-Wesley Professional, 2006.

ARNOLD, R. S. An Introduction to Software Restructuring. In: ARNOLD, R. S. (Ed.). **Tutorial on Software Restructuring**. [S.l.]: IEEE Press, 1986.

ASTELS, D. Refactoring with UML. In: THIRD INTERNATIONAL CONFERENCE ON EXTREME PROGRAMMING AND FLEXIBLE PROCESSES IN SOFTWARE ENGINEERING, 2002, Alghero, Italia. **Anais...** [S.l.: s.n.], 2002. p.67–70.

BACKUS, J. W. The IBM 701 Speedcoding System. **Journal of the ACM**, [S.l.], v.1, n.1, p.4–6, 1954.

BACON, D. F.; GRAHAM, S. L.; SHARP, O. J. Compiler Transformations for High-Performance Computing. **ACM Computing Surveys**, Nova Iorque, EUA, v.26, n.4, p.345–420, 1994.

BEATON, W.; RIVIERES, J. des. **Eclipse Platform Technical Overview**. [S.l.]: The Eclipse Foundation, 2006.

BOEHM, A. M.; SEIPEL, D.; SICKMANN, A.; WETZKA, M. Squash: a tool for analyzing, tuning and refactoring relational database applications. In: INTERNATIONAL

CONFERENCE ON APPLICATIONS OF DECLARATIVE PROGRAMMING AND KNOWLEDGE MANAGEMENT, 17., 2007, Berlin, Alemanha. **Anais...** Springer-Verlag, 2007.

BOGER, M.; STURM, T.; FRAGEMANN, P. Refactoring Browser for UML. In: INTERNATIONAL CONFERENCE NETOBJECTDAYS ON OBJECTS, COMPONENTS, ARCHITECTURES, SERVICES, AND APPLICATIONS FOR A NETWORKED WORLD, 2003, Londres, Inglaterra. **Anais...** Springer-Verlag, 2003. p.366–377.

BOIS, B. D. **A Study of Quality Improvements by Refactoring**. 2006. Tese de Doutorado — Universiteit Antwerpen, Antwerpen, Belgica.

CHEN, N.; OVERBEY, J. **Photran 4.0 Developer's Guide**. [S.l.: s.n.], 2008.

COPELAND, T. **Generating Parsers with JavaCC**. [S.l.]: Centennial Books, 2007.

CORNELIO, M. L. **Refactorings as Formal Refinements**. 2004. Tese de Doutorado — Universidade Federal de Pernambuco, Recife, Brasil.

DE, V. **A Foundation for Refactorin Fortran 90 in Eclipse**. Urbana-Champaign, EUA, 2004.

DELAMARO, M. E. **Como Construir um Compilador Utilizando Ferramentas Java**. [S.l.]: Novatec Editora Ltda, 2004.

DEURSEN, A. van; MOONEN, L.; BERGH, A. van den; KOK, G. Refactoring Test Code. In: SECOND INTERNATIONAL CONFERENCE ON EXTREME PROGRAMMING AND FLEXIBLE PROCESSES IN SOFTWARE ENGINEERING, 2001, Portland, EUA. **Anais...** IEEE Computer Society, 2001.

DONGARRA, J.; HINDS, A. R. Unrolling Loops in FORTRAN. **Software Practice and Experience**, [S.l.], v.9, n.3, p.219–226, 1979.

DONGARRA, J.; LUSZCZEK, P.; PETITET, A. The LINPACK Benchmark: past, present and future. **Concurrency and Computation: Practice and Experience**, [S.l.], v.15, n.9, p.803–820, 2003.

DRAGAN-CHIRILA, J. **Integrating a Fortran 90 Parser into an Eclipse Environment**. Urbana-Champaign, EUA, 2004.

EIPE, R. M. **Extending Eclipse to create an IDE plugin for a new language with FORTRAN as a case study**. Urbana-Champaign, EUA, 2004.

ETTINGER, R. Refactoring via Program Slicing and Sliding. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, 23., 2007, Paris, Franca. **Anais...** IEEE, 2007.

