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

CATÁLOGO DE REFATORAÇÕES PARA A EVOLUÇÃO DE PROGRAMAS EM LINGUAGEM FORTRAN

DISSERTAÇÃO DE MESTRADO

Gustavo Rissetti

Santa Maria, RS, Brasil

2011

CATÁLOGO DE REFATORAÇÕES PARA A EVOLUÇÃO DE PROGRAMAS EM LINGUAGEM FORTRAN

por

Gustavo Rissetti

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: Profª Drª Andrea Schwertner Charão (UFSM)

Co-orientador: Prof. Dr. Eduardo Kessler Piveta (UFSM)

Santa Maria, RS, Brasil

2011

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

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

CATÁLOGO DE REFATORAÇÕES PARA A EVOLUÇÃO DE PROGRAMAS EM LINGUAGEM FORTRAN

elaborada por
Gustavo Rissetti

como requisito parcial para obtenção do grau de
Mestre em Computação

COMISSÃO EXAMINADORA:

Prof. Dr. Eduardo Kessler Piveta (UFSM)
(Presidente/Co-orientador)

Prof. Dr. Jairo Panetta (INPE/Petrobras)

Prof. Dr. Nicolas Maillard (UFRGS)

Santa Maria, 8 de Julho de 2011.

AGRADECIMENTOS

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

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

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

Ao professor Eduardo Piveta, sempre disposto a revisar o trabalho e a passar dicas para melhorá-lo, tornando possível concluí-lo com êxito através de suas contribuições.

“Não sabendo que era impossível, foi lá e fez.” — JEAN COCTEAU

RESUMO

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

CATÁLOGO DE REFATORAÇÕES PARA A EVOLUÇÃO DE PROGRAMAS EM LINGUAGEM FORTRAN

Autor: Gustavo Rissetti
Orientador: Profª Drª Andrea Schwertner Charão (UFSM)
Co-orientador: Prof. Dr. Eduardo Kessler Piveta (UFSM)
Local e data da defesa: Santa Maria, 8 de Julho de 2011.

A evolução é uma característica natural no desenvolvimento de software. Durante o ciclo de vida de um sistema, geralmente existe a necessidade de evolução, seja para a adição de um novo requisito, para a alteração de funcionalidades existentes, ou para a evolução da linguagem de programação usada. A linguagem Fortran (*FORmula TRANslation*), apesar de possuir mais de cinquenta anos de existência, ainda é amplamente usada em aplicações científicas. A maioria das aplicações Fortran existentes é composta de códigos legados, que usam construções obsoletas ou de uso desencorajado da linguagem, e normalmente precisam passar por uma evolução para melhorar seus atributos de qualidade. Porém, muitas vezes, esse processo é conduzido manualmente, sem a existência de regras bem definidas a serem seguidas, podendo ocorrer a introdução de anomalias nessas aplicações. A evolução de software pode ser auxiliada através de refatoração, que oferece mecanismos bem definidos a serem seguidos, ajudando a manter e melhorar a qualidade dos sistemas existentes. Refatoração é uma técnica de engenharia de software que efetua transformações em artefatos de software a fim de melhorá-los, sem comprometer suas funcionalidades. Trata-se de uma tarefa permanentemente presente no ciclo de vida de uma aplicação e está diretamente associada à requisitos não funcionais de software, tais como modularização, legibilidade e desempenho. Essa técnica é amplamente difundida para linguagens orientadas a objetos, mas é ainda pouco explorada em linguagens procedurais como Fortran. Nesse contexto, este trabalho explora a carência de refatorações para a linguagem Fortran, aliada à questão da evolução de código legado. Esse objetivo é alcançado através da proposta de um catálogo de refatorações para a evolução de programas Fortran, e da automação de algumas delas no *framework* Photran. As refatorações propostas são avaliadas e validadas em aplicações escritas em Fortran.

Palavras-chave: Fortran, evolução, refatoração.

ABSTRACT

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

A REFACTORING CATALOG FOR THE EVOLUTION OF FORTRAN PROGRAMS

Author: Gustavo Rissetti
Advisor: Profª Drª Andrea Schwertner Charão (UFSM)
Coadvisor: Prof. Dr. Eduardo Kessler Piveta (UFSM)

Evolution is a natural characteristic in software development. During the life cycle of a system, usually there is the need of evolution, mainly to add a new requirement, to change existing functionality, or to evolve the programming language used. The Fortran (FORmula TRANslation) language, despite having more than fifty years of existence, is still widely used in scientific applications. The majority of the existent Fortran applications is composed of legacy code, using obsolete or deprecated constructions of the language and, thus, need to evolve to improve their quality attributes. However, this process is often done manually, without the existence of well-defined rules to be followed, facilitating the introduction of anomalies in these applications. Software evolution can be benefited from refactoring, which provides well-defined mechanisms to be followed, helping to maintain and to improve the quality of existent systems. Refactoring is a software engineering technique that transforms software artefacts in order to improve them, without compromising their functionality. It is a permanent task in the life cycle of an application and is directly associated with the software non-functional requirements, such as modularization, legibility, and performance. This technique is widely used in object-oriented languages, but is still largely unexplored in procedural languages such as Fortran. In this context, this work explores the need for refactorings for the Fortran language, together with the issue of evolving legacy code. This goal is achieved through the proposal of a refactoring catalog to the evolution of Fortran programs, and the automation of some of them in the Photran framework. The proposed refactorings are evaluated and validated in applications written in Fortran.

Keywords: Fortran, evolution, refactoring.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

SUPIM	*Sheffield University Plasmasphere Ionosphere Model*
INPE	Instituto Nacional de Pesquisas Espaciais
ANSI	*American National Standards Institute*
IDE	*Integrated Development Environment*
UFSM	Universidade Federal de Santa Maria
SPMD	*Single-Program, Multiple-Dada*
UML	*Unified Modeling Language*
CDT	*C/C++ Development Tools*
GCC	*GNU Compiler Collection*
MPI	*Message Passing Interface*
HPF	*High Performance Fortran*
VPG	*Virtual Program Graph*
AST	*Abstract Syntax Tree*

SUMÁRIO

1 INTRODUÇÃO

1.1 Contexto e Motivação

Evolução é uma característica natural no processo de desenvolvimento de software. Durante o ciclo de vida de sistemas de software, normalmente surge a necessidade de evolução, seja para adicionar um novo requisito, para atualizar padrões da linguagem de programação usada, ou para alterar funcionalidades existentes. Porém, muitas vezes esse processo de evolução é executado manualmente, sem a existência de regras a seguir (como um conjunto de passos bem definidos, por exemplo), e isso pode facilitar a introdução de anomalias nos sistemas.

Técnicas de refatoração podem ser usadas para auxiliar no processo de evolução de software, permitindo que o programador siga instruções bem definidas para modificar trechos de código e assim atender aos novos requisitos do sistema. Refatoração é o processo de modificar um sistema de software para melhorar sua estrutura interna, sem alterar seu comportamento externo observável (resultados, funcionalidades e saídas) (OPDYKE, 1992; FOWLER et al., 1999). O conceito de refatoração não é uma prática nova, pois aplicar melhorias e reestruturações em aplicativos é um processo implícito e contínuo à tarefa de desenvolvimento (ARNOLD, 1986; TOURWÉ; MENS, 2004). O processo de refatoração vem sendo amplamente utilizado em linguagens de orientação a objetos.

As refatorações normalmente são organizadas em coleções de padrões, chamados de **catálogos de refatorações**. Cada refatoração em um catálogo é descrita por um nome, um contexto em que ela deva ser aplicada, um conjunto de passos bem definidos para a sua aplicação e um ou mais exemplos mostrando como a transformação pode ocorrer em um código (FOWLER et al., 1999).

A refatoração de código está fortemente ligada ao paradigma da orientação a objetos, mas é possível aplicar o mesmo conceito em outros paradigmas, como o para-

digma estruturado (GARRIDO; JOHNSON, 2002, 2003; OVERBEY et al., 2005; OVERBEY; NEGARA; JOHNSON, 2009), ou lógico (SEREBRENIK; SCHRIJVERS; DEMOEN, 2008), ou funcional (LI, 1992; LI; THOMPSON, 2008), ou orientado a aspectos (WLOKA; HIRSCHFELD; HÄNSEL, 2008; PIVETA, 2009; YOKOMORI et al., 2011), por exemplo.

Um exemplo de linguagem estruturada é o Fortran, que é uma linguagem de programação voltada para aplicações científicas, desenvolvida a partir da década de 1950 e que continua a ser usada atualmente. O nome tem como origem a expressão *FORmula TRANslation* ou *Translator*, sendo que essa linguagem é principalmente usada em ciência da computação e em cálculo numérico (DE, 2004; ADAMS et al., 2008). Devido à idade da linguagem, algumas construções usadas nos programas legados podem ser menos eficientes do que construções de versões mais recentes da linguagem. Além disso, algumas construções usadas em padrões anteriores pioram a legibilidade de código, dificultando a manutenção dos aplicativos.

Muitas vezes é necessário evoluir um sistema, passando-o de uma versão do Fortran para outra. Esse processo normalmente é complicado e lento quando feito manualmente. Para evoluir um código pode-se utilizar refatoração, facilitando o entendimento da evolução e oferecendo uma mecânica clara a ser seguida. As refatorações podem ser aplicadas manualmente ou de maneira semi-automática, com a ajuda de ferramentas.

Um artefato de software pode ser melhorado através de sucessivas refatorações, sem modificar sua funcionalidade. Se a linguagem sendo trabalhada possui ferramentas de auxílio que ofereçam mecanismos automatizados de refatoração, sua aplicação em larga escala e em sistemas de software de grande porte é facilitada. Refatorar com apoio de uma ferramenta, embora não se aplique a toda e qualquer refatoração, tem as vantagens de se ter a possibilidade de se desfazer uma ação e reduzir o risco de erros e inconsistências (FOWLER et al., 1999).

Na literatura é possível encontrar diversas catalogações de refatorações, independentes de linguagem de programação. A atividade de catalogar técnicas de refatoração pode ser desenvolvida de forma genérica. Normalmente, é possível (em linguagem natural) descrever os objetivos, os pré-requisitos, as limitações e a mecânica de funcionamento de uma refatoração, sem se preocupar com o paradigma ou a linguagem de programação na qual a aplicação a ser refatorada foi escrita (DE, 2004).

Este trabalho propõe um catálogo de refatorações voltadas à evolução de códigos Fortran legados. Com tais refatorações é possível transformar construções de código de uma versão do Fortran para outra, adequando-se aos novos padrões da linguagem de programação. O catálogo proposto neste trabalho é composto por um conjunto de dez refatorações que possibilitam evoluir o padrão da linguagem de programação usada nos aplicativos de software.

O *framework* Photran (DRAGAN-CHIRILA, 2004; EIPE, 2004; OVERBEY et al., 2005), um *plugin* integrado ao Eclipse, foi usado para implementar algumas das refatorações do catálogo proposto neste trabalho. O Photran atua sobre código Fortran e disponibiliza uma infraestrutura básica para a análise sintática de programas Fortran e a manipulação de suas árvores sintáticas. Seis das dez refatorações propostas neste trabalho foram implementadas no Photran, e avaliadas em aplicações escritas em Fortran.

1.2 Organização do Texto

Este trabalho está organizado da seguinte forma: o capítulo 2 apresenta uma fundamentação sobre refatoração de programas Fortran. Nesse capítulo são abordados conceitos e objetivos relacionados a refatoração, assim como aspectos ligados à sua utilização na linguagem Fortran. Ainda no capítulo 2, são apresentadas as características da linguagem Fortran e os recursos adicionados em cada versão da linguagem (Fortran 77, 90, 95, 2003 e 2008). O capítulo 2 também apresenta alguns trabalhos relacionados com o tema abordado, tratando sobre a evolução de software e a refatoração de software.

O capítulo 3 apresenta o catálogo de refatorações para a evolução de programas Fortran proposto neste trabalho, contendo motivação, mecânica e exemplos de uso para cada uma das refatorações.

O capítulo 4 apresenta a ferramenta Photran, com seus recursos, características e sua infraestrutura para a automação de refatorações, além de detalhes da automação de seis refatorações propostas no catálogo.

No capítulo 5, as refatorações implementadas no Photran são avaliadas em aplicações escritas em Fortran. Por fim, o capítulo 6, apresenta as considerações finais do trabalho, e discute algumas ideias para sua continuidade.

2 FUNDAMENTAÇÃO

2.1 Refatoração

Refatoração é o processo de melhorar a estrutura interna de um sistema de software sem alterar seu comportamento externo (OPDYKE, 1992; FOWLER et al., 1999; TOURWÉ; MENS, 2004). Esse processo normalmente envolve a remoção de código duplicado, simplificação de lógica condicional e melhora de legibilidade de código (KERIEVSKY, 2004). Em particular, os resultados produzidos pelo aplicativo refatorado devem ser idênticos aos resultados obtidos antes de sua refatoração.

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

As refatorações são organizadas em coleções de padrões, chamados de **catálogos de refatorações**, conforme já mencionado na introdução. Cada refatoração em um catálogo é descrita por um nome, um contexto em que ela deva ser aplicada, um conjunto de passos bem definidos para a sua aplicação e um ou mais exemplos mostrando como a transformação pode ocorrer (FOWLER et al., 1999). Muito do que se faz atualmente é agrupar, catalogar e documentar técnicas de refatoração de forma que elas possam ser aplicadas de forma sistemática, melhoradas e compartilhadas.

A Figura 2.1 mostra um modelo de representação de uma refatoração em um catálogo, traduzido de *Refactoring: Improving the Design of Existing Code* (FOWLER et al., 1999).

Substituir o Algoritmo

Você quer substituir um algoritmo por um mais claro.
Substitua o corpo do método pelo novo algoritmo.

```
String pessoaEncontrada (String[] pessoas){
   for(int i = 0; i < pessoas.length; i++){
      if(pessoas[i].equals("Don")){
         return "Don";
      }
      if(pessoas[i].equals("John")){
         return "John";
      }
      if(pessoas[i].equals("Kent")){
         return "Kent";
      }
   }
   return "";
}
```

```
String pessoaEncontrada (String[] pessoas){
   List candidatos = Arrays.asList(new String[]{"Don","John","Kent"});
   for(int i = 0; i < pessoas.length; i++){
      if(candidatos.contains(pessoas[i]))
         return pessoas[i];
   return "";
}
```

Motivação

"...". Às vezes, quando você quer alterar o algoritmo para fazer alguma coisa ligeiramente diferente, é mais fácil substituí-lo antes por algo mais fácil para a alteração que você quer realizar.

Quando você tem quer dar esse passo, assegure-se de que decompôs o método o máximo que puder. Substituir um algoritmo grande e complexo é muito difícil; apenas tornando-o simples você pode tornar a substituição manejável.

Mecânica

- Prepare seu algoritmo alternativo. Faça com que ele compile.
- Execute o novo algoritmo em seus testes. Se os resultados forem os mesmos, você terminou.
- Caso os resultados não sejam os mesmos, compare o novo algoritmo com o antigo nos testes de depuração.
    - Execute cada conjunto de testes com os algoritmos novo e antigo e observe os resultados de ambos. Isso lhe ajudará a ver quais conjuntos de testes estão causando problemas e como.

Figura 2.1: Representação de refatoração em um catálogo (FOWLER et al., 1999)

O catálogo de refatorações proposto neste trabalho é baseado no modelo de catalogação usado em *Refactoring: Improving the Design of Existing Code* (FOWLER et al., 1999) (Figura 2.1). Para cada refatoração do catálogo proposto, é disponibilizada uma contextualização para o seu uso, sua motivação, sua mecânica de funcionamento e exemplos de uso, como pode ser observado no capítulo 3.

Algumas das situações que podem ser citadas como oportunidades de refatoração são: métodos ou classes muito extensos, métodos com excessivo número de parâmetros e falta de clareza ou legibilidade. Outras situações em que podem ser usadas refatorações são: adição de um novo recurso, correção de um erro e revisão de código-fonte (FOWLER et al., 1999).

Refatorações podem ser aplicadas manualmente ou com ajuda de ferramentas de apoio automatizadas. Uma refatoração deve preservar a semântica de equivalência de operações e referências de código. Assim, um programa deve produzir a mesma saída, dada a mesma entrada, antes e depois da aplicação do conjunto de refatorações (CORNELIO, 2004). Para refatorar com segurança, deve-se testar manualmente se as mudanças inseridas no código através da refatoração não causaram inconsistências ou anomalias no sistema de software. Os testes também podem ser executados de forma automatizada para a verificação da corretude do sistema, caso uma suíte de testes automatizados esteja disponível para o uso. Refatorar em pequenos passos ajuda a prevenir a introdução de defeitos e anomalias nos aplicativos de software. É melhor aplicar um conjunto de pequenas refatorações, testando o código após cada mudança, do que aplicar uma grande alteração em um único passo e depois executar os testes de corretude do código (FOWLER et al., 1999).

A evolução de sistemas legados representa uma motivação para o emprego de técnicas de refatoração. À medida que novas técnicas, práticas e ferramentas são incorporadas aos processos de desenvolvimento de software, é desejável que o código legado evolua para contemplar os novos recursos. A evolução significa em muitos casos substituir construções antigas (consideradas obsoletas ou de uso desencorajado) por novas construções que atendam a novos padrões de uma determinada linguagem de programação (geralmente com maior poder de expressão). Um exemplo disso seria substituir um trecho do código com desvios rotulados, como *GO TO*, por comandos de decisão ou por laços de repetição.

Embora o termo refatoração tenha origem na orientação a objetos, seu conceito pode ser usado em outros paradigmas, tais como programação estruturada (GARRIDO; JOHN-

SON, 2002, 2003; OVERBEY et al., 2005; OVERBEY; NEGARA; JOHNSON, 2009), programação funcional (LI, 1992; LI; THOMPSON, 2008), programação orientada a aspectos (WLOKA; HIRSCHFELD; HÄNSEL, 2008; PIVETA, 2009; YOKOMORI et al., 2011) e programação lógica (SEREBRENIK; SCHRIJVERS; DEMOEN, 2008), contendo algumas limitações. Em linguagens estruturadas, os fluxos de controle e de dados são fortemente interligados, dificultando assim a manipulação do código para ser refatorado (GARRIDO; JOHNSON, 2002). Em geral, reestruturações em códigos não orientados a objeto são limitadas a nível de sub-programa, bloco de código ou entidades (MENS et al., 2003). No entanto, existe um considerável conjunto de transformações que pode ser aplicado independentemente do paradigma de programação utilizado, atuando principalmente em sub-programas, blocos condicionais, blocos de repetição e nomes de entidades (TICHELAAR et al., 2000; ROYCHOUDHURY, 2004; DI PENTA et al., 2005; MARTICORENA, 2005).

Assim como para as linguagens orientadas a objetos, também existem ferramentas de auxílio ao processo de refatoração para linguagens estruturadas, como Fortran, por exemplo. Uma ferramenta disponível para essa linguagem é o Photran, um *plugin* do Eclipse que atua sobre código Fortran (descrito na seção 4.1), contendo refatorações automatizadas para essa linguagem e que permite a criação e a inserção de novas refatorações. Essa ferramenta é usada para a automação de algumas das refatorações propostas no catálogo de refatorações para a evolução de programas Fortran, descritas no capítulo 3.

2.2 Evolução da Linguagem Fortran

Fortran foi a linguagem de programação pioneira em apresentar uma semântica de alto nível e também a utilização de um compilador (DE, 2004), tendo sua primeira versão apresentada em 1957 (NYHOFF; LEESTMA, 1997).

A linguagem Fortran se tornou popular nos anos 60, quando diferentes fabricantes de computadores começaram a produzir suas próprias versões da linguagem, e isso levou ao crescimento de dialetos divergentes de Fortran. Foi reconhecido que tais divergências não eram do interesse nem dos usuários de computadores, nem dos fabricantes. Assim, Fortran 66 veio a ser a primeira linguagem oficialmente padronizada, no ano de 1972, tornando-se um padrão de programação largamente utilizado pelos fabricantes e usuários de computadores (ADAMS et al., 2008).