FOWLER, M.; BECK, K.; BRANT, J.; OPDYKE, W.; ROBERTS, D. **Refactoring**: improving the design of existing code. [S.l.]: Addison Wesley, 1999.

GARRIDO, A.; JOHNSON, R. Challenges of Refactoring C Programs. In: INTERNATIONAL WORKSHOP ON PRINCIPLES OF SOFTWARE EVOLUTION, 2002, Nova Iorque, EUA. **Anais...** ACM, 2002. p.6–14.

GARRIDO, A.; JOHNSON, R. Refactoring C with Conditional Compilation. In: INTERNATIONAL WORKSHOP ON PRINCIPLES OF SOFTWARE EVOLUTION, 2003, Orlando, EUA. **Anais...** IEEE Computer Society, 2003. p.323–326.

GRAF, E.; ZGRAGGEN, G.; SOMMERLAD, P. Refactoring Support for the C++ Development Tooling. In: ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS AND APPLICATIONS COMPANION, 22., 2007, Montreal, Canada. **Anais...** ACM, 2007. p.781–782.

GRISWOLD, W. G.; NOTKIN, D. Automated Assistance for Program Restructuring. **ACM Transactions on Software Engineering and Methodology**, [S.l.], v.2, n.3, p.228–269, 1993.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture**: a quantitative approach. [S.l.]: Morgan Kaufmann, 2002.

HERMANNS, M. **Parallel Programming in Fortran 95 using OpenMP**. [S.l.]: Online, 2002.

HUNT, A.; THOMAS, D. **Pragmatic Unit Testing in Java with JUnit**. [S.l.]: The Pragmatic Programmers, 2003.

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

ISO. **ISO/IEC 1539**: international standard programming language fortran. 1.ed. [S.l.: s.n.], 1997.

JOHNSON, R.; FOOTE, B.; OVERBEY, J.; XANTHOS, S. **Changing the Face of High-Performance Fortran Code**. A White Paper.

KOFFMANN, E.; FRIEDMAN, F. **FORTRAN**. 5.ed. [S.l.]: Addison Wesley, 2006.

LAKHOTIA, A.; DEPREZ, J.-C. Restructuring Programs by Tucking Statements into Functions. In: **Special Issue on Program Slicing**. [S.l.]: Elsevier, 1998. p.677–689. (Information and Software Technology, v.40).

LEE, B.; SCUSE, D. **Eclipse Project CDT (C/C++) Plugin Tutorial**. [S.l.: s.n.], 2004.

LI, H. **Refactoring Haskell Programs**. 1992. Tese de Doutorado — University of Kent, Canterbury, Reino Unido.

LI, H.; THOMPSON, S. Tool Support for Refactoring Functional Programs. In: ACM SYMPOSIUM ON PARTIAL EVALUATION AND SEMANTICS-BASED PROGRAM MANIPULATION, 2008, Sao Francisco, EUA. **Anais...** ACM, 2008.

LIU, J.; BATORY, D.; LENGAUER, C. Feature Oriented Refactoring of Legacy Applications. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 28., 2006, Shanghai, China. **Anais...** ACM, 2006.

MCCONNELL, S. **Code Complete**: a practical handbook of software construction. [S.l.]: Microsoft Press, 1993.

MENS, T.; DEMEYER, S.; BOIS, B. D.; STENTEN, H.; GORP, P. V. Refactoring: current research and future trends. **Language descriptions, Tools and Applications**, [S.l.], v.82, n.3, p.483–499, 2003.

MENS, T.; TAENTZER, G.; RUNGE, O. Analysing Refactoring Dependencies Using Graph Transformation. **Software and Systems Modeling**, [S.l.], v.6, n.3, p.269–285, 2007.

MERTON, R. K. The Matthew Effect in Science. **Science**, [S.l.], v.159, n.3810, p.56–63, 1968.