A principal aplicação de Fortran, desde o seu surgimento, foi a computação científica (DE, 2004; ADAMS et al., 2008). Mesmo tendo mais de cinquenta anos de existência, essa linguagem de programação continua sendo amplamente utilizada em aplicações científicas de alto desempenho. Uma das principais vantagens da linguagem, considerando esse tipo de aplicação, é a existência de operações para tratamento de dados multidimensionais, que normalmente não são encontradas de forma nativa em outras linguagens de programação (KOFFMAN; FRIEDMAN, 1996).

A linguagem Fortran ainda mantém a compatibilidade com versões anteriores. Uma característica de suas primeiras versões eram as regras de alinhamento das linhas do código-fonte, uma herança da era dos cartões perfurados (Figura 2.2). As regras de alinhamento foram usadas obrigatoriamente até o padrão Fortran 77, inclusive.

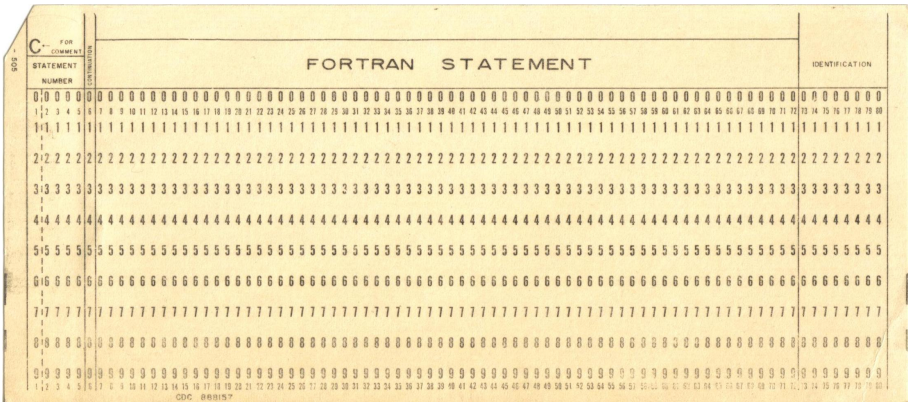


Figura 2.2: Cartão para programação em Fortran (JONAS, 2011)

As mesmas regras continuaram em vigor, mesmo depois de os cartões perfurados terem caído em desuso e os programas em Fortran 77 passarem a ser escritos diretamente em editores de texto. Como a linguagem ainda mantém compatibilidade com versões anteriores, é possível encontrar em uma mesma aplicação, 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.

2.2.1 Diferenças dos Padrões da Linguagem

Esta seção aborda as diferenças dos padrões da linguagem Fortran, considerando as versões Fortran 77, 90, 95, 2003, e 2008. As versões anteriores da linguagem, tais como Fortran I, II, III, IV, e Fortran 66 (BACKUS, 1978; GREENFIELD, 1982), não serão

abordadas neste trabalho, pois grande parte dos códigos legados existentes usam o padrão do Fortran 77, que foi a versão da linguagem mais usada desde seu surgimento (desde a década de 1970 até o início da década de 1990).

Nas sub-seções seguintes são descritos os principais recursos adicionados nos padrões Fortran 77, 90, 95, 2003, e 2008, respectivamente.

2.2.1.1 Recursos adicionados no Fortran 77

A definição do padrão Fortran 66 foi atualizada no final dos anos 70, pois ainda existiam algumas questões de definição da linguagem que não seguiam um padrão preciso. O novo padrão da linguagem ficou comumente conhecido como Fortran 77 e esta foi a versão da linguagem de uso mais difundido até o início da década de 1990, quando foi lançado um novo padrão do Fortran (RAMSDEN; LIN, 1995). Os recursos mais significativos presentes no Fortran 77 são (ANSI, 1978):

- Blocos *IF-END IF*, com as cláusulas opcionais *ELSE* e *ELSE IF*;
- Extensões para os laços *DO*, incluindo parâmetros e incrementos negativos;
- Comandos *OPEN, CLOSE* e *INQUIRE* (comandos de entrada/saída);
- Acesso direto a arquivos de entrada/saída;
- Comando *IMPLICIT*;
- Tipo de dado *CHARACTER* para processar dados baseados em caracteres;
- Comando *PARAMETER* para especificar constantes;
- Comando *SAVE* para variáveis locais persistentes;
- Nomes genéricos para funções intrínsecas;
- Funções intrínsecas para comparações léxicas de strings (*LGE, LGT, LLE, LLT*).

As palavras-chave definidas no Fortran 77 são: *assign, backspace, block data, call, close, common, continue, data, dimension, do, else, else if, end, endfile, endif, entry, equivalence, external, format, function, goto, if, implicit, inquire, intrinsic, open, parameter, pause, print, program, read, return, rewind, rewrite, save, stop, subroutine, then* e *write*.

2.2.1.2 Recursos adicionados no Fortran 90

O Fortran 90 surgiu como um aprimoramento do Fortran 77, tendo novos recursos como: a habilidade de realizar operações matriciais nativas (nas quais a matriz pode ser tratada como uma "variável"), a criação de módulos que podem conter dados e sub-rotinas, a habilidade de definir tipos derivados de dados, dentre outros.

Uma das maiores diferenças dessa versão da linguagem é o formato livre do código-fonte. A partir do Fortran 90 não era mais obrigatório seguir as regras de alinhamento de caracteres usadas nos cartões perfurados. Existem ferramentas gratuitas que convertem (refatoram) códigos de formato fixo para formato livre (MILLER, 2000). Os recursos mais significativos presentes no Fortran 90 são (ANSI, 1991):

- Operações matriciais nativas;
- Ponteiros;
- Recursos avançados para computação numérica com o uso de funções intrínsecas;
- Parametrização dos tipos intrínsecos;
- Tipos derivados de dados, definidos pelo programador;
- Módulos, para agrupar dados e sub-rotinas;
- Estrutura de controle *SELECT CASE*;
- Nova forma para o laço *DO*, com *CYCLE* e *EXIT*;
- Alocação dinâmica de memória;
- Sub-rotinas recursivas;
- Melhoramento nos recursos de entrada/saída;
- Novas sub-rotinas intrínsecas.

O Fortran 90 contém as seguintes palavras-chave (além das contidas no Fortran 77): *allocate, allocatable, case, contains, cycle, deallocate, elsewhere, exit, include, interface, intent, module, namelist, nullify, only, operator, optional, pointer, private, procedure, public, result, recursive, select, sequence, target, use, while* e *where*.

2.2.1.3 Recursos adicionados no Fortran 95

O Fortran 95 surgiu após a criação do HPF (*High Performance Fortran* - Fortran de Alto Desempenho) (KOELBEL; ZOSEL, 1993), que consiste em extensões para a linguagem Fortran 90, permitindo a construção de código portável quando computadores paralelos são usados para resolver problemas com dados que podem ser representados por matrizes regulares. As principais extensões do HPF são na forma de diretivas, que são vistas pelo Fortran 90 como comentários, mas são interpretados pelo compilador HPF.

Além de ser elaborado para fazer correções de alguns problemas do Fortran 90, o padrão Fortran 95 teve a incorporação dos recursos do HPF, sendo que esses constituem as novidades mais importantes desse padrão. Os recursos mais significativos presentes no Fortran 95 são (ANSI, 1997):

- Construção *FORALL* e *WHERE* para a vetorização;
- Procedimentos *PURE* e *ELEMENTAL* definidos pelo usuário;
- Inicialização padrão de componentes de tipos derivados e ponteiros;

O Fortran 95 contém as seguintes palavras-chave (além das contidas no Fortran 90): *elemental, forall* e *pure*.

2.2.1.4 Recursos adicionados no Fortran 2003

No Fortran 2003 foi introduzido um direcionamento ainda maior para programação orientada a objetos, oferecendo uma maneira efetiva de separar a programação de um código grande e complexo em tarefas independentes, e permitindo a construção de novos códigos baseados em rotinas já existentes, além de uma capacidade expandida de interface com a linguagem C, necessária para que os programadores em Fortran possam acessar rotinas escritas em C e vice-versa. Os recursos mais significativos presentes no Fortran 2003 são (ANSI, 2004; REID, 2007):

- Aprimoramentos dos tipos derivados;
- Suporte para programação orientada a objetos;
- Aperfeiçoamentos na manipulação de dados;
- Aperfeiçoamentos em operações de entrada/saída de dados;

- Ponteiros de sub-rotinas;
- Interoperabilidade com a linguagem de programação C;
- Integração aperfeiçoada com o sistema operacional hospedeiro.

O Fortran 2003 contém as seguintes palavras-chave (além das contidas no Fortran 95): *abstract, associate, asynchronous, bind, class, deferred, enum, enumerator, extends, final, flush, generic, import, non_overridable, nopass, pass, protected, value, volatile* e *wait*.

2.2.1.5 Recursos adicionados no Fortran 2008

O Fortran 2008 surgiu como um aprimoramento que incorpora correções ao padrão anterior, e incorpora um conjunto de novas funcionalidades à linguagem. Os recursos mais significativos presentes no Fortran 2008 são (ANSI, 2010; REID, 2008):

- Submódulos: facilidades estruturais adicionais para módulos;
- *Coarray* Fortran: um modelo de execução paralelo para Fortran;
- Construção *DO CONCURRENT*: paraleliza laços sem dependências de dados;
- Atributo *CONTIGUOUS*: especifica restrições no modelo de armazenamento;
- Construção *BLOCK*: pode conter declarações em um escopo;

O Fortran 2008 contém as seguintes palavras-chave (além das contidas no Fortran 2003): *block, codimension, do concurrent, contiguous, critical, error stop, submodule, sync all, sync images, sync memory, lock* e *unlock*.

2.3 Trabalhos Relacionados

Nesta seção são mostrados alguns trabalhos que se relacionam com o tema abordado neste trabalho, a **evolução** de código Fortran através de refatorações. Eles foram divididos em duas sub-seções, que apresentam trabalhos relacionados à evolução de software e à refatoração de software, respectivamente.

2.3.1 Evolução de Software

Mudanças são características essenciais no desenvolvimento de software (GODFREY; GERMAN, 2008). Existem algumas métricas e leis da evolução de software que podem ser seguidas para se obter um melhor resultado no processo de evolução (LEHMAN et al., 1997). Um resumo sobre as oito leis de evolução de software descritas no trabalho de Lehman et al. pode ser observado na Tabela 2.1.

Lei	Descrição
Mudança contínua	Um sistema de software deve ser continuamente adaptado, caso contrário se torna progressivamente menos satisfatório.
Complexidade Crescente	À medida que um aplicativo de software é alterado, sua complexidade cresce, a menos que um trabalho seja feito para mantê-la ou diminuí-la.
Auto-regulação	O processo de evolução de software é auto-regulado próximo à distribuição normal com relação às medidas dos atributos de produtos e processos.
Conservação da Estabilidade Organizacional	A não ser que mecanismos de retro-alimentação tenham sido ajustados de maneira apropriada, a taxa média de atividade global efetiva num aplicativo de software em evolução tende a se manter constante durante o tempo de vida do produto.
Conservação da Familiaridade	De maneira geral, a taxa de crescimento incremental e a taxa de crescimento a longo prazo tendem a declinar.
Crescimento Contínuo	O conteúdo funcional de um sistema de software deve ser continuamente aumentado durante seu tempo de vida para manter a satisfação do usuário.
Qualidade Decrescente	A qualidade de software será entendida como declinante à menos que o aplicativo de software seja rigorosamente adaptado às mudanças no ambiente operacional.
Sistema de Retro-alimentação	Processos de evolução de software são sistemas de retro-alimentação em múltiplos níveis, em múltiplos laços e envolvendo múltiplos agentes.

Tabela 2.1: Resumo das leis de evolução de software (LEHMAN et al., 1997)

Normalmente, é necessário que aplicativos de software escritos em Fortran passem por um processo de evolução, uma vez que a linguagem possui mais de cinquenta anos de existência, e foi elaborada, inicialmente, com recursos computacionais bastante limitados, se comparados com os recursos disponíveis hoje. Devido a esse fato, a maioria dos aplicativos legados possui diversas regiões nos códigos onde construções obsoletas ainda são usadas, e regiões onde construções atuais poderiam oferecer maior eficiência no desempenho da aplicação. A refatoração desses aplicativos oferece uma oportunidade para melhorar a qualidade de seus códigos, possibilitando a utilização de recursos mais atuais da linguagem de programação Fortran.

No contexto da evolução de software, é possível encontrar trabalhos que abordam a

evolução de software antes de a refatoração ocorrer, ou seja, é feita uma análise (mineração) do aplicativo de software para prever que refatorações vão ocorrer durante sua evolução, através do uso de algoritmos que analisam o código e criam visualizações das possíveis mudanças, além de verificar em que sequência as refatorações devem ser aplicadas no sistema de software (MELTON; TEMPERO, 2006; TOURWÉ; MENS, 2003; RATZINGER et al., 2007; PIVETA et al., 2008, 2009).

A análise da evolução pode ser uma boa fonte de informação para diversas atividades do ciclo de vida de um sistema, como engenharia reversa, manutenção, e visualização de futuras evoluções de um sistema. Muitas pesquisas sobre evolução de software tratam apenas de registros de evolução encontrados em repositórios de versionamento de software, como CVS, SVN, etc. Mas existem também trabalhos que tratam da evolução do software baseado em qualquer mudança semântica ou sintática que o código sofre, como refatorações (ROBBES; LANZA; LUNGU, 2007). Esse é o caso de evolução de código de que trata este trabalho, no qual, através de refatorações de código-fonte, é possível fazer a evolução do padrão de programação usado no código das aplicações.

A questão da evolução de linguagens é extensamente estudada. O trabalho de Pizka e Juergens (PIZKA; JUERGENS, 2007) explica sobre a necessidade de automatizar a evolução de linguagens de alto nível, discutindo as dificuldades que normalmente são encontradas no processo de automação de evoluções, e propondo alguns conceitos em relação ao assunto, assim como um protótipo de uma ferramenta que suporta a evolução incremental de uma linguagem. Esse tipo de ferramenta pode ajudar a reduzir o custo de manutenção de aplicativos de software, sendo esse também um dos objetivos das refatorações propostas neste trabalho.

2.3.2 Refatoração de Software

O uso de refatoração não está restrito apenas ao código fonte de aplicações, podendo também ser aplicado a outros artefatos de software, como no projeto da aplicação, modelos de análise, bancos de dados, dentre outros. Alguns trabalhos abordam o uso de refatorações em diagramas UML (ASTELS, 2002) assim como a automação de refatorações integradas a editores UML (BOGER; STURM; FRAGEMANN, 2003). Existem também, na área de banco de dados, trabalhos que discutem a refatoração no contexto da evolução de esquemas relacionais (BOEHM et al., 2007). Um dos grandes desafios é

definir mecanismos para manter a consistência entre diferentes artefatos (implementação e modelo) de software com o uso de refatoração.

Refatorações podem ser usadas em sequência para chegar a um resultado desejado (PIVETA et al., 2008). Um exemplo disso pode ser observado no trabalho de Xu e Butler (XU; BUTLER, 2006), onde é apresentada uma metodologia de refatorações em sequência para um framework existente. Também existem trabalhos que fazem um estudo sobre o monitoramento da evolução de software quando se usam múltiplos tipos de mudanças no código-fonte (ALI; MAQBOOL, 2009). Em tais trabalhos são mostradas mudanças de software onde recursos são adicionados, removidos e modificados.

Há trabalhos que mostram como uma evolução de software pode ser feita considerando o seu comportamento histórico (ZHAO et al., 2009). Nesse caso a evolução pode ser feita de diversas maneiras, entre elas, com o uso de refatorações de código-fonte, como as refatorações propostas no catálogo de refatorações de que trata este trabalho.

Existem também outros trabalhos que descrevem catálogos para refatorações de Fortran, com refatorações para diferentes objetivos no código-fonte (MÉNDEZ et al., 2010a). No trabalho de Méndez et al. é mostrado um catálogo de refatorações para Fortran classificadas de acordo com os atributos que alteram o código fonte, sendo que a maioria delas está presente de forma automatizada na ferramenta Photran. O trabalho de Méndez et al. divide as refatorações apresentadas em dois grandes grupos: refatorações para melhorar a manutenibilidade dos sistemas, e refatorações para melhorar o desempenho dos sistemas. A diferença em relação a este trabalho está no fato de que as refatorações propostas neste trabalho tratam apenas sobre a evolução de códigos Fortran, especificando o que se pode evoluir de um padrão para o outro da linguagem, e neste trabalho as refatorações são divididas em grupos para a evolução de determinadas versões da linguagem Fortran.

Na pesquisa de refatoração para Fortran também pode-se encontrar trabalhos que fazem reestruturação de código sequencial para código paralelo/distribuído (EVERAARS; ARBAB; BURGER, 1996). Nesse caso, com a refatoração de software pode ser possível aproveitar melhor o desempenho oferecido pelos sistemas paralelo/distribuídos, sem ter de reescrever todo o código legado para conseguir a transformação.

A questão da evolução de programas Fortran através de refatorações também é descrita nos trabalhos de Overbey, Negara e Johnson (OVERBEY; NEGARA; JOHNSON, 2009) e Méndez et al. (MÉNDEZ et al., 2010b). Tais trabalhos tratam sobre o uso de ferramentas

de refatoração automatizadas para eliminar problemas em códigos legados, adequando-os a novos padrões de programação. Apesar desses trabalhos tratarem sobre a evolução de códigos Fortran, não é apresentado um catálogo específico com refatorações voltadas à evolução de códigos Fortran, como ocorre neste trabalho.

3 CATÁLOGO DE REFATORAÇÕES

O objetivo principal deste trabalho consiste em definir um catálogo de refatorações para a evolução de programas Fortran. O catálogo proposto contém dez refatorações, com objetivos de usar melhores construções da linguagem em determinadas circunstâncias, evoluindo o padrão da linguagem usada nas aplicações. As refatorações propostas operam nos seguintes recursos de cada padrão da linguagem:

- **Evolução de Fortran 77 para Fortran 90:**
  - Operações com matrizes;
  - Uso de módulos;
  - Uso de tipos derivados de dados;
  - Uso da estrutura de controle *SELECT CASE*.
- **Evolução de Fortran 90 para Fortran 95:**
  - Uso da construção *FORALL*.
- **Evolução de Fortran 95 para Fortran 2003:**
  - Suporte para programação orientada a objetos.
- **Evolução de Fortran 2003 para Fortran 2008:**
  - Uso de laços *DO* concorrentes;
  - Uso de *Coarrays*.

A Tabela 3.1 mostra um resumo sobre as refatorações propostas neste trabalho, descrevendo os objetivos de cada refatoração.

Tabela 3.1: Resumo das refatorações propostas

Refatoração	Objetivo
Converter laços DO em Operações Matriciais Nativas	Converte laços *DO* que efetuam operações matriciais (como somas, multiplicação, inicialização, etc) em operações matriciais nativas do Fortran 90, podendo melhorar a legibilidade de código e o desempenho de execução da aplicação.
Mover Subprograma para um Módulo	Move um subprograma para um módulo existente, melhorando a organização estrutural do código e possibilitando que o subprograma seja usado em qualquer escopo da aplicação.
Extrair Subprograma para Módulo	Extrai um subprograma para um novo módulo definido pelo usuário, melhorando a organização estrutural do código e possibilitando que o subprograma seja usado em qualquer escopo da aplicação.
Mover Variáveis para um Tipo Derivado de Dados	Move variáveis para um tipo derivado de dado existente, facilitando a visualização e a manipulação de tais variáveis, servindo também como um auxilio para a refatoração *Extrair Variáveis para Tipo Derivado de Dados*.
Extrair Variáveis para Tipo Derivado de Dados	Extrai um conjunto de variáveis semelhantes, usadas em um artefato de software, para um tipo derivado de dados, facilitando a visualização e a manipulação de tais variáveis.
Converter IF-THEN-ELSE Aninhados em SELECT CASE	Converte comandos *IF-THEN-ELSE* aninhados que avaliam a igualdade de valores de uma única variável em uma estrutura de controle *SELECT CASE*, que é específica para esse tipo de operação, melhorando a legibilidade de código.
Converter laços DO em construções FORALL	Converte laços *DO* que operam sobre vetores e matrizes em construções do tipo *FORALL*, permitindo executar os laços paralelamente em uma arquitetura multiprocessada, e aumentar o desempenho de execução da aplicação.
Converter Programa Procedural em Programa Orientado a Objetos	Converte o paradigma de programação usado no código-fonte, oferecendo melhores maneiras de separar o código em tarefas independentes, e permitindo construir códigos baseados em rotinas já existentes.
Converter laços DO em laços DO CONCURRENT	Converte laços *DO* convencionais em laços *DO CONCURRENT*, permitindo a execução paralela do laço quando não existem dependências de dados em seu interior, aumentando o desempenho de execução da aplicação.
Converter MPI em Coarrays	Usa o recurso de *coarrays* no lugar de bibliotecas externas que promovem paralelismo, como a MPI (*Message Passing Interface*), por exemplo. Com *coarrays* o código fica com um bom desempenho e com melhor legibilidade.

As refatorações descritas neste trabalho não são as únicas refatorações existentes para a evolução de programas em linguagem Fortran. Na literatura existem outras refatorações catalogadas que tratam sobre a evolução de construções Fortran legadas. A Tabela 3.2 mostra algumas das refatorações existentes que podem ser usadas para a evolução de programas Fortran (MÉNDEZ et al., 2010a).

Refatoração	Objetivo
Remove Real Type Iteration Index	Converte parâmetros ou variáveis de controle de laços *DO* declaradas como variáveis reais em variáveis inteiras.
Introduce Implicit None	Adiciona o comando *Implicit None* em um arquivo e converte todas as declarações que eram implícitas em declarações explícitas.
Introduce Intent In/Out	Adiciona o atributo de intenção de uso para cada argumento de funções ou sub-rotinas.
Replace Obsolete Operators	Substitui os estilos antigos de operadores de comparação (como *.EQ.*) pela nova versão (como ==).
Change Fixed Form To Free Form	Converte arquivos Fortran de formato fixo para formato livre.
Transform Character to Character(Len =) declaration*	Converte as declarações *Character** pela forma *Character(Len =)* para a declaração de *strings*.
Remove Computed Go To statement	Converte uma construção do tipo *Computed Go To* por uma construção *SELECT CASE* contendo *Go Tos*, ou se possível, remove todos os *Go Tos* e usa apenas a construção *SELECT CASE*.
Remove Arithmetic If Statement	Substitui um *IF* aritmético antigo, sendo análogo para remover construções *Computed Go To*.
Remove Assigned Go Tos	Remove construções *assigned Go To*.
Replace Old Styles DO loops	Substitui estilos antigos de laços *DO CONTINUE* por laços *DO END DO*.
Replace Shared Do Loop Termination	Substitui todas as terminações compartilhadas de laços *DO* por terminações *END DO* em cada laço.
Transform To While Sentence	Remove o *WHILE* simulado, feito por comandos *IF* e *Go Tos*.
Move Common Block to Module	Remove todas as declarações de um *COMMON BLOCK* em particular e move suas declarações para um módulo, introduzindo os comandos *USE* onde for necessário.
Move Saved Variables To Common Block	Cria um *COMMON BLOCK* para todas as variáveis com atributo *SAVE* de um subprograma.
Convert Data To Parameter	Converte declarações do tipo *DATA* em declarações do tipo *PARAMETER*, deixando claro quais variáveis são constantes em um código-fonte.

Tabela 3.2: Outras refatorações para evoluir códigos Fortran

As seções seguintes dividem as refatorações propostas quanto à evolução dos padrões da linguagem, e as respectivas sub-seções detalham cada refatoração com sua motivação, sua mecânica de funcionamento e exemplos de uso.

3.1 Refatorações para a Evolução de Fortran 77 para Fortran 90

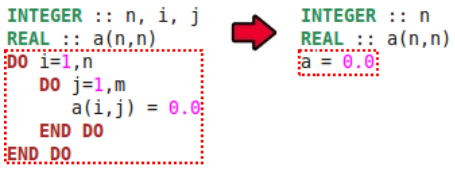
3.1.1 Converter laços *DO* em Operações Matriciais Nativas

Você tem um código com laços *DO* realizando operações matriciais.

♢ ♢ ♢

Converta os laços DO em operações matriciais nativas do Fortran 90, melhorando a qualidade do código e facilitando o seu entendimento.

♢ ♢ ♢



3.1.1.1 Motivação

Um recurso introduzido no Fortran 90 é a capacidade estendida de processamento de matrizes, ou seja, a capacidade de tratar matrizes como se fossem variáveis comuns, efetuando operações diretamente sobre toda a matriz, sem ser necessário percorrê-la com laços *DO* e efetuar operações individualmente em cada um de seus elementos.

Para manipular uma matriz em Fortran 77, cada um de seus elementos deveria ser envolvido na expressão separadamente, em um processo que, frequentemente, usava laços *DO* aninhados. Um recurso introduzido no Fortran 90 é a habilidade de realizar operações envolvendo toda a matriz, tratando a matriz como um objeto único, facilitando a construção, a leitura e a interpretação do código. As normas do padrão Fortran 90 supõem que compiladores usados em sistemas paralelos devem se encarregar de distribuir automaticamente os processos numéricos envolvidos nas expressões com matrizes de forma equilibrada entre os diversos processadores que compõe a arquitetura. Com o conceito de operações sobre matrizes inteiras, a tarefa de implantar a paralelização dessas operações fica a cargo do compilador, e não do programador.

Para que as operações envolvendo matrizes inteiras sejam possíveis, é necessário que as matrizes envolvidas nas operações sejam conformáveis, ou seja, todas elas devem ter a mesma forma (dimensões). Todos os operadores numéricos definidos para operações entre escalares também são definidos para operações entre matrizes.

3.1.1.2 Mecânica

1. Selecionar um laço *DO* que opera sobre uma matriz.
2. Identificar o tipo de operação realizada sobre seus elementos.
3. Escrever uma expressão nativa, fora do laço, equivalente às operações realizadas em seu interior, tais como inicialização, adição, subtração, multiplicação ou divisão.
4. Excluir as referências da matriz identificada do interior do laço.
    - Se o corpo do laço ficou vazio, removê-lo do código-fonte.
    - Excluir as variáveis usadas como índices do laço (caso não sejam usadas em outros locais do código-fonte).
5. Compilar e testar.

3.1.1.3 Exemplos

Um exemplo de operação matricial é a cópia do valor de uma matriz para outra. No Fortran 77 isso é feito com laços de repetição, como pode ser observado na Figura 3.1.

```
INTEGER, PARAMETER :: n = 10
REAL :: a(n,n), r(n,n)
INTEGER :: i, j
DO i=1,n
   DO j=1,n
      r(i,j) = a(i,j)
   END DO
END DO
```

Figura 3.1: Cópia matricial em Fortran 77

No Fortran 90, a mesma operação pode ser feita usando apenas uma linha de código, tratando as matrizes como variáveis comuns, como pode ser observado na Figura 3.2.

```
INTEGER, PARAMETER :: n = 10
REAL :: a(n,n), r(n,n)
r = a
```

Figura 3.2: Cópia matricial em Fortran 90

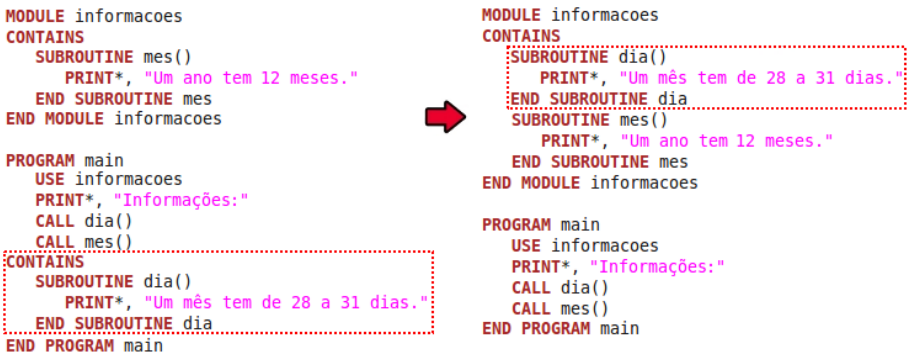
3.1.2 Mover Subprograma para um Módulo

Você dispõe de um subprograma e deseja disponibilizá-lo em um módulo existente.

♢ ♢ ♢

Mova o subprograma para o módulo desejado, e use-o onde for necessário.

♢ ♢ ♢



3.1.2.1 Motivação

Com a aplicação dessa refatoração, é possível mover subprogramas para módulos existentes, servindo também como uma refatoração de apoio para a refatoração *Extrair Subprograma para Módulo* (descrita na seção 3.1.3).

Essa refatoração re-organiza o código-fonte, movendo sub-rotinas e funções específicas para módulos que agrupam tipos distintos de subprogramas, como uma biblioteca, que pode ser usada no código-fonte da aplicação. Caso existam variáveis declaradas como *PARAMETER* no código-fonte, e elas sejam usadas apenas no interior do subprograma a ser movido, elas também são movidas para o interior do módulo de destino.

3.1.2.2 Mecânica

1. Selecionar uma sub-rotina ou função no código-fonte.
2. Fornecer o nome do módulo de destino para o código selecionado.
   - Verificar se o nome fornecido é um módulo existente no código.
3. Verificar se no escopo original do subprograma selecionado existe alguma variável do tipo *PARAMETER* que seja usada apenas em seu interior.

4. Encontrar uma referência do módulo de destino no código-fonte.
    - Adicionar o comando *CONTAINS* no módulo, caso já não o tenha.
    - Adicionar as variáveis identificadas no passo 3 na seção de declarações do módulo, e remover suas declarações do escopo original do código.
5. Adicionar o subprograma selecionado abaixo do comando *CONTAINS* do módulo.
6. Remover a definição do código selecionado do escopo original.
7. Remover o comando *CONTAINS* do escopo original, caso ele não possua outros subprogramas.
8. Verificar em quais escopos o subprograma é usado, adicionando o comando *USE*, seguido do nome do módulo de destino, no início de cada escopo onde ocorre o uso.
9. Compilar e testar.

3.1.2.3 Exemplos

A Figura 3.3 mostra um código que contém um módulo que define um tipo *ponto*. No programa principal, existe uma sub-rotina que faz a soma de dois pontos.

```
MODULE mod_ponto
   TYPE ponto
      REAL :: x, y
   END TYPE ponto
END MODULE mod_ponto
PROGRAM p_ponto
   USE mod_ponto
   TYPE(ponto) :: px, py, pz
   px%x = 1.0
   px%y = 2.0
   py%x = 1.0
   py%y = 4.0
   CALL add(px, py, pz)
   PRINT*, "pz = ", pz%x, ",", pz%y
CONTAINS
   SUBROUTINE add(px, py, pz)
      TYPE(ponto), INTENT(in) :: px, py
      TYPE(ponto), INTENT(out):: pz
      pz%x = px%x + py%x
      pz%y = px%y + py%y
   END SUBROUTINE add
END PROGRAM p_ponto
```

Figura 3.3: Soma de dois pontos no programa principal

Seria vantajoso que a sub-rotina *add* estivesse definida no módulo, para ser usada em qualquer região do código onde o tipo *ponto* fosse usado. A Figura 3.4 mostra a sub-rotina movida para o interior do módulo, com o uso dessa refatoração. Assim, a soma de dois pontos poderá ser realizada em todo o escopo em que o módulo for usado.

```
MODULE mod_ponto
   TYPE ponto
      REAL :: x, y
   END TYPE ponto
CONTAINS
   SUBROUTINE add(px, py, pz)
      TYPE(ponto), INTENT(in) :: px, py
      TYPE(ponto), INTENT(out):: pz
      pz%x = px%x + py%x
      pz%y = px%y + py%y
   END SUBROUTINE add
END MODULE mod_ponto
PROGRAM p_ponto
   USE mod_ponto
   TYPE(ponto) :: px, py, pz
   px%x = 1.0
   px%y = 2.0
   py%x = 1.0
   py%y = 4.0
   CALL add(px, py, pz)
   PRINT*, "pz = ", pz%x, ",", pz%y
END PROGRAM p_ponto
```

Figura 3.4: Soma de dois pontos no interior do módulo

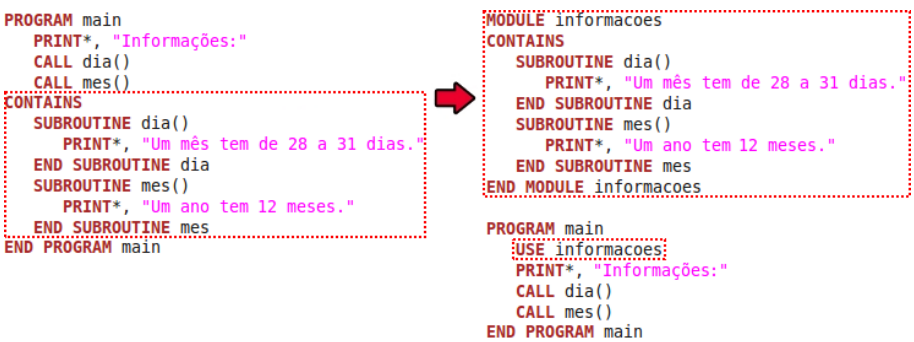
3.1.3 Extrair Subprograma para Módulo

Você deseja usar um subprograma em qualquer escopo do código-fonte.

♢ ♢ ♢

Extraia o subprograma para um novo módulo, e utilize-o onde for necessário.

♢ ♢ ♢



3.1.3.1 Motivação

Os módulos são um dos novos recursos introduzidos pelo Fortran 90. Um módulo pode ser usado para transferir dados entre sub-rotinas (módulos de variáveis globais) e para organizar a arquitetura global de um programa grande e complexo.

A funcionalidade de um módulo pode ser explorada por qualquer unidade de programa que deseja usar (através de uma instrução *USE*) seus recursos disponibilizados, tais como: declaração de objetos globais, blocos de interfaces, subprogramas de módulos, acesso controlado a objetos, interfaces genéricas, sobrecarga de operadores e extensão semântica.

O foco desta refatoração está no uso de subprogramas de módulo. As sub-rotinas ou funções podem ser definidas internamente em um módulo, sendo tornadas acessíveis a qualquer unidade de programa que use o módulo através do comando *USE*. Essa estratégia é mais vantajosa que usar um subprograma externo, pois em um módulo a interface dos subprogramas internos é sempre explícita, facilitando a programação.

Essa refatoração organiza o código-fonte, extraindo sub-rotinas e funções específicas para módulos específicos, podendo-se utilizar os subprogramas inseridos no módulo em qualquer escopo do código-fonte que faça uso do módulo. Caso existam variáveis declaradas como *PARAMETER* no código-fonte, e elas sejam usadas apenas no interior do subprograma a ser extraído, elas também são extraídas para o módulo criado, sendo declaradas apenas em seu interior.

3.1.3.2 Mecânica

1. Criar um módulo com um nome global único.

2. Aplicar *Mover Subprograma para um Módulo* no subprograma desejado, conforme visto na seção 3.1.2.

3. Compilar e testar.

3.1.3.3 Exemplos

A Figura 3.5 mostra um código Fortran 77 que calcula o seno e o logaritmo de um valor multiplicado pelo coeficiente de *Euler*. Nesse código, as funções *sen()* e *ln()* podem ser usadas apenas pelo programa principal, pois só são reconhecidas neste escopo, não podendo ser usadas em outros componentes de software. A Figura 3.6 mostra a aplicação dessa refatoração no código apresentado, permitindo usar as funções onde for necessário.

```
PROGRAM maths
   REAL, PARAMETER :: pi = 3.1415926536
   REAL, PARAMETER :: euler_e = 2.718281828
   REAL :: x

   PRINT*, "Entre com o valor de x: "
   READ*, x

   PRINT*, "sen(pi*x) = ", sen(x)
   PRINT*, "ln(e*x) = ", ln(x)

CONTAINS

   FUNCTION sen(x)
      REAL :: sen, x
      sen = sin(pi*x)
      RETURN
   END FUNCTION sen

   FUNCTION ln(x)
      REAL :: ln, x
      ln = log(euler_e*x)
      RETURN
   END FUNCTION ln

END PROGRAM maths
```

Figura 3.5: Cálculo de seno e de logaritmo no programa principal

```
MODULE senln
   REAL, PARAMETER :: pi = 3.1415926536
   REAL, PARAMETER :: euler_e = 2.718281828
CONTAINS

   FUNCTION sen(x)
      REAL :: sen, x
      sen = sin(pi*x)
      RETURN
   END FUNCTION sen

   FUNCTION ln(x)
      REAL :: ln, x
      ln = log(euler_e*x)
      RETURN
   END FUNCTION ln

END MODULE senln

PROGRAM maths
   USE senln
   REAL :: x

   PRINT*, "Entre com o valor de x: "
   READ*, x

   PRINT*, "sen(pi*x) = ", sen(x)
   PRINT*, "ln(e*x) = ", ln(x)

END PROGRAM maths
```

Figura 3.6: Cálculo de seno e de logaritmo no interior de um módulo

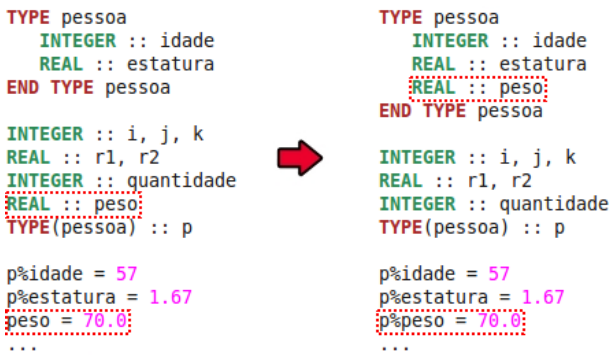
3.1.4 Mover Variáveis para um Tipo Derivado de Dados

Você dispõe de variáveis com semântica de uso compatível à semântica de uso de variáveis contidas em um tipo derivado de dados existente no código-fonte.

♢ ♢ ♢

Selecione tais variáveis e mova-as para esse tipo derivado de dados.

♢ ♢ ♢



3.1.4.1 Motivação

Essa refatoração permite re-organizar a distribuição de variáveis com semântica de uso semelhante em um código-fonte. Aplicando essa refatoração, é possível mover variáveis para um tipo derivado de dados qualquer existente no código-fonte da aplicação, desde que exista uma instância do mesmo sendo usada no escopo onde se encontra a declaração das variáveis a serem movidas para seu interior.

Outro motivo para a proposta dessa refatoração é dar apoio à refatoração *Extrair Variáveis para Tipo Derivado de Dados*, descrita na seção 3.1.5.

3.1.4.2 Mecânica

1. Identificar as variáveis a serem adicionadas ao tipo derivado de dados.
2. Localizar uma referência para o tipo derivado de dados de destino.
3. Adicionar as variáveis ao tipo derivado de dados de destino.
4. Remover as declarações das variáveis do escopo original do código-fonte.