METCALF, M.; REID, J. K.; COHEN, M. **Fortran 95/2003 Explained**. 3.ed. [S.l.]: Oxford University Press, Inc., 2004.

NYHOFF, L.; LEESTMA, S. **Fortran 90 for Engineers and Scientists**. [S.l.]: Prentice-Hall, 1997.

OPDYKE, W. **Refactoring Object-Oriented Frameworks**. 1992. Tese de Doutorado — University of Illinois, Urbana-Champaign, EUA.

OVERBEY, J.; JOHNSON, R. A Survey of Software Refactoring: a foundation for the rapid development of source code transformation tools. In: FIRST INTERNATIONAL CONFERENCE ON SOFTWARE LANGUAGE ENGINEERING, 2008, Toulouse, Franca. **Anais...** [S.l.: s.n.], 2008.

OVERBEY, J.; JOHNSON, R. Generating Rewritable Abstract Syntax Trees. In: FIRST INTERNATIONAL CONFERENCE ON SOFTWARE LANGUAGE ENGINEERING, 2009, Toulouse, Franca. **Anais...** Springer-Verlag, 2009. p.114–133.

OVERBEY, J.; NEGARA, S.; JOHNSON, R. Refactoring and the Evolution of Fortran. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING: SECOND INTERNATIONAL WORKSHOP ON SOFTWARE ENGINEERING FOR COMPUTATIONAL SCIENCE AND ENGINEERING, 31., 2009, Vancouver, Canada. **Anais...** [S.l.: s.n.], 2009.

OVERBEY, J.; XANTHOS, S.; JOHNSON, R.; FOOTE, B. Refactorings for Fortran and High-Performance Computing. In: SECOND INTERNATIONAL WORKSHOP ON SOFTWARE ENGINEERING FOR HIGH PERFORMANCE COMPUTING SYSTEM APPLICATIONS, 2005, St. Louis, EUA. **Anais...** [S.l.: s.n.], 2005.

PARR, T. **The Definitive ANTLR Reference**: building domain-specific languages. [S.l.]: Pragmatic Bookshelf, 2007.

RASMUSSEN, C. E.; SQUYRES, J. M. A Case for New MPI Fortran Bindings. In: EUROPEAN PVM/MPI USERS' GROUP MEETING, 12., 2005, Sorrento, Italia. **Anais...** [S.l.: s.n.], 2005.

RIEGER, M.; ROMPAEY, B. V.; BOIS, B. D.; MEIJFROIDT, K.; OLIEVIER, P. Refactoring for Performance: an experience report. In: THIRD INTERNATIONAL ERCIM

SYMPOSIUM ON SOFTWARE EVOLUTION, 2007, Paris, Franca. **Anais...** [S.l.: s.n.], 2007.

ROBERTS, D.; BRANT, J.; JOHNSON, R. An Automated Refactoring Tool. In: INTERNATIONAL CONFERENCE ON ADVANCED SCIENCE AND TECHNOLOGY, 1996, Chicago, EUA. **Anais...** [S.l.: s.n.], 1996.

ROBERTS, D.; BRANT, J.; JOHNSON, R. E. A Refactoring Tool for Smalltalk. **Theory and Practice of Object Systems**, [S.l.], v.3, n.4, p.253–263, 1997.

SANDERS, B. A.; DEUMENS, E.; LOTRICH, V. Refactoring a Language for Parallel Computational Chemistry. In: SECOND WORKSHOP ON REFACTORING TOOLS, 2008, Nashville, EUA. **Anais...** ACM, 2008.

SEREBRENIK, A.; SCHRIJVERS, T.; DEMOEN, B. Improving Prolog Programs: refactoring for prolog. **Theory and Practice of Logic Programming**, [S.l.], v.8, n.2, p.201–215, 2008.

SUNYE, G.; POLLET, D.; TRAON, Y. L.; JEZEQUEL, J.-M. Refactoring UML Models. In: INTERNATIONAL CONFERENCE ON THE UNIFIED MODELING LANGUAGE, MODELING LANGUAGES, CONCEPTS, AND TOOLS, 4., 2001, Londres, Inglaterra. **Anais...** Springer-Verlag, 2001. p.134–148.