5. Identificar uma instância do tipo derivado a ser usada nas referências das variáveis.

6. Substituir as referências das variáveis pela instância do tipo derivado de dados.

7. Compilar e testar.

3.1.4.3 Exemplos

Como exemplo do uso dessa refatoração, pode ser considerado o caso de um código que contém diversas variáveis declaradas, onde o programador identificou uma relação nas variáveis que caracterizam os lados de um objeto geométrico. A refatoração *Extrair Variáveis para Tipo Derivado de Dados* foi usada para extrair essas variáveis para o tipo *lados*, mas a variável *lado_d* não foi incluída no tipo criado, como mostra a Figura 3.7.

```
TYPE lados
   INTEGER :: lado_a
   INTEGER :: lado_b
   INTEGER :: lado_c
END TYPE lados
INTEGER :: linha_a
INTEGER :: vertice_a
INTEGER :: lado_d
TYPE(lados) :: lado
lado%lado_a = 10
lado%lado_b = 20
lado%lado_c = 10
lado_d = 10
```

Figura 3.7: Identificando variável a ser adicionada ao tipo *lados*

Com a aplicação dessa refatoração, é possível adicionar a variável *lado_d* ao tipo *lados*, e substituir as referências da variável pela instância *lado* do tipo derivado de dados, que é usada no código-fonte, como pode ser observado na Figura 3.8.

```
TYPE lados
   INTEGER :: lado_a
   INTEGER :: lado_b
   INTEGER :: lado_c
   INTEGER :: lado_d
END TYPE lados
INTEGER :: linha_a
INTEGER :: vertice_a
TYPE(lados) :: lado
lado%lado_a = 10
lado%lado_b = 20
lado%lado_c = 10
lado%lado_d = 10
```

Figura 3.8: Variável *lado_d* adicionada ao tipo *lados*

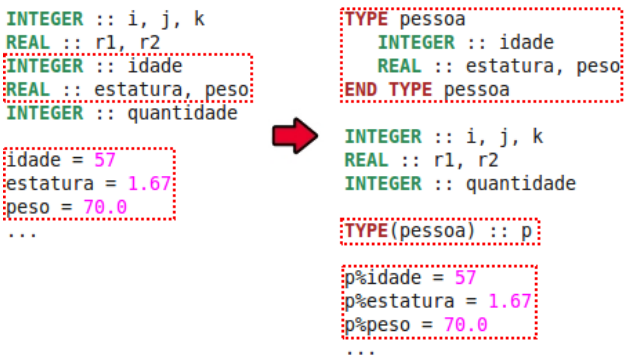
3.1.5 Extrair Variáveis para Tipo Derivado de Dados

Você tem um código Fortran 77 com variáveis relacionadas entre si que atuam sobre um mesmo atributo de software.

♢ ♢ ♢

Extraia as variáveis relacionadas entre si para um novo tipo derivado de dados.

♢ ♢ ♢



3.1.5.1 Motivação

Outro recurso introduzido no Fortran 90 é a possibilidade de o programador definir seus próprios tipos derivados de dados (estruturas de dados). O uso desse recurso facilita a programação, e possibilita uma abstração das variáveis, permitindo o agrupamento de alguns atributos em uma entidade (estrutura), facilitando o seu acesso e sua modificação.

Extraindo variáveis semelhantes que atuam sobre um mesmo artefato de software para um tipo derivado de dados, o código pode ficar mais claro, podendo facilitar futuras manutenções do sistema.

3.1.5.2 Mecânica

1. Identificar variáveis que atuam em conjunto sobre um mesmo interesse do sistema.
2. Criar um novo tipo derivado de dados, e uma instância do mesmo, nomeando-os de acordo com a semântica de uso das variáveis selecionadas.
3. Aplicar *Mover Variáveis para um Tipo Derivado de Dados* em tais variáveis, conforme visto na seção 3.1.4.
4. Compilar e testar.

3.1.5.3 Exemplos

Como exemplo do uso dos tipos derivados pode-se citar o uso de um ponto geométrico de duas coordenadas (x e y), e de um triângulo. No Fortran 77, por exemplo, para manipular um ponto geométrico e um triângulo, as variáveis correspondentes a cada coordenada do ponto e a cada lado do triângulo tinham de ser declaradas para cada instância do objeto a ser usado. A Figura 3.9 mostra um código em Fortran 77 que permite manipular um ponto e um triângulo.

```
INTEGER :: x, y
INTEGER :: a, b, c
! Ponto XY
x = 1
y = 0
! Triangulo ABC
a = 3
b = 4
c = 5
```

Figura 3.9: Manipulação de formas geométricas

Em Fortran 90, usando tipos derivados de dados, pode-se criar o tipo *ponto* (contendo dois atributos) e o tipo *triangulo* (contendo três atributos). A Figura 3.10 mostra a criação desses dois tipos derivados. Com os tipos derivados, seria possível instanciar e manipular diversos pontos e triângulos, sem ter de repetir códigos sequenciais de declaração de variáveis semelhantes. A Figura 3.10 mostra também a declaração de uma variável do tipo *ponto* e uma variável do tipo *triangulo*. Cada componente de um tipo derivado pode ser acessado com o uso do seletor de componente do Fortran (%).

```
TYPE ponto
   INTEGER :: x, y
END TYPE ponto
TYPE triangulo
   INTEGER :: a, b, c
END TYPE triangulo
TYPE(ponto) :: p
TYPE(triangulo) :: t

! Ponto XY
p%x = 1
p%y = 0
! Triangulo ABC
t%a = 3
t%b = 4
t%c = 5
```

Figura 3.10: Manipulação de formas geométricas com tipo derivado de dados

Um detalhe a ser observado é que a única operação que pode ser feita diretamente entre dois tipos derivados do Fortran, é a cópia do valor de uma variável para outra, por exemplo, *ponto1 = ponto2*. Para efetuar outras operações, como adição, subtração, multiplicação, divisão, entre outras, deve-se usar sobrecargas de operadores ou criar funções auxiliares que recebam como argumentos os tipos derivados a serem modificados, e retornem uma nova variável do tipo derivado com o resultado da operação (ANSI, 1991).

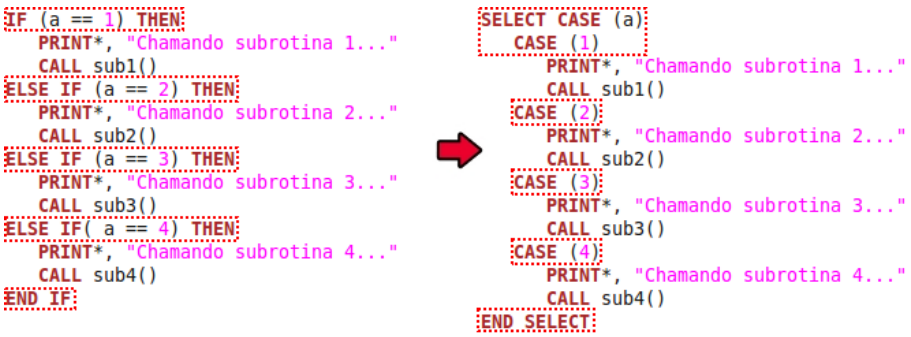
3.1.6 Converter *IF-THEN-ELSE* Aninhados em *SELECT CASE*

Você tem um código com comandos *IF-THEN-ELSE* aninhados que avaliam a igualdade de valores de uma única variável.

♢ ♢ ♢

Converta os comandos aninhados em uma estrutura de controle SELECT CASE.

♢ ♢ ♢



3.1.6.1 Motivação

No Fortran 77, quando se desejava escolher um valor dentre vários possíveis para uma única variável, a escolha podia ser feita apenas utilizando-se comandos *IF-THEN-ELSE* aninhados.

No Fortran 90, foi adicionada a estrutura de controle *SELECT CASE*, que possibilita fazer a mesma operação sem o uso de comandos *IF-THEN-ELSE* aninhados, melhorando a legibilidade de código. Usando essa refatoração, é possível percorrer o código-fonte em busca dos comandos aninhados e substituí-los pela nova estrutura de controle, atualizando o padrão da linguagem de programação usada.

3.1.6.2 Mecânica

1. Identificar comandos *IF-THEN-ELSE* no código-fonte.
2. Verificar se existem comandos *IF-THEN-ELSE* aninhados em cada expressão *ELSE*.
3. Verificar se são feitas apenas comparações de igualdade (==) e ou-lógico (*.or.*) nas expressões de condição (apenas esses dois tipos de comparação são necessários para avaliar a igualdade de uma única variável em uma construção *SELECT CASE*).
4. Identificar se apenas uma variável é avaliada nos comandos aninhados, e armazenar uma referência para ela.
5. Substituir a construção *IF-THEN-ELSE* pela construção *SELECT CASE*, usando a variável avaliada nos comandos aninhados.
    - Usar cada valor de comparação dos comandos aninhados como um *CASE*.
    - Usar o mesmo corpo de cada comparação em cada *CASE*.
6. Compilar e testar.

3.1.6.3 Exemplos

Como exemplo de uso dessa refatoração, pode-se considerar o caso de uma aplicação que faz a conversão de uma data no formato *dia, mês, ano* para o dia da semana correspondente. A aplicação usa um algoritmo simples que calcula em que dia da semana cai um determinado dia, mês e ano.

Se a aplicação para converter o formato da data fosse desenvolvida em Fortran 77, seriam usados comandos *IF-THEN-ELSE* aninhados para fazer a verificação do resultado do dia da semana (Figura 3.11).

Com a aplicação da refatoração *Substituir IF-THEN-ELSE Aninhados por SELECT CASE*, é possível substituir os comandos *IF-THEN-ELSE* aninhados que testam a igualdade da variável *resultado* pela estrutura de controle *SELECT CASE* do Fortran 90, promovendo a evolução do programa (Figura 3.12).

```
PROGRAM dia_da_semana
   INTEGER dia, mes, ano, x, resultado
   WRITE (*,*) "Escreva: dia, mes, ano"
   READ (*,*) dia, mes, ano
   IF (mes >= 3) THEN
      mes = mes - 2
      x = 8
   ELSE
      ano = ano - 1
      x = 13
   END IF
   resultado = mod((x+dia+(31*mes)/12+(5*ano)/4-(3*(1+ano/100)/4)),7)
   IF (resultado == 0) THEN
      print*, "Sábado."
   ELSE IF (resultado == 1) THEN
      print*, "Domingo."
   ELSE IF (resultado == 2) THEN
      print*, "Segunda-feira."
   ELSE IF (resultado == 3) THEN
      print*, "Terça-feira."
   ELSE IF (resultado == 4) THEN
      print*, "Quarta-feira."
   ELSE IF (resultado == 5) THEN
      print*, "Quinta-feira."
   ELSE IF (resultado == 6) THEN
      print*, "Sexta-feira."
   END IF
END PROGRAM dia_da_semana
```

Figura 3.11: Aplicação usando comandos *IF-THEN-ELSE* aninhados

```
PROGRAM dia_da_semana
   INTEGER dia, mes, ano, x, resultado
   WRITE (*,*) "Escreva: dia, mes, ano"
   READ (*,*) dia, mes, ano
   IF (mes >= 3) THEN
      mes = mes - 2
      x = 8
   ELSE
      ano = ano - 1
      x = 13
   END IF
   resultado = mod((x+dia+(31*mes)/12+(5*ano)/4-(3*(1+ano/100)/4)),7)
   SELECT CASE (resultado)
      CASE (0)
         print*, "Sábado."
      CASE (1)
         print*, "Domingo."
      CASE (2)
         print*, "Segunda-feira."
      CASE (3)
         print*, "Terça-feira."
      CASE (4)
         print*, "Quarta-feira."
      CASE (5)
         print*, "Quinta-feira."
      CASE (6)
         print*, "Sexta-feira."
   END SELECT
END PROGRAM dia_da_semana
```

Figura 3.12: Aplicação usando *SELECT CASE*

3.2 Refatorações para a Evolução de Fortran 90 para Fortran 95

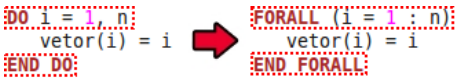
3.2.1 Converter laços *DO* em construções *FORALL*

Você tem um código com laços *DO* operando sobre vetores e matrizes, e deseja melhorar seu desempenho de execução.

♢ ♢ ♢

Converta os laços DO em construções FORALL, podendo executar os laços paralelamente quando a arquitetura de execução for multiprocessada.

♢ ♢ ♢



3.2.1.1 Motivação

Com o padrão Fortran 95 surgiu a construção *FORALL*, que permite uma execução mais eficiente de laços *DO*. Quando um laço *DO* é executado, o processador deve realizar cada iteração sucessiva em ordem[1], com uma atribuição após a outra. Isto representa um impedimento severo na otimização do código em uma plataforma paralelizada.

A ideia da construção *FORALL* é oferecer ao programador uma estrutura que possibilite os mesmos recursos obtidos com laços *DO*, porém que sejam automaticamente paralelizáveis, quando o programa estiver sendo executado em uma plataforma paralela.

Existe uma restrição quanto aos tipos de operações que podem ser realizadas em uma construção *FORALL*. Nesse tipo de construção, são permitidas apenas operações de atribuição de vetores e de matrizes, não sendo permitido o uso de outros recursos da linguagem, tais como laços *DO* aninhados, comandos de desvio, entre outros.

Com a aplicação dessa refatoração, os laços *DO* que contêm apenas operações de atribuição em seu interior são convertidos em construções *FORALL*, possibilitando que as atribuições individuais sejam realizadas em qualquer ordem, inclusive simultaneamente, caso o programa seja executado em uma arquitetura paralela.

[1] alguns compiladores (como os compiladores Fortran da empresa The Portland Group (PGI, 2011)) paralelizam automaticamente a execução de laços *DO*, não sendo necessária a aplicação dessa refatoração quando esse tipo de compilador é usado.

3.2.1.2 *Mecânica*

1. Selecionar o laço *DO* a ser substituído (podem ser laços aninhados).
2. Verificar se são realizadas apenas operações de atribuição no interior do laço.
3. Substituir o comando *DO* por *FORALL*, mantendo o mesmo corpo do laço.
4. Ajustar a sintaxe dos índices do laço para atender ao padrão usado no *FORALL*.
5. Compilar e testar.

3.2.1.3 *Exemplos*

Como exemplo pode-se considerar a decomposição LU de uma matriz. Um algoritmo padrão para a decomposição LU transforma uma matriz quadrada "no mesmo lugar", do inglês "*in place*", armazenando os elementos de L e de U no mesmo espaço de memória onde a matriz original foi armazenada (PRESS et al., 1992). O algoritmo para a decomposição LU pode ser implementado com o uso de laços de repetição *DO*, como mostra a Figura 3.13. A forma refatorada do algoritmo pode ser observada na Figura 3.14.

```
DO k = 1, n-1
   DO x = k+1, n
      A(x,k) = A(x,k) / A(k,k)
   END DO
   DO i = k+1, n
      DO j = k+1, n
         A(i,j) = A(i,j) - A(i,k) * A(k,j)
      END DO
   END DO
END DO
```

Figura 3.13: Decomposição LU de uma matriz usando laços *DO*

```
DO k = 1, n-1
   FORALL (x = k+1:n)
      A(x,k) = A(x,k) / A(k,k)
   END FORALL
   FORALL (i = k+1:n)
      FORALL (j = k+1:n)
         A(i,j) = A(i,j) - A(i,k) * A(k,j)
      END FORALL
   END FORALL
END DO
```

Figura 3.14: Decomposição LU de uma matriz usando construções *FORALL*

3.3 Refatorações para a Evolução de Fortran 95 para Fortran 2003

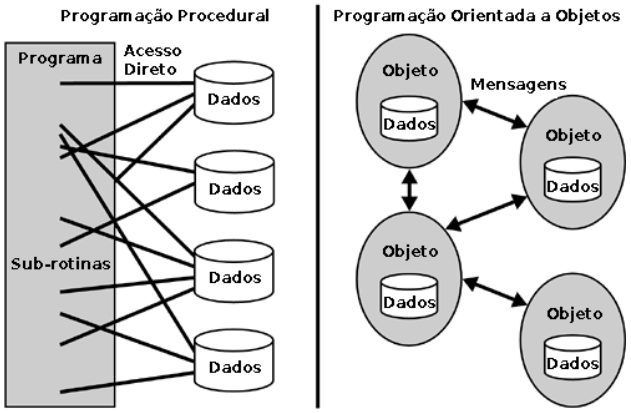
3.3.1 Converter Programa Procedural em Programa Orientado a Objetos

Você deseja transformar um código procedural em um código orientado a objetos.

♢ ♢ ♢

Transforme os dados em objetos, quebre o comportamento, e mova-o para os objetos.

♢ ♢ ♢



3.3.1.1 Motivação

Fortran não é fundamentalmente uma linguagem orientada a objetos, mas alguns recursos introduzidos no Fortran 2003, como a palavra-chave *CLASS* e seu uso em conjunto com outros recursos da linguagem, permitem que um programador escreva códigos no paradigma orientado a objetos. Desta forma, tem-se uma nova forma de manipulação e organização dos artefatos de software de um sistema, permitindo também a construção de um novo código baseado em rotinas já existentes.

A conversão de códigos procedurais em códigos orientados a objetos é discutida em diversos trabalhos. Alguns tratam sobre a evolução incremental de códigos procedurais em códigos orientados a objetos com o uso de *frameworks* que permitem automatizar partes do processo (ZOU; KONTOGIANNIS, 2001, 2003). Outros abordam o mapeamento de padrões de projetos procedurais em padrões de projetos orientados a objetos, para converter o paradigma de programação usado em uma aplicação (PIDAPARTHI; LUKER; ZEDAN, 1999; LANO; MALIK, 1999). Nesse contexto, a proposta dessa refatoração é fazer a conversão de códigos procedurais escritos em Fortran em códigos orientados a

objetos, utilizando os recursos disponibilizados pela linguagem. Essa refatoração é baseada na refatoração *Convert Procedural Design to Objects* (FOWLER et al., 1999), com a diferença de que essa refatoração é destinada à evolução de códigos Fortran.

3.3.1.2 Mecânica

1. Transformar cada tipo de registro em um objeto de dado individual com métodos de acesso.
2. Colocar todo o código procedural dentro de uma única classe.
3. Quebrar códigos procedurais longos em métodos menores.
    - Mover cada método gerado para a classe de dados apropriada.
4. Continuar até remover todo o comportamento da classe original.
5. Compilar e testar.

3.3.1.3 Exemplos

Um exemplo do uso dessa refatoração pode ser observado a seguir, onde é feito o cálculo da área de um círculo. O exemplo é bastante simples, mas serve para ilustrar o conceito da transformação almejada por essa refatoração. A Figura 3.15 mostra o código procedural que faz o cálculo da área de um círculo.

```
PROGRAM circuloProcedural
   IMPLICIT NONE
   REAL :: pi = 3.1415926535897931d0
   REAL :: raio, area
   raio = 1.5 ! Define o raio do círculo
   CALL calcularArea(area)
   CALL imprimirCirculo()
CONTAINS
   SUBROUTINE calcularArea(area)
      REAL :: area
      area = pi * raio**2
   END SUBROUTINE calcularArea

   SUBROUTINE imprimirCirculo()
      PRINT*, "Raio = ", raio, " Area = ", area
   END SUBROUTINE imprimirCirculo
END PROGRAM circuloProcedural
```

Figura 3.15: Aplicação com código procedural

A Figura 3.16 mostra uma forma de transformar o código procedural da aplicação em código orientado a objetos, usando a mecânica descrita nessa refatoração.

```
MODULE classeCirculo
   IMPLICIT NONE

   PRIVATE
   REAL :: pi = 3.1415926535897931d0

   TYPE, PUBLIC :: Circulo
      REAL :: raio

   CONTAINS

      PROCEDURE :: area => calcularArea

      PROCEDURE :: imprimir => imprimirCirculo

   END TYPE Circulo

CONTAINS

   FUNCTION calcularArea(this) RESULT(area)

      CLASS(Circulo), INTENT(IN) :: this
      REAL :: area

      area = pi * this%raio**2

   END FUNCTION calcularArea

   SUBROUTINE imprimirCirculo(this)

      CLASS(Circulo), INTENT(IN) :: this
      REAL :: area

      ! Chama a função calcularArea()
      area = this%area()
      PRINT*, "Raio = ", this%raio, " Area = ", area

   END SUBROUTINE imprimirCirculo

END MODULE classeCirculo

PROGRAM circuloOObjeto
   USE classeCirculo
   IMPLICIT NONE

   ! Instância de um círculo
   TYPE(Circulo) :: c

   ! Construtor implicito, raio = 1.5
   c = Circulo(1.5)

   ! Imprime o círculo
   CALL c%imprimir()

END PROGRAM circuloOObjeto
```

Figura 3.16: Aplicação com código orientado a objetos

Com o uso de orientação a objetos é possível definir classes e métodos para manipular os objetos, facilitando futuras construções da aplicação, e possibilitando que mais de um objeto (no caso um círculo) possa ser instanciado, sem ter de repetir códigos sequenciais para cada instanciação.

3.4 Refatorações para a Evolução de Fortran 2003 para Fortran 2008

3.4.1 Converter laços *DO* em laços *DO CONCURRENT*

Você deseja melhorar o desempenho de execução dos laços *DO* em seu código-fonte.

♢ ♢ ♢

Converta os laços DO em laços DO CONCURRENT.

♢ ♢ ♢

3.4.1.1 Motivação

A forma *DO CONCURRENT*, introduzida no Fortran 2008, permite que o programador afirme que não há dependências de dados entre as iterações do laço, e possibilita a ativação de otimizações como vetorização, desdobramento de laços e *multi-threading*.

As construções *DO CONCURRENT* e *FORALL* possuem duas diferenças importantes: o *FORALL* é essencialmente para operações com vetores e matrizes; e existem menos restrições quanto aos tipos de comandos que podem aparecer em uma construção *DO CONCURRENT*. Com essa refatoração é possível melhorar o desempenho de execução dos laços de repetição do código-fonte que não possuam dependências de dados.

3.4.1.2 Mecânica

1. Selecionar o laço *DO* a ser substituído.
2. Verificar a existência de dependência de dados no laço.
    - Caso não haja dependências, inserir a diretiva *CONCURRENT* e alterar a forma de uso dos índices para o padrão *DO CONCURRENT*.
3. Compilar e testar.

3.4.1.3 Exemplos

Um exemplo do uso de laços *DO CONCURRENT* pode ser observado a seguir. A Figura 3.17 mostra um laço *DO* convencional, iterando sobre elementos de um vetor. Observe o uso de um comando de desvio do tipo *IF-ELSE-END IF* no interior do laço de repetição. Se o laço em questão fosse submetido à refatoração *Converter laços DO em construções FORALL*, a refatoração não poderia ser concluída, pois existe uma operação que não é uma atribuição no interior do laço.

```
DO i=1, m
   IF (i < m/2) THEN
      a(k+i) = a(k+i) + factor*a(l+i)
   ELSE
      a(k+i) = factor*a(l+i)
   END IF
END DO
```

Figura 3.17: Uso de laço *DO* convencional

Como o laço apresentado não possui dependências de dados, é possível então transformá-lo em um laço *DO CONCURRENT* (Figura 3.18).

```
DO CONCURRENT (i=1:m)
   IF (i < m/2) THEN
      a(k+i) = a(k+i) + factor*a(l+i)
   ELSE
      a(k+i) = factor*a(l+i)
   END IF
END DO
```

Figura 3.18: Uso de laço *DO CONCURRENT*

3.4.2 Converter MPI em *Coarrays*

Você tem um código que usa MPI e deseja melhorar sua legibilidade, mantendo a paralelização sem usar bibliotecas externas, usando a mesma sintaxe do Fortran.

♢ ♢ ♢

Converta as construções que usam a interface MPI em operações nativas de coarrays, paralelizando a execução do código-fonte com o uso de coarrays do Fortran 2008.

♢ ♢ ♢

```
INCLUDE 'mpif.h'
INTEGER :: idProcesso, numProcessos, ierr
CALL MPI_INIT(ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, idProcesso, ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, numProcessos, ierr)
PRINT *, "Executando processo: ", idProcesso
PRINT *, "Total de processos: ", numProcessos
CALL MPI_FINALIZE(ierr)
```

```
INTEGER :: idProcesso, numProcessos
idProcesso = THIS_IMAGE()
numProcessos = NUM_IMAGES()
PRINT *, "Executando processo: ", idProcesso
PRINT *, "Total de processos: ", numProcessos
```

3.4.2.1 Motivação

Coarrays consistem em uma extensão introduzida no Fortran 2008 para a programação paralela. São variáveis especiais que podem ser compartilhadas entre várias instâncias do mesmo programa, chamadas de "imagens". A principal vantagem do uso de *coarrays* é o elevado nível de integração com a linguagem Fortran, fazendo com que os programas fiquem mais legíveis se comparados ao uso de bibliotecas externas que promovem o paralelismo, como a MPI (*Message Passing Interface*) (MPI FORUM, 1993), por exemplo.

A categoria de paralelismo dos *coarrays* se enquadra em SPMD (*Single-Program, Multiple-Dada*), semelhante ao modelo seguido pela interface MPI: uma aplicação é executada como um processo independente que passa e recebe dados através da rede ou por memória compartilhada. Assim como as aplicações que usam a interface MPI, as aplicações que usam *coarrays* podem ser executadas em sistemas de memória compartilhada ou em sistemas de memória distribuída, como *clusters*, por exemplo.

Estudos sobre a transformação de código com MPI em código com *coarrays* demonstram que o desempenho do uso de *coarrays* é muito semelhante ao desempenho obtido

com o uso de MPI, tendo a vantagem de se ter um código mais legível e de mais fácil manutenção quando utilizados os *coarrays* do Fortran (ASHBY; REID, 2008).

Os *coarrays* são parecidos com matrizes do Fortran, exceto que são acessados através de colchetes em vez de parênteses. Os colchetes indicam uma referência a um *coarray* em outra imagem (ou talvez na mesma). Um exemplo simples de declaração de um *coarray* pode ser *integer :: variavel_coarray[*]*. Nesse exemplo, é declarada a variável *variavel_coarray* como um número inteiro que é compartilhável através de imagens. Em uma expressão, *variavel_coarray* refere-se ao *coarray* na imagem atual, enquanto *variavel_coarray[1]* refere-se ao *coarray* da imagem um, e assim por diante. A notação *[*]* é usada pois os limites do *coarray* são determinados em tempo de execução, e não em tempo de compilação (para configurar os limites do *coarray*, os dados são passados por argumentos através da linha de comando, como *./app images=10*, por exemplo).

Em operações com variáveis do tipo *coarray*, no lado direito de uma atribuição, o valor é carregado a partir de uma imagem. No lado esquerdo de uma atribuição, o valor é armazenado para essa imagem. A instrução *variavel_coarray[1] = variavel_coarray[2]* faz com que o valor da variável *variavel_coarray* da imagem dois seja carregado e armazenado na variável *variavel_coarray* da imagem um. Essa instrução tem o mesmo efeito quando executada em qualquer imagem.

Assim, com o uso dessa refatoração, pode-se usar *coarrays* no código-fonte, conseguindo paralelizar a execução da aplicação, sem depender de bibliotecas externas que promovem o paralelismo, facilitando a programação e a manutenção do aplicativo.

3.4.2.2 Mecânica

1. Procurar as regiões do código onde a paralelização é feita.
2. Identificar as variáveis compartilhadas.
3. Redeclarar as variáveis identificadas no passo 2 como *coarrays*.
4. Substituir as operações *SEND/RECEIVE* por operações de atribuição, com lógica equivalente à do MPI.
5. Sincronizar as operações.
6. Compilar e testar.

3.4.2.3 Exemplos

Como exemplo do uso dessa refatoração, pode-se considerar o caso de fazer a troca de valores entre matrizes em processos distintos. A Figura 3.19 mostra um trecho de código que usa MPI para fazer a troca entre vizinhos norte e sul de uma matriz.

```
COMMON/XCTILB4/ B(N,4)
SAVE  /XCTILB4/
CALL MPI_SEND(B(1,1),N,MPI_REAL,IMG_N,9905,MPI_COMM_WORLD,MPIERR)
CALL MPI_SEND(B(1,2),N,MPI_REAL,IMG_S,9906,MPI_COMM_WORLD,MPIERR)
CALL MPI_RECV(B(1,3),N,MPI_REAL,IMG_S,9905,MPI_COMM_WORLD,MPISTAT,MPIERR)
CALL MPI_RECV(B(1,4),N,MPI_REAL,IMG_N,9906,MPI_COMM_WORLD,MPISTAT,MPIERR)
```

Figura 3.19: Troca de valores entre matrizes usando MPI

Cada imagem primeiramente envia seus *buffers* e então espera pela chegada dos *buffers* correspondentes das imagens vizinhas. Todas as chamadas bloqueiam até ser seguro reusar a variável compartilhada *B*.

Quando a mesma operação é realizada utilizando-se *coarrays*, a maneira de expressar a troca de valores entre as matrizes de diferentes processos é mais simples e mais legível, como pode ser observado no trecho de código exibido na Figura 3.20.

```
COMMON/XCTILB4/ B(N,4)[*]
SAVE  /XCTILB4/
CALL SYNC_ALL(WAIT=(/IMG_S,IMG_N/))
B(:,3) = B(:,1)[IMG_S]
B(:,4) = B(:,2)[IMG_N]
CALL SYNC_ALL(WAIT=(/IMG_S,IMG_N/))
```

Figura 3.20: Troca de valores entre matrizes usando *Coarrays*

A primeira chamada *SYNC_ALL* faz com que o processo espere até que a imagem remota *B(:,1:2)* esteja pronta para ser copiada, e a segunda chamada faz com que o processo espere até que seja seguro sobrescrever a variável local *B(:,1:2)*. Somente os vizinhos mais próximos são envolvidos na sincronização.

3.5 Considerações sobre o Catálogo

O objetivo deste trabalho é a criação de refatorações que permitem evoluir códigos Fortran legados. As dez refatorações descritas no catálogo almejam evoluir construções obsoletas ou de uso desencorajado da linguagem para construções mais recentes, que são indicadas para melhorar a qualidade dos sistemas de software legados.

Normalmente é necessário evoluir aplicativos Fortran para adequá-los a novos padrões da linguagem e para facilitar a manipulação de seus códigos em rotinas de manutenção que fazem parte do ciclo de vida de um aplicativo de software. De acordo com as leis da evolução de software, vistas na seção 2.3.1, um sistema de software deve sofrer mudanças contínuas para continuar satisfatório aos usuários. Além disso, a complexidade de um sistema aumenta à medida em que ele é atualizado. Em programas escritos na linguagem Fortran, a complexidade de entendimento da lógica e da legibilidade dos aplicativos pode ser bastante elevada se os códigos forem atualizados seguindo padrões antigos da linguagem. Por exemplo, para fazer operações consideradas complexas em códigos legados, poderiam ser necessárias diversas linhas de comando, enquanto as mesmas operações poderiam ser reduzidas a poucas linhas de código quando usadas algumas construções mais atuais da linguagem. Um exemplo disso é o caso das operações matriciais, vistas na seção 3.1.1. Evoluindo o padrão da linguagem de programação usada nos trechos de códigos que fazem a manipulação de matrizes em uma aplicação, podem-se obter resultados de melhora de legibilidade de código e melhora de desempenho de execução da aplicação.

A evolução da linguagem Fortran foi bastante significativa desde o seu surgimento. Diversos padrões do Fortran foram lançados, sendo cada vez mais aprimorados e apresentado novos recursos. Como visto na seção 2.2, cada versão da linguagem recebeu alguns recursos fundamentais para sua evolução, que permitem a construção de programas científicos de alto desempenho, utilizando-se apenas recursos específicos da linguagem.

As refatorações propostas no catálogo focam em recursos diferenciados de cada padrão da linguagem Fortran. Procurou-se não considerar as pequenas mudanças sintáticas inseridas em cada versão da linguagem como oportunidades de refatoração para a evolução de programas, como a adição da diretiva *PARAMETER* para especificar variáveis constantes em um código-fonte, por exemplo. O foco das refatorações propostas leva em consideração apenas os recursos adicionados em cada versão da linguagem que oferecem maiores vantagens e facilidades de programação, como a modularização de código,

criação de tipos derivados de dados, melhora de legibilidade de código e melhora de desempenho na execução das aplicações. Considerando esse foco de atuação, a qualidade do aplicativo de software é melhorada quando ele é refatorado com alguma das refatorações descritas no catálogo.

Além da evolução dos códigos legados, as refatorações presentes no catálogo têm o objetivo de melhorar a legibilidade e a organização estrutural do código-fonte dos aplicativos, facilitando o processo de programação e de manutenção dos mesmos. Com o uso dessas refatorações, é possível seguir passos bem definidos para a evolução dos programas legados, permitindo o crescimento contínuo dos sistemas e mantendo sua qualidade, como previsto nas leis da evolução de software, descritas na seção 2.3.1.

4 AUTOMAÇÃO DE REFATORAÇÕES

4.1 Photran

Durante o desenvolvimento deste trabalho, algumas das refatorações propostas no catálogo foram automatizadas como extensões do IDE Eclipse. O Eclipse é um IDE de desenvolvimento que tem suporte para diversas linguagens de programação, como C, C++, Java, e recentemente Fortran, com o uso do *plugin* Photran.

Photran (ECLIPSE.ORG, 2011a) é um IDE para código Fortran baseado na plataforma Eclipse (BEATON; RIVIERES, 2006), sendo uma extensão do *plugin* CDT (*C/C++ Development Tools*) (ECLIPSE.ORG, 2011b). Assim como outras ferramentas que são extensões do Eclipse, o Photran é desenvolvido em linguagem Java e é composto de *plugins* e de recursos (*features*). *Plugins* adicionam funcionalidades ao Eclipse, enquanto os recursos são unidades de desenvolvimento (diversos *plugins* são empacotados em recursos, que são distribuídos aos usuários). Arquiteturalmente, o projeto Photran divide-se em subprojetos onde cada um se constitui de um *plugin* ou um recurso.

O Photran oferece suporte à programação em linguagem Fortran 77, 90, 95, 2003 e 2008, com ênfase na refatoração automatizada de códigos Fortran (CHEN; OVERBEY, 2010). A Figura 4.1 mostra a IDE Photran em execução, sendo possível observar três abas distintas nessa aplicação. A primeira mostra o diretório do projeto, contendo os arquivos de código-fonte, bibliotecas, etc. A segunda aba é referente ao editor de código-fonte, enquanto a terceira é referente a visualização da *Outline View*, que exibe a AST (*Abstract Syntax Tree*) do código-fonte em tempo real durante sua edição.

O Photran possui um analisador sintático completo e uma representação de programa, com o Photran VPG (*Virtual Program Graph*), que gera ASTs (*Abstract Syntax Tree*) do código-fonte, permitindo a sua manipulação e edição, além de outras utilidades (OVERBEY, 2007). Com isso, o Photran oferece uma infraestrutura para refatoração de código-

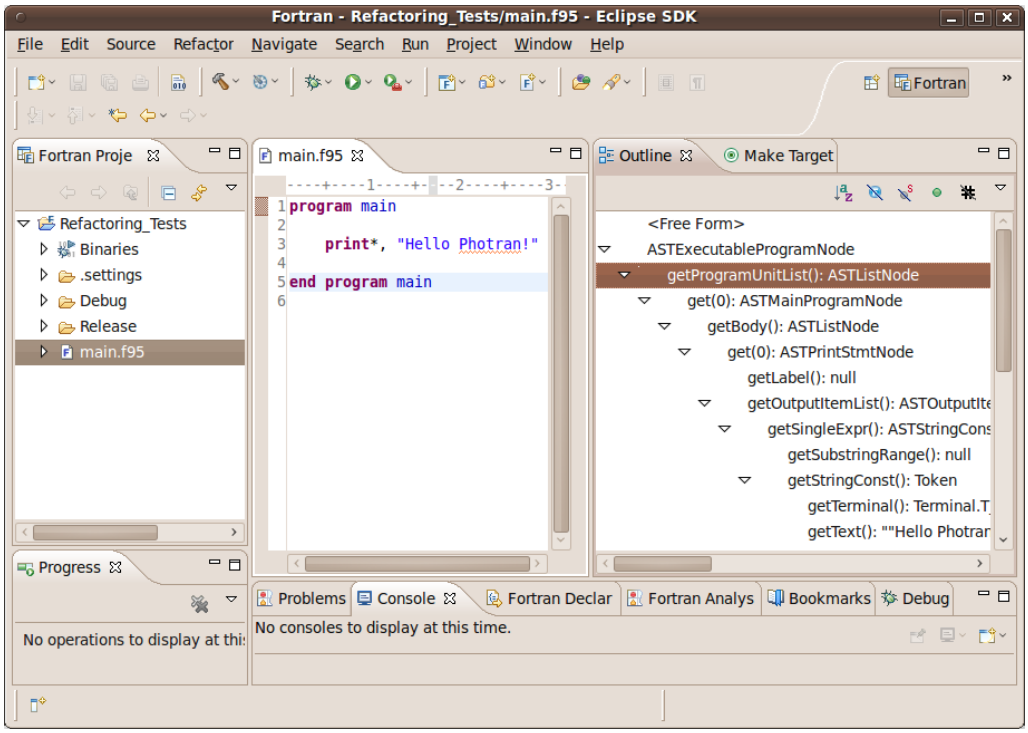


Figura 4.1: Visualização da IDE Photran em execução

fonte que consiste de representações abstratas do programa (que permitem navegar e alterar sua estrutura em alto nível), mecanismos para visualizar e comparar diferenças antes e depois da refatoração e, ainda, meios de cancelar ou desfazer uma refatoração efetuada.

Uma árvore sintática abstrata (AST) é uma estrutura para representação do código-fonte. Ela é composta por uma raiz da qual são derivados nós, sendo que no último nível dessa árvore normalmente são representados os elementos da gramática da linguagem de programação (*tokens*) (JONES, 2003). Cada nó é classificado de acordo com seu funcionamento e com suas ações. Por exemplo, um nó do tipo declaração derivará outros nós que representarão o nome da variável declarada e o tipo da mesma, podendo ainda conter um valor para a inicialização da variável.

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

A AST é uma peça chave para automatizar refatorações, dado que ela possibilita ao desenvolvedor navegar pela estrutura do código-fonte, detectando correlações entre seus nós e identificando oportunidades de refatoração (OVERBEY; JOHNSON, 2009). Quando é

disponibilizada uma estrutura de representação como a AST, a refatoração consiste em introduzir modificações diretamente na AST, fazendo inclusão, exclusão ou alterando o posicionamento de determinados nós da árvore. O Photran possibilita que o programador visualize a AST em tempo real enquanto está programando, através da *Outline View*. A Figura 4.2 mostra um código Fortran no IDE Photran e sua AST na *Outline View*.

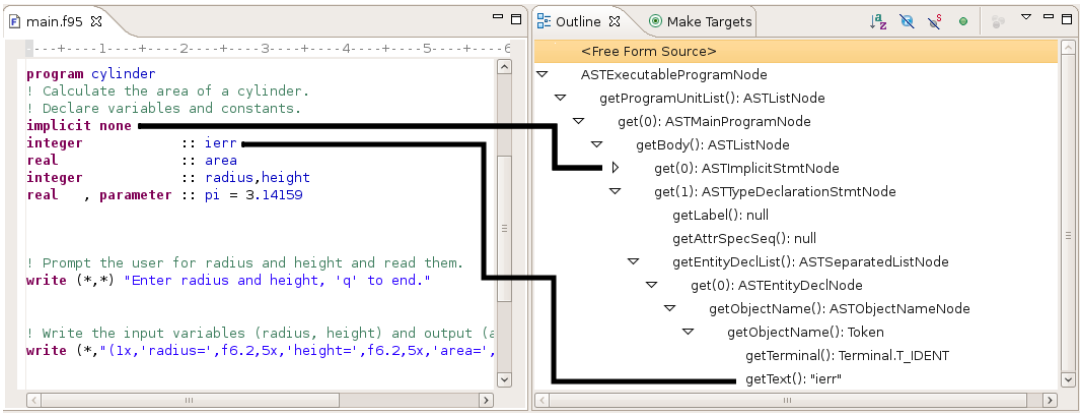


Figura 4.2: Código Fortran e sua AST na IDE Photran

O analisador sintático do Photran, além de fornecer a AST, gera o VPG (*Virtual Program Graph*), que é uma estrutura que possibilita a agregação de outras ligações entre os nós da árvore sintática abstrata, não representando necessariamente a hierarquia da árvore. Algumas das ligações adicionais podem ser, por exemplo, um vínculo entre a utilização de determinada variável (em uma expressão) e sua respectiva declaração. É por meio do VPG que se pode obter um conjunto de informações para subsidiar a manipulação dos nós de uma AST.