TOP500. 2009.

TOURWÉ, T.; MENS, T. A Survey of Software Refactoring. **IEEE Transactions on Software Engineering**, [S.l.], v.30, n.2, p.126–139, 2004.

VANTER, M. L. V. D.; POST, D. E.; ZOSEL, M. E. HPC Needs a Tool Strategy. In: SECOND INTERNATIONAL WORKSHOP ON SOFTWARE ENGINEERING FOR HIGH PERFORMANCE COMPUTING SYSTEM APPLICATIONS, 2005, St. Louis, EUA. **Anais...** ACM, 2005. p.55–59.

WATSON, G. R.; DEBARDELEBEN, N. Developing Scientific Applications Using Eclipse. **Computing in Science and Engineering**, [S.l.], v.8, n.4, p.50–61, 2006.

APÊNDICE A CLASSES E PROJETOS DE UMA REFATORAÇÃO

Neste apêndice apresenta-se um resumo de como começar o desenvolvimento de uma nova ação de refatoração , indicando em que projetos os arquivos precisam estar dispostos e qual seu formato básico. Considera-se que os fontes do Photran estejam instalados e funcionais. Para instalar e configurar os fontes do Photran consulte o apêndice A (*Getting the Photran 4.0 Sources from CVS*) no guia do desenvolvedor Photran (CHEN; OVERBEY, 2008).

Para implementar e integrar uma ação de refatoração ao Photran, faz-se necessário estender o comportamento de algumas classes (CHEN; OVERBEY, 2008). 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 que a chamada do usuário seja feita a partir do menu principal) e *IEditorActionDelegate* (permitindo que a chamada do usuário seja feita a partir 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. Essa classe é estendida de *AbstractFortranRefactoringWizard* e basicamente se responsabiliza pela construção gráfica da janela do assistente.

Ambas as classes são implementadas em um mesmo arquivo (no caso do nosso exemplo, nomeado de *TestAction.java*). O arquivo resultante dessa classe deverá ser adicionado ao projeto org.eclipse.photran.ui.vpg (conforme pode-se observar na estrutura à direita da figura A.1. A figura A.2 permite observar o esqueleto do código-fonte das duas classes que compõem o arquivo *TestAction.java*.

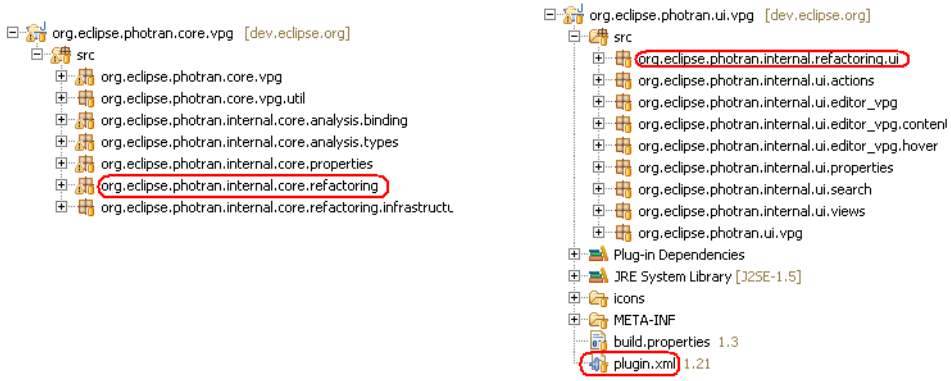

Figura A.1: Projetos onde os fontes devem ser adicionados

A ação de refatoração em si é implementada em outro arquivo (neste exemplo nomeado de *TestRefactoring.java*). O arquivo da refatoração deverá ser adicionado ao projeto org.eclipse.photran.core.vpg (conforme pode-se observar na estrutura à esquerda da figura A.1. A figura A.3 permite observar o esqueleto do código-fonte da classe que implementa a ação de refatoração.