O VPG pode ser visto como um conjunto de arestas que ligam os nós de uma AST de forma não hierarquizada (OVERBEY, 2007), como pode ser observado em um exemplo simples mostrado na Figura 4.3.

Na Figura 4.3, os nós (elipses) com ligações de linhas contínuas são os nós que compõem a AST. Os nós retangulares representam os *tokens* no código do programa fonte. As ligações extras (em linhas tracejadas), que são rotuladas, transformam a AST em um VPG. Nesse exemplo, o VPG apresenta dois tipos de ligação, a *binding*, que liga referências de variáveis às suas respectivas declarações, e *scope of declaration*, que liga cada declaração com o nó que representa o seu escopo na AST.

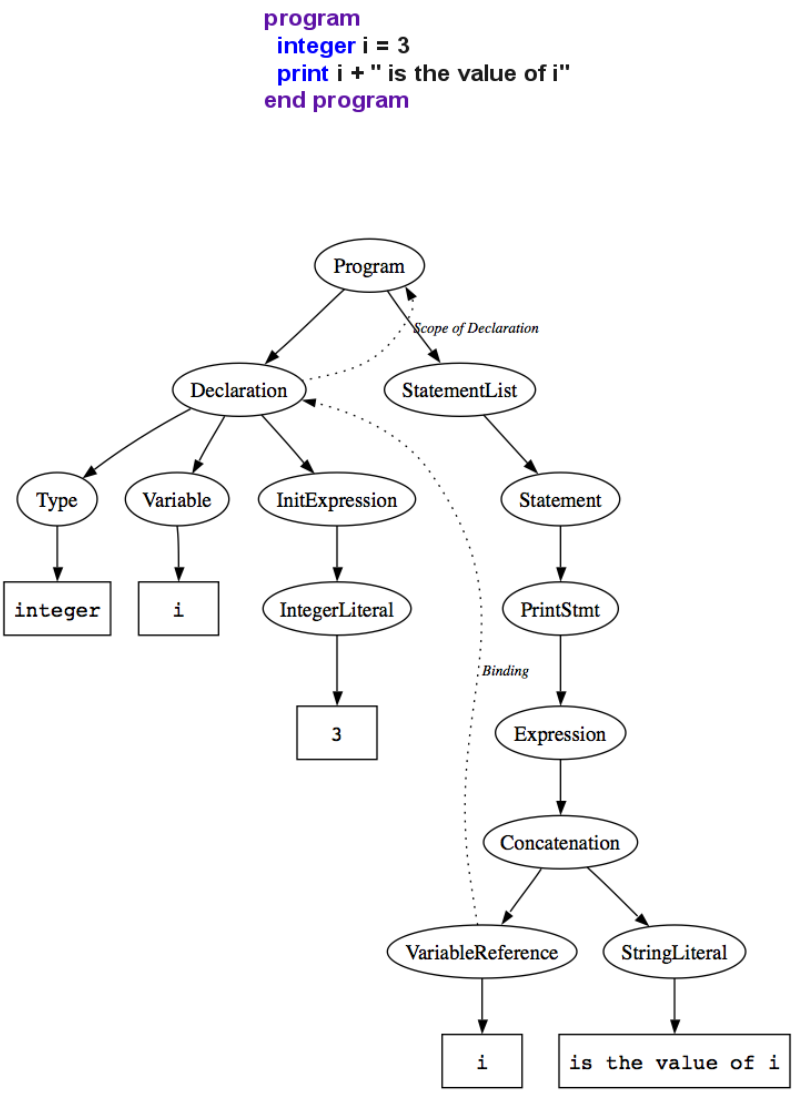


Figura 4.3: Visualização de um programa e de seu VPG (OVERBEY, 2007)

4.2 Infraestrutura para a Automação de Refatorações

Para automatizar uma refatoração no Photran são necessários três passos: fazer uma pré-validação da refatoração; fazer a manipulação da AST (introduzindo as mudanças no código-fonte); e fazer uma pós-validação da refatoração, para garantir a integridade da AST. O Photran possibilita atuar sobre o código Fortran por meio da manipulação de ASTs e da utilização da base de informações existentes nos VPGs. Existem métodos que possibilitam navegar sobre ASTs, recuperar informações acerca de seus nós, e ainda executar operações de atualização sobre elas (remoção, adição ou substituição de nós).

Para implementar e integrar uma refatoração no Photran, é necessário estender o comportamento de algumas classes presentes em seu *framework*, como mostra a Figura 4.4.

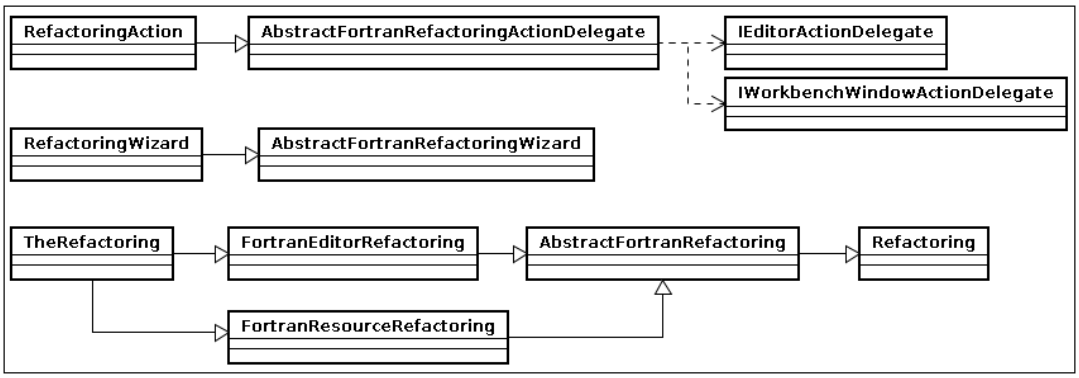


Figura 4.4: Diagrama de classes a serem estendidas

A primeira classe (*RefactoringAction*) é responsável por receber a chamada do usuário e associar a refatoração a seu respectivo assistente. Essa classe deve estender a classe *AbstractFortranRefactoringActionDelegate* e deve implementar os métodos de duas interfaces: *IWorkbenchWindowActionDelegate*, permitindo que a chamada do usuário seja feita a partir do menu principal, e *IEditorActionDelegate*, que permite ao usuário fazer a chamada a partir de um menu de contexto no próprio editor.

A segunda classe a ser estendida é o assistente da refatoração (*RefactoringWizard*). É comum que uma refatoração solicite ao usuário alguma informação adicional para que possa ser executada, como um novo nome para uma variável, um nome para um novo método, etc. Essa classe é estendida de *AbstractFortranRefactoringWizard* e tem objetivo de fazer a construção gráfica da janela do assistente. Seu método construtor recebe como parâmetro um objeto para a terceira classe necessária (*TheRefactoring*), que faz a refatoração propriamente dita.

A classe que faz a refatoração pode estender a superclasse *FortranEditorRefactoring* ou a superclasse *FortranResourceRefactoring*, sendo que ambas são classes específicas do Photran. A primeira se destina a fazer refatorações que são aplicadas a um arquivo por vez, que exigem entradas de dados pelo usuário, como a seleção de um trecho do código-fonte a ser refatorado, por exemplo, e a segunda é destinada às refatorações que podem ser aplicadas a lotes de arquivos, que não exigem entradas de dados pelo usuário, realizando pequenas alterações independentes nos arquivos. Essas duas classes, por sua vez, estendem a classe *AbstractFortranRefactoring*, que também é específica do Photran, e que estende a classe *Refactoring*, a qual faz parte do *framework* do Eclipse. Na classe da refatoração (*TheRefactoring*), quatro métodos precisam ser codificados:

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

- ***doCheckInitialConditions*:** esse método serve para fazer uma verificação das pré-condições da AST e dos critérios exigidos pela refatoração usada. O método pode lançar uma exceção *PreconditionFailure* indicando que alguma condição para a refatoração não foi satisfeita e que ela não será realizada;

- ***doCreateChange*:** esse método aplica a refatoração propriamente dita, fazendo as modificações na AST, caso a pré-validação tenha sido positiva. Ao término de tais modificações, este método chama dois outros métodos, o *addChangeFromModifiedAST()*, que tem o objetivo de adicionar as modificações feitas à AST utilizada pelo Photran e o método *releaseAllASTs()*, visando forçar o Photran a atualizar os controles visuais da AST assim como sua validação;

- ***doCheckFinalConditions*:** esse método é responsável por verificar as pós-condições, que confirmam ou não as alterações aplicadas sobre a AST. Esse método também pode lançar uma exceção *PreconditionFailure* indicando que alguma condição para a refatoração não foi satisfeita e que ela não será realizada.

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

4.3 Refatorações Automatizadas

A escolha da IDE Eclipse e do *plugin* Photran como ferramentas para a automação das refatorações propostas neste trabalho é justificada por essas ferramentas disponibilizarem uma infraestrutura completa e bem documentada para a criação e incorporação de novas refatorações para a linguagem Fortran. Além disso, tais ferramentas são distribuídas como software livre, e possuem uma grande rede de colaboradores, permitindo que o suporte ao programador seja aprimorado através da troca de informações entre seus colaboradores em listas de discussão.

Existem também outras ferramentas disponíveis para a manipulação e re-estruturação de códigos Fortran, como a plusFORT (POLYHEDRON, 2011), que oferece transformações para evoluir determinadas construções de uma versão de Fortran para outra. Porém,

essa ferramenta é proprietária, não sendo possível acessar seu código fonte para adicionar novas funcionalidades. Outra ferramenta que permite realizar refatorações em códigos Fortran é a ROSE (QUINLAN et al., 2011). Essa ferramenta consiste em uma infraestrutura com recursos para efetuar transformações em códigos Fortran, distribuída como software livre, permitindo a adição de novas funcionalidades. No entanto, outro fator determinante na escolha do Photran como ferramenta para a automação das refatorações propostas neste trabalho é a intenção de dar continuidade a trabalhos realizados anteriormente pelo autor e por colegas de laboratório (BONIATI, 2009; RISSETTI; CHARÃO; BONIATI, 2010; BONIATI et al., 2010, 2011; TIETZMANN et al., 2011), nos quais essa mesma ferramenta foi usada para a automatizar outras refatorações para Fortran.

Das dez refatorações propostas neste trabalho, seis foram implementadas no *plugin* Photran: *Extrair Subprograma para Módulo*, *Mover Subprograma para um Módulo*, *Extrair Variáveis para Tipo Derivado de Dados*, *Mover Variáveis para um Tipo Derivado de Dados*, *Converter IF-THEN-ELSE Aninhados em SELECT CASE* e *Converter laços DO em construções FORALL*. As cinco primeiras tratam da evolução de código Fortran 77 para Fortran 90, enquanto a última se refere à evolução de Fortran 90 para Fortran 95.

Algumas das refatorações propostas no catálogo ainda são inviáveis de serem automatizadas (em sua totalidade), tendo de ser aplicadas manualmente. Esse é o caso das refatorações *Converter laços DO em Operações Matriciais Nativas*, *Converter Programa Procedural em Programa Orientado a Objetos* e *Converter MPI em Coarrays*. Essas refatorações exigem muitos casos distintos de análise para atuar em um código-fonte, sendo difícil automatizar o processo de refatoração. A refatoração *Converter laços DO em laços DO CONCURRENT* também não pôde ser implementada no Photran pois alguns dos recursos mais recentes do Fortran 2008 (como os laços *DO CONCURRENT*) ainda não estão implementados em alguns compiladores, e suas sintaxes ainda não são reconhecidas pelo Photran, gerando inconsistências na criação das ASTs e impossibilitando a aplicação de refatorações no código-fonte dos aplicativos.

No Photran, as refatorações implementadas foram: *Extract Subroutine or Function to Module*, *Move Subroutine or Function to Module*, *Transform to Derived Data Type*, *Add Variable to Derived Data Type*, *Nested If-Then-Else to Select Case* e *Replace Do Loop by Forall*. As sub-seções seguintes discutem os detalhes de implementação de cada uma delas (além dessas implementações, uma refatoração extra é mostrada no Apêndice A).

4.3.1 *Extract Subroutine or Function to Module*

Essa é uma refatoração que necessita de argumentos de entrada para sua execução. Para implementá-la no Photran, foi necessário criar a classe *ExtractSubroutineOrFunctionToModuleRefactoring* e estender o comportamento da classe *FortranEditorRefactoring*.

4.3.1.1 Pré-condições

Na implementação, solicita-se que o usuário selecione uma sub-rotina ou função para ser extraída para um novo módulo. No método *doCheckInitialConditions()*, é feita uma verificação para ter certeza de que o usuário selecionou realmente uma sub-rotina ou uma função, que na AST do Photran, correspondem, respectivamente, aos tipos de nó *ASTSubroutineSubprogramNode* e *ASTFunctionSubprogramNode*. Caso o trecho de texto selecionado não corresponda a um dos dois tipos de nó, é exibida uma mensagem ao usuário informando que ele deve selecionar apenas uma sub-rotina ou uma função. Se o código selecionado é um subprograma válido, uma referência para o código é armazenada em uma variável do tipo *IASTNode*, nomeada como *selectedFunctionOrSubroutine*. Ainda nesse método, é guardada uma referência para o escopo original de onde será extraído o subprograma, em uma variável do tipo *ScopingNode*, nomeada como *originalScope*.

Após o usuário selecionar o código, é exibida uma caixa de diálogo requisitando que ele entre com um nome para a criação de um novo módulo para inserir a sub-rotina ou função selecionada, como mostra a Figura 4.5.

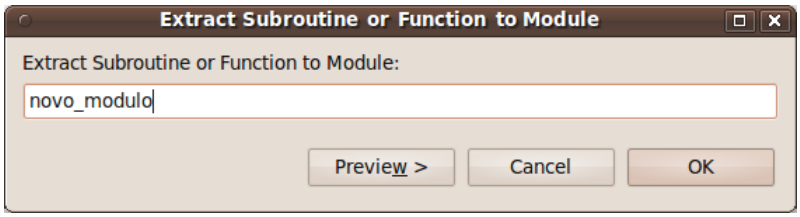


Figura 4.5: Requisitando o nome de um novo módulo

4.3.1.2 Pós-condições

No método *doCheckFinalConditions()*, são feitas verificações sobre o nome do módulo fornecido pelo usuário. É verificado se o nome contêm espaços em branco, pontos de exclamação, e se inicia com dígitos numéricos. Caso ocorra alguma das condições citadas, é exibida uma mensagem correspondente a cada condição, informando o problema

e solicitando que o usuário digite um nome válido para o módulo. Nesse método também é verificado se o nome fornecido pelo usuário já é usado no código-fonte. Caso esteja em uso, uma mensagem é exibida informando a ocorrência do problema, requisitando um novo nome para o módulo a ser criado.

4.3.1.3 Ações

No método *doCreateChange()*, é modificada a AST do Photran. Nesse método, inicialmente é encontrado o local onde o novo módulo deve ser inserido no código-fonte (no topo do arquivo). Para isso, com a chamada *astOfFileInEditor.getRoot().findFirstToken()*, é obtida uma referência para o início do arquivo, e então, após construir o nó referente ao módulo, ele é adicionado antes do primeiro caractere do código-fonte do arquivo.

O escopo de onde a função é extraída (*originalScope*) é percorrido em busca de definições de variáveis do tipo *PARAMETER*. Todas as definições de variáveis do escopo são obtidas com a chamada *originalScope.getAllDefinitions()*. Caso seja encontrada alguma variável do tipo *PARAMETER* entre as definições obtidas, e ela seja usada apenas no interior da sub-rotina ou função selecionada, ela é removida deste escopo e inserida no corpo do módulo a ser construído. Caso ela seja usada por outras sub-rotinas ou funções, ou até mesmo dentro do escopo original, sua declaração permanece inalterada.

Após coletar os dados, é criado um nó correspondente a um módulo na AST, com o nome fornecido pelo usuário. Em seu corpo, são inseridas as variáveis do tipo *PARAMETER* (caso existam) e a palavra-chave *CONTAINS*, seguida do subprograma selecionado. O código selecionado pelo usuário é removido de seu escopo original com a chamada *selectedFunctionOrSubroutine.removeFromTree()*, permanecendo apenas no interior do novo módulo. Se no escopo onde ocorreu a seleção só havia uma sub-rotina ou função (*if(originalScope.getInternalSubprograms().size()==1)*), a palavra-chave *CONTAINS* é removida do escopo com a chamada *originalScope.getContainsStmt().removeFromTree()*.

Por fim, todos os escopos do código-fonte são percorridos em busca de chamadas para o subprograma selecionado pelo usuário. Em cada escopo onde ocorre uma chamada, é adicionado o comando *USE* seguido do nome do módulo criado, para que a sub-rotina ou função seja visível e possa ser usada no escopo. Ao final do processo, a AST é atualizada com a chamada à função *addChangeFromModifiedAST()*, e a refatoração está concluída.

4.3.2 *Move Subroutine or Function to Module*

Essa também é uma refatoração que necessita de argumentos de entrada para sua execução. Para implementá-la no Photran, foi necessário criar a classe *MoveSubroutineOrFunctionToModuleRefactoring*, e estender o comportamento da classe *FortranEditorRefactoring*.

4.3.2.1 Pré-condições

Assim como na implementação da refatoração *Extract Subroutine or Function to Module* (vista na seção 4.3.1), na implementação dessa refatoração o usuário também deve selecionar um subprograma para movê-lo para um módulo existente. As mesmas verificações sobre a seleção do código realizadas no método *doCheckInitialConditions()* da seção 4.3.1.1 são feitas para essa refatoração. Porém, nessa refatoração é guardada uma referência para o escopo original de onde será movido o subprograma, em uma variável do tipo *ScopingNode*, nomeada como *originalScope*.

Após o usuário selecionar o código, é exibida uma caixa de diálogo requisitando a entrada de um nome de um módulo existente, para inserir a sub-rotina ou função selecionada no corpo do mesmo, como mostra a Figura 4.6.

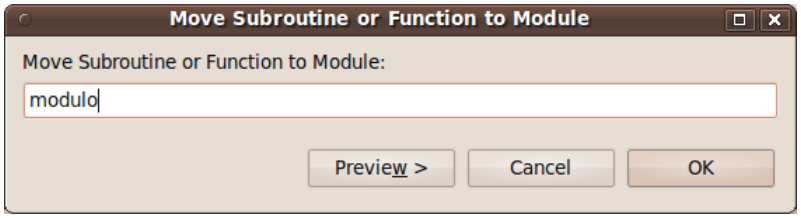


Figura 4.6: Requisitando o nome de um módulo existente

4.3.2.2 Pós-condições

No método *doCheckFinalConditions()* são feitas as mesmas verificações realizadas na seção 4.3.1.2, sobre o nome do módulo fornecido pelo usuário. Porém, nessa refatoração, caso não exista nenhum módulo com o nome fornecido, é exibida uma mensagem informando o problema, solicitando um nome válido. Caso o nome digitado seja válido, é guardada uma referência para o módulo em uma variável do tipo *ASTModuleNode*, nomeada como *module*.

4.3.2.3 Ações

No método *doCreateChange()*, é modificada a AST do Photran. O escopo de onde o subprograma será movido (*originalScope*) é percorrido em busca de definições de variáveis do tipo *PARAMETER*, como descrito na seção 4.3.1.3.

Antes de adicionar o subprograma selecionado no módulo de destino, é necessário verificar se o mesmo já possui a palavra-chave *CONTAINS*, através da chamada *module.getContainsStmt()*. Se o módulo já contém essa palavra-chave, o código selecionado é inserido imediatamente após o *CONTAINS* do módulo. Caso contrário, é necessário encontrar o final do módulo (*END MODULE*) e adicionar a palavra-chave *CONTAINS* seguida da sub-rotina ou função selecionada pelo usuário imediatamente antes do mesmo. Para localizar o final do módulo é usada a chamada *module.getEndModuleStmt().findFirstToken()*, que retorna uma referência para o início do comando *END MODULE*. O código selecionado pelo usuário é então removido de seu escopo original, com a chamada *selectedFunctionOrSubroutine.removeFromTree()*, permanecendo apenas no interior do módulo para onde o código foi movido. Se no escopo onde ocorreu a seleção só havia uma sub-rotina ou função (*if(originalScope.getInternalSubprograms().size()==1)*), a palavra-chave *CONTAINS* do escopo é removida com a chamada *originalScope.getContainsStmt().removeFromTree()*.

Por fim, todos os escopos do código-fonte são percorridos em busca de chamadas para o subprograma selecionado. Em cada escopo onde ocorre uma chamada, é adicionado o comando *USE*, seguido do nome do módulo para onde o código foi movido, para que o subprograma seja visível e possa ser usado no escopo. A refatoração é concluída com a chamada à função *addChangeFromModifiedAST()*, que atualiza a AST do código-fonte.

4.3.3 *Transform to Derived Data Type*

Nessa refatoração, é necessário que o usuário forneça argumentos de entrada para completar sua execução. Para implementá-la no Photran, foi necessário criar a classe *TransformToDerivedDataTypeRefactoring*, e estender o comportamento da classe *FortranEditorRefactoring*.

4.3.3.1 *Pré-condições*

Na implementação, o usuário deve selecionar um conjunto de declarações de variáveis que devem ser transformadas em um tipo derivado de dados. No método *doCheckInitialConditions()*, é verificado se o usuário selecionou apenas declarações de variáveis. Caso ele tenha selecionado uma porção de código que não atenda a esse requisito, a refatoração é abortada, e é exibida uma mensagem indicando que apenas devem ser selecionadas declarações de variáveis. As variáveis selecionadas são armazenadas em uma lista do tipo *LinkedList<IBodyConstruct>*, nomeada como *statementsNodes*. Outra verificação feita antes de prosseguir com a refatoração, é quanto aos atributos usados nas declarações selecionadas. A definição de tipo derivado de dados não permite variáveis declaradas com atributos tais como *parameter, intent, target, optional, save, external* e *intrinsic*. Caso alguma das variáveis selecionadas apresente algum desses atributos, ela é removida da lista *statementsNodes*, e a refatoração continua normalmente com o restante das variáveis.

Após selecionar as variáveis, é exibida uma tela requisitando um nome para o tipo derivado de dados e um nome para uma instância do mesmo, como mostra a Figura 4.7.

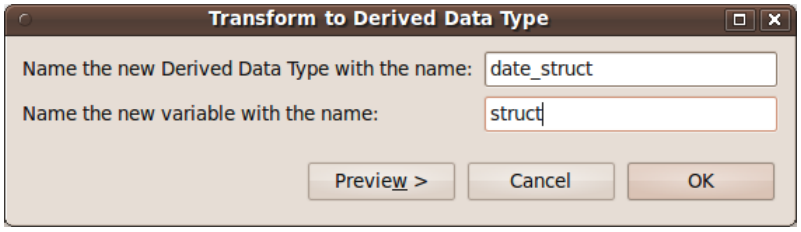


Figura 4.7: Requisitando o nome e uma instância do tipo derivado de dados

4.3.3.2 *Pós-condições*

No método *doCheckFinalConditions()*, é verificado se os nomes fornecidos pelo usuário contêm espaços em branco, pontos de exclamação, e se começam com dígitos numéricos. Caso alguma dessas condições ocorra em algum dos nomes, é exibida uma mensagem correspondente a cada condição, informando o problema, e solicitando um nome válido.

4.3.3.3 *Ações*

As ações da refatoração ocorrem no método *doCreateChange()*. Nesse método é localizado o escopo em que as variáveis foram selecionadas, com a chamada *astOfFileInEditor.getRoot().getEnclosingScope(node)*, onde *node* é o nó obtido na seleção das variáveis

através da chamada *findEnclosingNode(astOfFileInEditor, selectedRegionInEditor)*.

Duas situações podem ocorrer na busca pelo local onde o tipo derivado de dados deve ser inserido no escopo. Caso no escopo seja usado o comando *IMPLICIT NONE*, o tipo derivado deve ser adicionado imediatamente abaixo do comando, caso contrário, deve ser adicionado no topo do escopo. Após localizar e armazenar uma referência para o local onde o tipo derivado deve ser adicionado, é criado um novo nó na AST contendo o tipo derivado de dados, sendo que as declarações das variáveis selecionadas pelo usuário são colocadas dentro deste nó. Após criar o novo nó, ele é adicionado no local apropriado do escopo em que o usuário selecionou as variáveis, e uma instância do novo tipo é declarada nesse escopo, com o nome fornecido pelo usuário.

Depois de adicionar o nó contendo o tipo derivado de dados no escopo, as variáveis selecionadas pelo usuário são removidas da AST, ficando declaradas apenas dentro do nó criado. Por fim, a AST é percorrida em busca de referências das variáveis adicionadas ao tipo derivado. Quando uma referência é encontrada, ela é substituída pela referência da instância declarada do tipo derivado, com o modificador de acesso "%"referente à variável em questão, com a chamada *var.findToken().setText(derivedTypeVariableName + "%"+ var.findToken().getText())*, onde *var* é a referência encontrada da variável, e *derivedTypeVariableName* é o nome da instância do tipo derivado fornecido pelo usuário.

Ao final do processo, a AST é atualizada com a chamada à função *addChangeFromModifiedAST()*, e a refatoração está concluída.

4.3.4 *Add Variable to Derived Data Type*

Essa é outra refatoração que necessita de argumentos de entrada para sua execução. Para implementá-la no Photran, foi necessário criar a classe *AddVariableToDerivedDataTypeRefactoring*, e estender o comportamento da classe *FortranEditorRefactoring*.

4.3.4.1 Pré-condições

Na implementação, o usuário deve selecionar um conjunto de declarações de variáveis que devem ser adicionadas em um tipo derivado de dados existente (que contenha pelo menos uma instância sendo usada no escopo de seleção das variáveis). No método *doCheckInitialConditions()*, é feita a mesma verificação realizada na seção 4.3.3.1 para ver se o usuário selecionou apenas declarações de variáveis. As variáveis selecionadas são armazenadas em uma lista de variáveis do tipo *LinkedList<IBodyConstruct>*, nomeada

como *statementsNodes*. Outra verificação feita antes de prosseguir com a refatoração é quanto aos atributos usados nas declarações selecionadas, como descrita na seção 4.3.3.1. As variáveis que não satisfazerem as condições da verificação são removidas da lista *statementsNodes*, e a refatoração continua normalmente com o restante das variáveis contidas na lista.

Após selecionar as variáveis, é exibida uma tela requisitando o nome do tipo derivado de destino para as variáveis, e um nome de uma instância do mesmo usada no código-fonte, para usá-lo nas referências das variáveis em questão, como mostra a Figura 4.8.

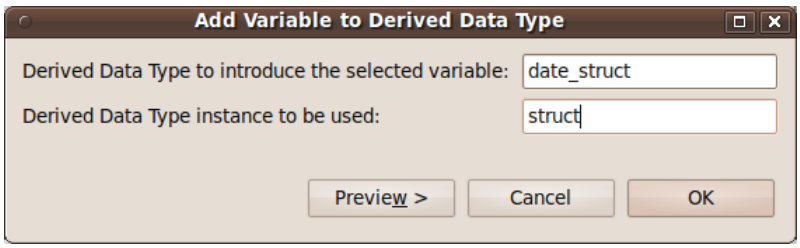


Figura 4.8: Requisitando o nome e uma instância de um tipo derivado existente

4.3.4.2 Pós-condições

No método *doCheckFinalConditions()*, são feitas as mesmas verificações realizadas na seção 4.3.3.2 sobre os nomes fornecidos pelo usuário. Nesse método também é verificado se o usuário digitou corretamente o nome de um tipo derivado de dados existente. Caso o nome fornecido seja inconsistente, é exibida uma mensagem informando que o nome digitado não existe, requisitando um nome correto. Caso o nome digitado esteja correto, uma referência para o tipo derivado de dados é armazenada em uma variável do tipo *PhotranTokenRef*, nomeada como *derivedDataTypeRef*. Também é feita uma verificação para ter certeza de que o usuário digitou corretamente o nome de uma instância do tipo derivado de dados a ser usada. Caso o nome digitado seja inconsistente, é exibida uma mensagem informando o problema e requisitando a entrada de um nome correto.

4.3.4.3 Ações

No método *doCreateChange()*, as variáveis selecionadas são adicionadas no início do tipo derivado de dados através da chamada *derivedDataTypeRef.findToken().setText(newStatements)*, onde *newStatements* é uma *string* que contém as variáveis que estavam na lista *statementsNodes*.

Depois de adicionadas ao tipo derivado de dados, as variáveis selecionadas são removidas da AST, ficando declaradas apenas dentro do nó correspondente ao tipo derivado de dados. Por fim, a AST é percorrida em busca de referências das variáveis adicionadas ao tipo derivado. Quando uma referência de uma das variáveis é encontrada, ela é substituída pela referência da instância do tipo derivado com o modificador de acesso "%"referente à variável em questão, com a chamada *var.findToken().setText(derivedTypeVariableName + "%"+ var.findToken().getText())*, onde *var* é a referência encontrada da variável, e *derivedTypeVariableName* é o nome da instância do tipo derivado fornecido pelo usuário.

Ao final do processo, a AST é atualizada com a chamada à função *addChangeFromModifiedAST()*, e a refatoração está concluída.

4.3.5 *Nested If-Then-Else to Select Case*

Essa é uma refatoração que pode ser aplicada a múltiplos arquivos simultaneamente, e o usuário não precisa fornecer argumentos de entrada para sua execução. Para implementá-la no Photran, foi necessário criar a classe *NestedIfThenElseToSelectCaseRefactoring*, e estender o comportamento da classe *FortranResourceRefactoring*.

4.3.5.1 Ações

Nessa refatoração, o método *makeChangesTo()* é quem faz as mudanças no código-fonte. Esse método é chamado para cada arquivo selecionado a ser refatorado. Inicialmente é feita uma busca de nós da AST do tipo *ASTIfConstructNode*, que contêm comandos *IF-THEN-ELSE* aninhados, através da chamada do método *node.getElseIfConstruct()*, que retorna a construção *ELSE-IF* de cada nó do tipo *ASTIfConstructNode*. Os nós que contém comandos *IF-THEN-ELSE* aninhados são armazenados em uma lista do tipo *List<ASTIfConstructNode>*, nomeada como *ifNodes*.

Após identificados todos os nós que contêm comandos aninhados, é feita a verificação sobre quais comparações são realizadas em cada nó. As comparações aceitas para o caso desta refatoração consistem em igualdade (==) e ou-lógico (*.or.*). Os nós que contêm outro tipo de comparação são descartados, sendo removidos da lista *ifNodes*.

São retirados os pares de caso/ação de cada nó restante na lista *ifNodes*. São obtidos de cada nó, o nome da variável sendo comparada, o valor com que a variável é comparada, e a ação que é feita para cada valor da variável. Após obter esses dados, é verificado se dentro do nó, em todos os níveis do *IF-THEN-ELSE*, apenas uma variável é avaliada.

Caso exista mais de uma variável sendo avaliada, o nó em questão é descartado, sendo também excluído da lista *ifNodes*.

Depois de obtidos todos os pares de caso/ação, um novo nó do tipo *SELECT CASE* é construído, onde para cada caso do valor da variável há uma ação a ser executada. Após construir o novo nó, o nó antigo, que continha o comando *IF-THEN-ELSE* aninhado, é substituído na AST pelo novo nó, através da chamada *ifNode.replaceWith(selectCaseNode)*, onde *ifNode* é o nó que contém os comandos *IF-THEN-ELSE* aninhados, e *selectCaseNode* é o novo nó, que contém a estrutura de controle *SELECT CASE*.

Ao final do processo, a AST do arquivo sendo refatorado é atualizada com a chamada à função *addChangeFromModifiedAST()*, e a refatoração está concluída.

4.3.6 *Replace Do Loop by Forall*

Essa é uma refatoração na qual o usuário deve fornecer argumentos de entrada (nesse caso, a seleção de um trecho de código). Para implementá-la no Photran, foi necessário criar a classe *ReplaceDoLoopByForallRefactoring*, e estender o comportamento da classe *FortranEditorRefactoring*.

4.3.6.1 Pré-condições

Na implementação, o usuário deve selecionar um laço *DO* a ser substituído por uma construção *FORALL*. No método *doCheckInitialConditions()*, é verificado se o usuário realmente selecionou um laço *DO*. Caso ele não tenha selecionado um laço *DO*, a refatoração é abortada, e uma mensagem é exibida informando o problema. Se a seleção estiver correta, uma referência ao nó da AST correspondente ao laço é armazenada em uma variável do tipo *ASTProperLoopConstructNode*, nomeada como *selectedDoLoop*.

Após selecionar o laço *DO*, é necessário verificar se no corpo do laço são feitas apenas operações de atribuição de valores (operações sobre vetores e matrizes), sem o uso de outras funcionalidades da linguagem, como desvios, declarações de variáveis, entre outros. Ou seja, para o laço *DO* satisfazer essas condições, no corpo do laço podem existir apenas nós do tipo *ASTAssignmentStmtNode*, que são nós de atribuição. Caso o usuário selecione laços *DO* aninhados, os nós correspondentes aos laços aninhados são armazenados em uma lista do tipo *LinkedList<ASTProperLoopConstructNode>*, nomeada como *nestedSelectedDoLoop*, e a mesma verificação realizada no nó principal é feita para cada um dos laços, e todos os laços selecionados são substituídos por *FORALL*.

Porém, na implementação dessa refatoração não são realizados testes para verificar se nos laços aninhados as condições[1] dos índices do *FORALL* são satisfeitas, ficando a cargo do usuário a responsabilidade por fazer tal verificação (o compilador normalmente avisa na forma de um *warning* quando esse tipo de condição não é satisfeita).

4.3.6.2 Ações

Após obter todos os nós que satisfazem as condições da refatoração, é então construído para cada laço *DO* selecionado, um nó do tipo *FORALL* (*ASTForallConstructNode*), contendo o mesmo corpo do laço *DO* correspondente. Depois de construir o nó do tipo *FORALL*, o nó antigo contendo o laço *DO* é substituído na AST pelo nó recém criado, através da chamada *node.replaceWith(forall)*, onde *node* é o nó que contém o laço *DO* e *forall* é o nó que contém a construção *FORALL*.

Ao final do processo, a AST é atualizada com a chamada à função *addChangeFromModifiedAST()*, e a refatoração está concluída.

[1] todos os índices usados no *FORALL* devem aparecer no lado esquerdo das atribuições, pois podem ocorrer múltiplas definições de valor para as variáveis da expressão caso algum deles não apareça no lado esquerdo da expressão.

5 AVALIAÇÃO

5.1 Avaliação das Refatorações Implementadas

Para avaliar o funcionamento das refatorações implementadas no Photran, as mesmas foram aplicadas em três aplicativos:

- SUPIM - *Sheffield University Plasmasphere Ionosphere Model*
- ASA159 - *Random Generation of a Table*
- MULTMAT - *Multiplicação Matricial*

Usando-se tais aplicativos, objetiva-se verificar o funcionamento das refatorações em códigos de diferentes versões da linguagem Fortran, que usam padrões antigos de programação, possibilitando uma evolução de seus códigos.

Os experimentos realizados com as aplicações SUPIM e ASA159 demonstram o funcionamento das refatorações *Extract Subroutine or Function to Module*, *Move Subroutine or Function to Module*, *Transform to Derived Data Type*, *Add Variable to Derived Data Type* e *Nested If-Then-Else to Select Case*. O objetivo desses experimentos consiste em refatorar o código-fonte dos aplicativos e comparar seus resultados de execução antes e depois da refatoração, para certificar que nenhuma anomalia tenha sido introduzida no código no processo de refatoração. Na avaliação dessas refatorações não foi necessária a realização de medidas de desempenho do aplicativo refatorado, uma vez que tais refatorações não introduzem mudanças na forma de execução do código.

Dentre as seis refatorações implementadas no *plugin*, apenas a refatoração *Replace Do Loop by Forall* é capaz de gerar diferenças de desempenho na execução da aplicação refatorada, visto que as demais apenas proporcionam mudanças sintáticas no código-fonte, não afetando o modo de executá-lo. Na avaliação dessa refatoração, foi usada a aplicação

MULTMAT, que é um código simples para a multiplicação entre duas matrizes. O objetivo do uso desse código é verificar se o desempenho de um aplicativo de software pode sofrer alterações positivas ou negativas com a aplicação dessa refatoração, uma vez que seu fluxo de execução pode ser alterado com a refatoração.

Para a realização dos experimentos, foram utilizados os compiladores GCC 4.4.1 (*gfortran*) e Intel Fortran Compiler 12.0.2 (*ifort*), no sistema operacional GNU/Linux Ubuntu 9.10, em um computador Intel Core 2 Duo 1.8 *GHz*, com 3 *GBytes* de memória.

Nas sub-seções seguintes são detalhados os experimentos realizados com cada refatoração para avaliar seu funcionamento.

5.1.1 Extrair e Mover Subprograma para um Módulo

Para avaliar as refatorações *Extract Subroutine or Function to Module* e *Move Subroutine or Function to Module*, as mesmas foram aplicadas no código SUPIM (BAILEY, 1978), que é uma aplicação escrita em Fortran 90, com construções de Fortran 77, usada em pesquisas de simulações na área de geofísica espacial (SOUZA, 1997) no Instituto Nacional de Pesquisas Espaciais (INPE). A aplicação foi disponibilizada em um arquivo único, contendo mais de 11 mil linhas de código. Porém, para realizar os experimentos, o código teve de ser particionado em oito arquivos, pois no modo de desenvolvimento do Photran não há espaço de memória para manipular um arquivo tão grande.

O código SUPIM é composto de diversas sub-rotinas e funções, sendo um forte candidato para o uso das refatorações *Extract Subroutine or Function to Module* e *Move Subroutine or Function to Module* para melhorar a organização de seu código-fonte.

Com o uso dessas duas refatorações foram gerados dois novos módulos no código-fonte da aplicação, um contendo sub-rotinas para a escrita de dados (*WRITER*), e um contendo sub-rotinas para cálculos sobre curvas *splines* (*SPLINECALCS*).

A Figura 5.1 mostra a extração da sub-rotina *WRIT15* para o módulo *WRITER*, com o uso da refatoração *Extract Subroutine or Function to Module*. Durante a aplicação da refatoração, o comando *USE WRITER* foi inserido no início do escopo do programa principal, onde a sub-rotina é usada.

Com o uso da refatoração *Move Subroutine or Function to Module* outras três sub-rotinas foram movidas para o módulo *WRITER* (*WRIT20, WRIT16* e *WRIT04*). A Figura 5.2 mostra a sub-rotina *WRIT04* sendo movida para o interior do módulo.