Uma vez que os dois arquivos estão implementados, eles precisam ser referenciados pelo arquivo *plugin.xml*, que fica localizado no projeto org.eclipse.photran.ui.vpg (o arquivo *plugin.xml* a ser editado pode ser observado na figura A.1 ao lado direito). O apêndice B descreve em detalhes um exemplo de como a configuração do arquivo *plugin.xml* deve ser realizada. Após estes procedimentos, ao compilar o Photran uma nova ação de refatoração ficará disponível no menu *Refactoring*.

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

import org.eclipse.ltk.ui.refactoring.UserInputWizardPage;
import org.eclipse.photran.internal.core.refactoring.TestRefactoring;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.ModifyEvent;
import org.eclipse.swt.events.ModifyListener;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;
import org.eclipse.ui.IEditorActionDelegate;
import org.eclipse.ui.IWorkbenchWindowActionDelegate;

public class TestAction
    extends AbstractFortranRefactoringActionDelegate
    implements IWorkbenchWindowActionDelegate, IEditorActionDelegate
{
  public TestAction()
  {
    super(TestRefactoring.class, FortranTestRefactoringWizard.class);
  }

  public static class FortranTestRefactoringWizard
                      extends AbstractFortranRefactoringWizard
  {
    protected TestRefactoring testRefactoring;

    public FortranTestRefactoringWizard(TestRefactoring r)
    {
        super(r);
        this.testRefactoring = r;
    }

    @Override
    protected void doAddUserInputPages() {
        addPage(new UserInputWizardPage(testRefactoring.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("Clique OK to test refactor in the current
                Fortran file. To see what changes will be
                    made, click Preview.");
        }
        });
    }
  }
}
```

Figura A.2: Arquivo *TestAction.java*

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

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;
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.jface.text.ITextSelection;
import org.eclipse.ltk.core.refactoring.RefactoringStatus;
import org.eclipse.photran.core.IFortranAST;
import org.eclipse.photran.core.vpg.PhotranTokenRef;
import org.eclipse.photran.core.vpg.PhotranVPG;

public class TestRefactoring extends FortranRefactoring
{
    public TestRefactoring(IFile file, ITextSelection selection)  {
        super(file, selection);
    }

    @Override
    public String getName() {
        return "Teste Refatoração Fortran";
    }

    protected void doCheckInitialConditions(
        RefactoringStatus status,
        IProgressMonitor pm) throws PreconditionFailure {
    }

    protected void doCheckFinalConditions(
        RefactoringStatus status,
        IProgressMonitor pm) throws PreconditionFailure {
    }

    protected void doCreateChange(IProgressMonitor pm)
         throws CoreException, OperationCanceledException {
        try {

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

Figura A.3: Arquivo *TestRefactoring.java*

APÊNDICE B INTEGRAÇÃO DE REFATORAÇÕES AO PHOTRAN

Neste apêndice apresenta-se uma breve descrição dos pontos de extensão que podem ser descritos no arquivo *manifest* para integrar as classes de uma ação de refatoração a IDE Photran/Eclipse. Existem basicamente cinco pontos de extensão que podem ser estendidos para fornecer ao usuário as ações de refatoração (CHEN; OVERBEY, 2008):

- **org.eclipe.ui.comands**: cria uma nova categoria de comandos para representar a refatoração. Essa categoria pode ser referenciada por outros pontos de extensão no arquivo *manifest*.
- **org.eclipse.ui.actionSets**: ponto de extensão utilizado para adicionar menus e submenus à perspectiva Fortran.
- **org.eclipse.ui.actionSetPartAssociations**: permite à ação de refatoração ser habilitada ou desabilitada em função do tipo de editor de código que é utilizado (formato fixo ou livre).
- **org.eclipse.ui.popupMenus**: habilita a ação de refatoração ser acionada através de um menu pop-up.
- **org.eclipse.ui.bindings** (opcional): permite que a ação de refatoração seja acionada por meio de combinações de teclas de atalho.

A figura B.1 contém uma seção do arquivo *manifest* relativo à configuração da ação *Rename* (que é distribuída com o Photran desde a versão 4). É possível observar no código a configuração dos pontos de extensão para adicionar ao Photran a refatoração Rename.

```
<extension point="org.eclipse.ui.commands">
  <category name="Fortran Refactoring Commands"
            id="org.eclipse.photran.ui.RefactoringCategory">
  </category>
  <command  name="Rename"
    categoryId="org.eclipse.photran.ui.RefactoringCategory"
    id="org.eclipse.photran.ui.RenameRefactoringCommand">
  </command>
</extension>

<extension point="org.eclipse.ui.bindings">
  <key sequence="M3+M2+R"
     schemeId="org.eclipse.ui.defaultAcceleratorConfiguration"
     contextId="org.eclipse.photran.ui.FortranEditorContext"
     commandId="org.eclipse.photran.ui.RenameRefactoringCommand"/>
</extension>

<extension point="org.eclipse.ui.actionSets">
  <actionSet label="Fortran Refactorings"
     description="Fortran Refactorings" visible="false"
     id="org.eclipse.photran.ui.RefactoringActionSet">
     <menu label="Refac&amp;amp;tor" path="edit"
        id="org.eclipse.jdt.ui.refactoring.menu">
        <separator name="reorgGroup"/> <separator name="typeGroup"/>
        <separator name="codingGroup"/>
     </menu>
     <action label="Re&amp;amp;name (Fortran)..."
      definitionId="org.eclipse.photran.ui.RenameRefactoringCommand"
      class="org.eclipse.photran.internal.refactoring.ui.RenameAction"
      menubarPath="org.eclipse.jdt.ui.refactoring.menu/reorgGroup"
      id="org.eclipse.photran.ui.RenameRefactoringAction"/>
  </actionSet>
</extension>

<extension point="org.eclipse.ui.actionSetPartAssociations">
   <actionSetPartAssociation
       targetID="org.eclipse.photran.ui.RefactoringActionSet">
      <part id="org.eclipse.photran.ui.FreeFormFortranEditor"/>
       <!--part id="org.eclipse.photran.ui.FixedFormFortranEditor"/-->
   </actionSetPartAssociation>
</extension>

<extension point="org.eclipse.ui.popupMenus">
  <viewerContribution targetID="#FreeFormFortranEditorContextMenu"
    id="org.eclipse.photran.refactoring.refactoringEditorContribution">
    <menu id="org.eclipse.photran.ui.RefactoringMenu"
       label="Refac&amp;amp;tor" path="group.reorganize">
       <separator name="refactorings"/>
    </menu>
    <action label="&amp;amp;Rename..."
       class="org.eclipse.photran.internal.refactoring.ui.RenameAction"
       menubarPath="org.eclipse.photran.ui.RefactoringMenu/refactorings"
       id="org.eclipse.photran.ui.RenameRefactoringAction"/>
  </viewerContribution>
</extension>
```

Figura B.1: Seção do arquivo *manifest* (*plugin.xml*)

APÊNDICE C PUBLICAÇÕES GERADAS A PARTIR DO TRABALHO

Durante a realização deste trabalho, foram desenvolvidas e submetidas 2 publicações. A primeira apresenta a idéia inicial do tema bem como uma conceituação acerca de refatorações para aplicações Fortran de alto desempenho. Já a segunda apresenta a conclusão do trabalho e os resultados das avaliações realizadas.

1. Título: *Refatoração de Programas Fortran de Alto Desempenho*, resumo expandido, submetido e aceito para publicação no Fórum de Pós-graduação da *VIII Escola Regional de Alto Desempenho*, Santa Cruz do Sul, Março/2008.

2. Título: *Automação de Refatorações para Programas Fortran de Alto Desempenho*, submetido ao *X Simpósio em Sistemas Computacionais - WSCAD-SSC* (www.dcc.ufrj.br/wscad2009) e aguardando avaliação no momento da conclusão desta dissertação.