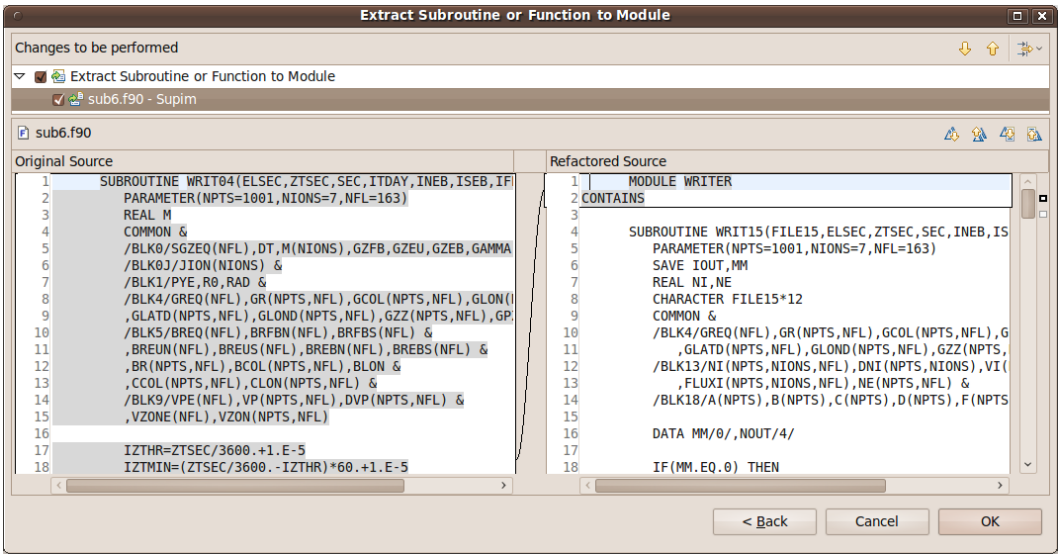


Figura 5.1: Extraindo a sub-rotina *WRIT15* para o módulo *WRITER*

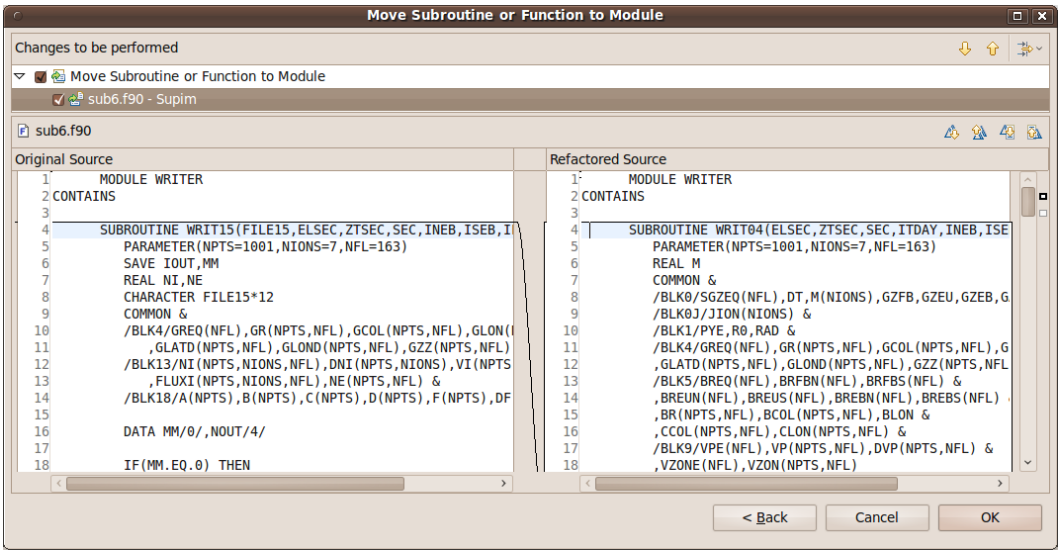


Figura 5.2: Movendo a sub-rotina *WRIT04* para o módulo *WRITER*

A Figura 5.3 mostra a extração da sub-rotina *SPLINE* para o módulo *SPLINECALCS*. Na refatoração, o comando *USE SPLINECALCS* foi adicionado nas funções *DENSU* e *DENSM*, que usam a sub-rotina extraída para o módulo. A Figura 5.4 mostra o comando *USE SPLINECALCS* sendo inserido na função *DENSU*.

Com o uso da refatoração *Move Subroutine or Function to Module*, mais duas sub-rotinas foram adicionadas ao módulo *SPLINECALCS* (*SPLINI* e *SPLINT*). A Figura 5.5 mostra a sub-rotina *SPLINT* sendo movida para o interior do módulo. As duas sub-rotinas movidas para o módulo também eram usadas apenas nas funções *DENSU* e *DENSM*, que já continham o comando *USE SPLINECALCS*.

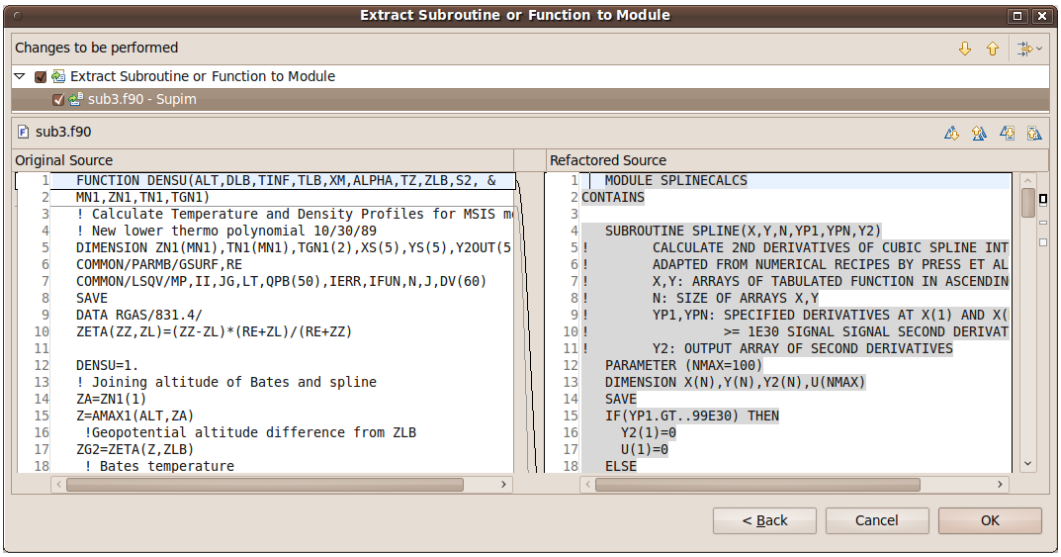


Figura 5.3: Extraindo a sub-rotina *SPLINE* para o módulo *SPLINECALCS*

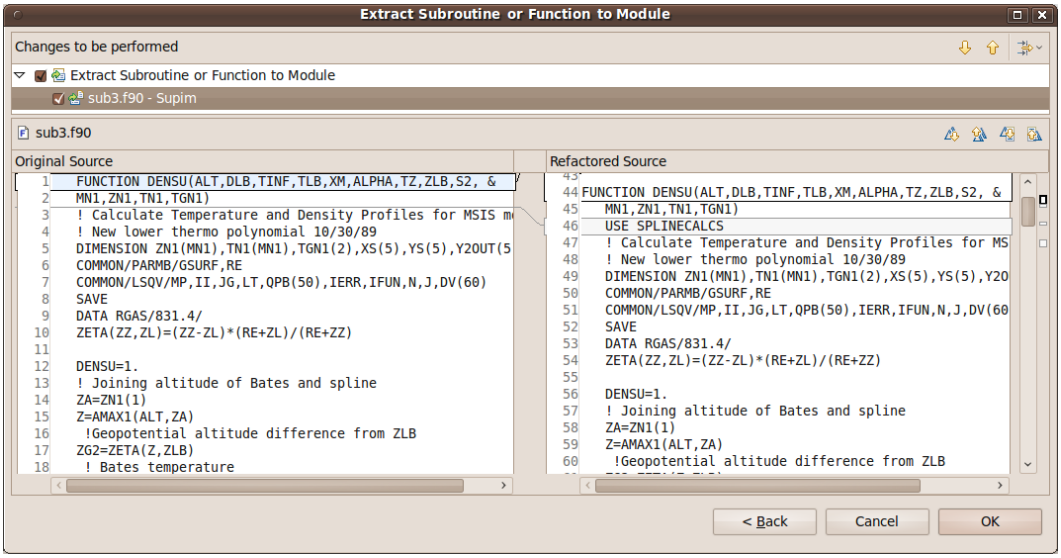


Figura 5.4: Inserindo o comando *USE SPLINECALS* na função *DENSU*

No experimento com as refatorações *Extract Subroutine or Function to Module* e *Move Subroutine or Function to Module* foi possível organizar melhor algumas sub-rotinas do código SUPIM, gerando dois novos módulos na aplicação e distribuindo sub-rotinas semelhantes no interior dos mesmos. Existem diversas outras sub-rotinas no aplicativo que podem ser extraídas e movidas para outros módulos, mas para o experimento realizado optou-se por criar apenas os módulos vistos anteriormente. Depois de refatorada, a aplicação executou normalmente, obtendo os mesmos resultados de execução obtidos antes da refatoração. Assim, pode-se concluir que as duas refatorações avaliadas cumpriram satisfatoriamente seus objetivos.

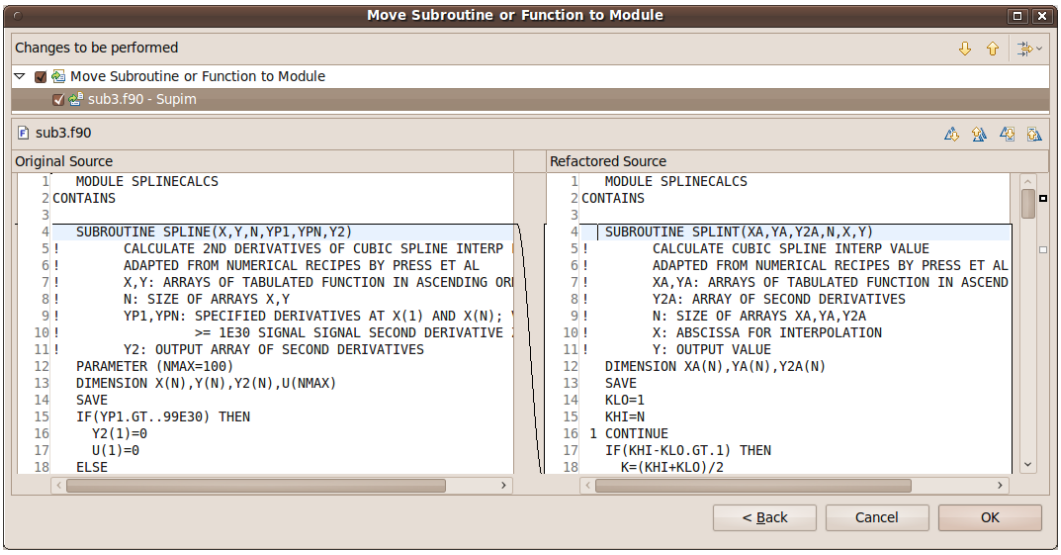


Figura 5.5: Movendo a sub-rotina *SPLINT* para o módulo *SPLINECALCS*

5.1.2 Extrair e Mover Variáveis para um Tipo Derivado de Dados

Para avaliar o funcionamento das refatorações *Transform to Derived Data Type* e *Add Variable to Derived Data Type*, foi usada a aplicação ASA159 - *Random Generation of a Table* (BURKARDT, 2000), que usa o algoritmo *AS 159* (PATEFIELD, 1981). Esse algoritmo recebe como entrada a forma de uma tabela (número de linhas e número de colunas), e dois vetores contendo as somas das linhas e colunas de uma tabela. Com os dados passados, o algoritmo consegue reconstruir a tabela original, a partir das somas das linhas e colunas.

Analisando o código da aplicação, foi observada a declaração de diversas variáveis em cada sub-rotina. Muitas delas atuam sobre atributos semelhantes do aplicativo, podendo então, ser expressadas na forma de um tipo derivado de dados. Com a aplicação da refatoração *Transform to Derived Data Type* foi gerado um tipo derivado de dados em uma porção do código onde são manipuladas variáveis que operam com datas, horas, minutos, etc. O tipo derivado de dados gerado foi nomeado como *date_struct* (para corresponder semanticamente às variáveis contidas em seu interior) e uma instância do mesmo foi nomeada como *struct*, como pode ser observado na Figura 5.6.

Nas regiões do código onde as variáveis transformadas em tipo derivado de dados eram usadas, as referências das mesmas foram substituídas pela referência da instância *struct*, sendo que cada variável é acessada pelo modificador de acesso usado nos tipos derivados de dados (%), como pode ser observado na Figura 5.7.

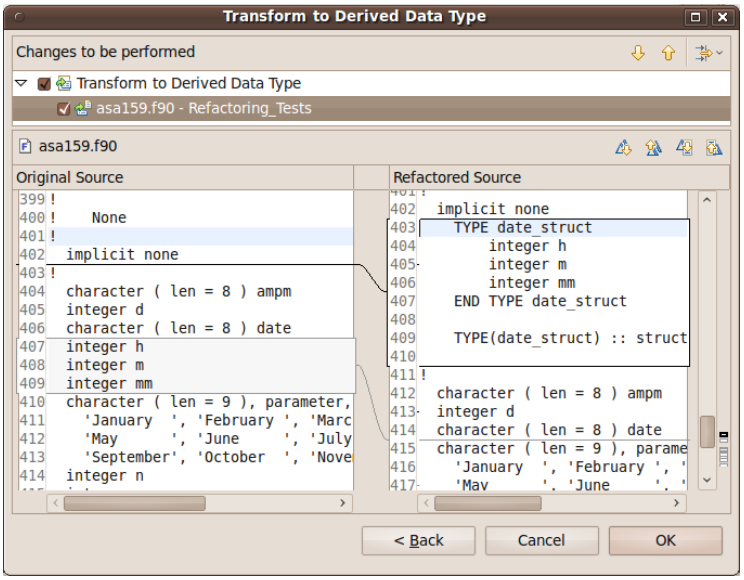


Figura 5.6: Criando o tipo *date_struct* e sua instância (*struct*)

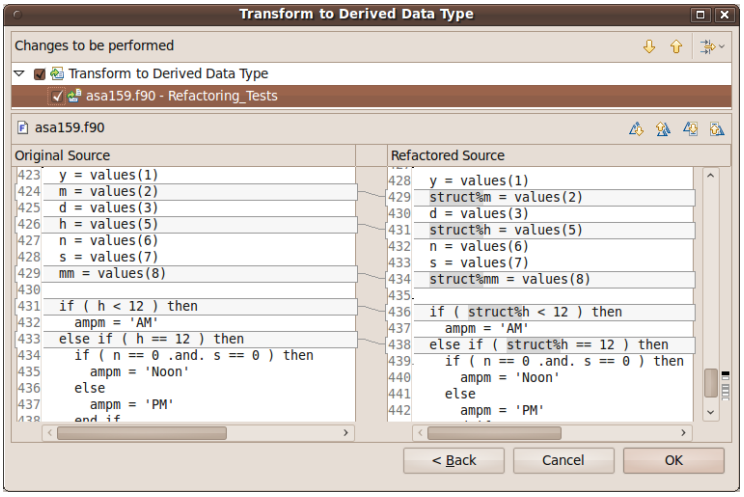


Figura 5.7: Substituindo referências das variáveis selecionadas pela instância *struct*

Após a aplicação da refatoração *Transform to Derived Data Type*, foram identificadas outras variáveis que poderiam ser adicionadas ao tipo *date_struct*. Para adicionar tais variáveis ao tipo derivado de dados *date_struct* foi usada a refatoração *Add Variable to Derived Data Type*, como mostram as Figuras 5.8 e 5.9.

Nas regiões do código onde as variáveis adicionadas ao tipo *date_struct* eram usadas, as referências às mesmas foram substituídas pela referência à instância do tipo derivado (*struct*), como pode ser obervado na Figura 5.10.

Com a aplicação das refatorações *Transform to Derived Data Type* e *Add Variable to Derived Data Type*, foi possível gerar um novo tipo derivado de dados no código-fonte da aplicação (*date_struct*), agrupando variáveis que atuam sobre atributos semelhantes

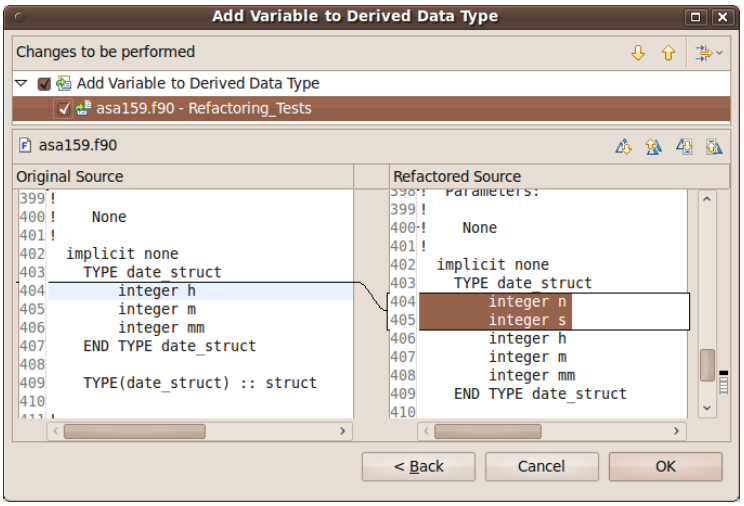


Figura 5.8: Adicionando variáveis *n* e *s* no tipo *date_struct*

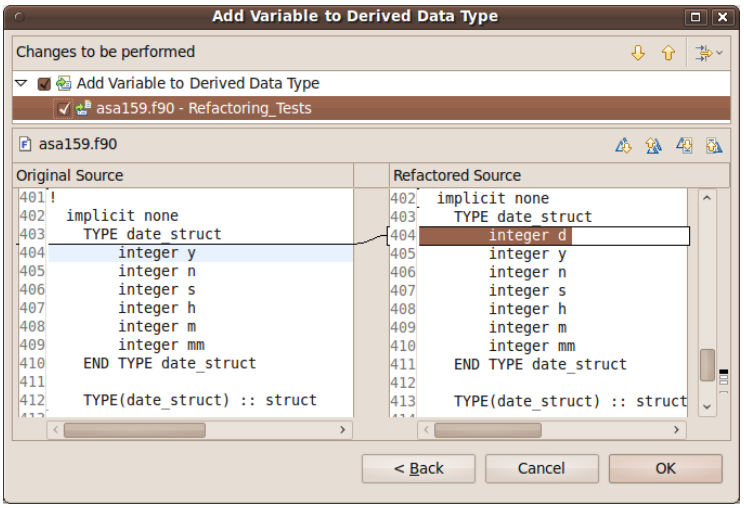


Figura 5.9: Adicionando a variável *d* no tipo *date_struct*

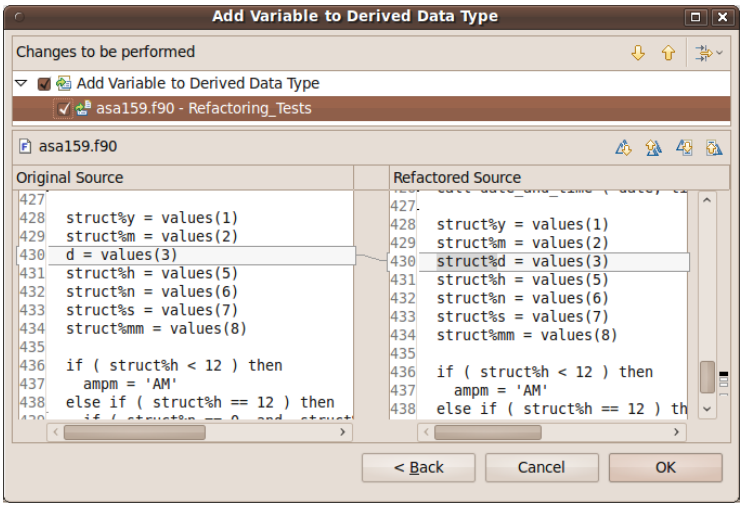


Figura 5.10: Substituindo referências da variável *d* pela instância *struct*

do aplicativo, facilitando sua manipulação e sua alteração. Com essas refatorações o desempenho de execução das aplicações não é afetado, uma vez que apenas a sintaxe do código é alterada, e o fluxo de execução permanece o mesmo. Assim, pode-se concluir que as duas refatorações cumprem seus objetivos.

5.1.3 Converter *IF-THEN-ELSE* Aninhados em *SELECT CASE*

Para avaliar a refatoração *Nested If-Then-Else to Select Case* também foi usado o aplicativo SUPIM (visto na seção 5.1.1). Com a aplicação da refatoração no código do programa, foram identificadas 8 regiões em que se usavam comandos *IF-THEN-ELSE* aninhados para avaliar a igualdade de valor de uma única variável. As Figuras 5.11 e 5.12 mostram duas regiões onde a refatoração encontrou ocorrências de comandos *IF-THEN-ELSE* aninhados e efetuou a substituição pelo comando *SELECT CASE*.

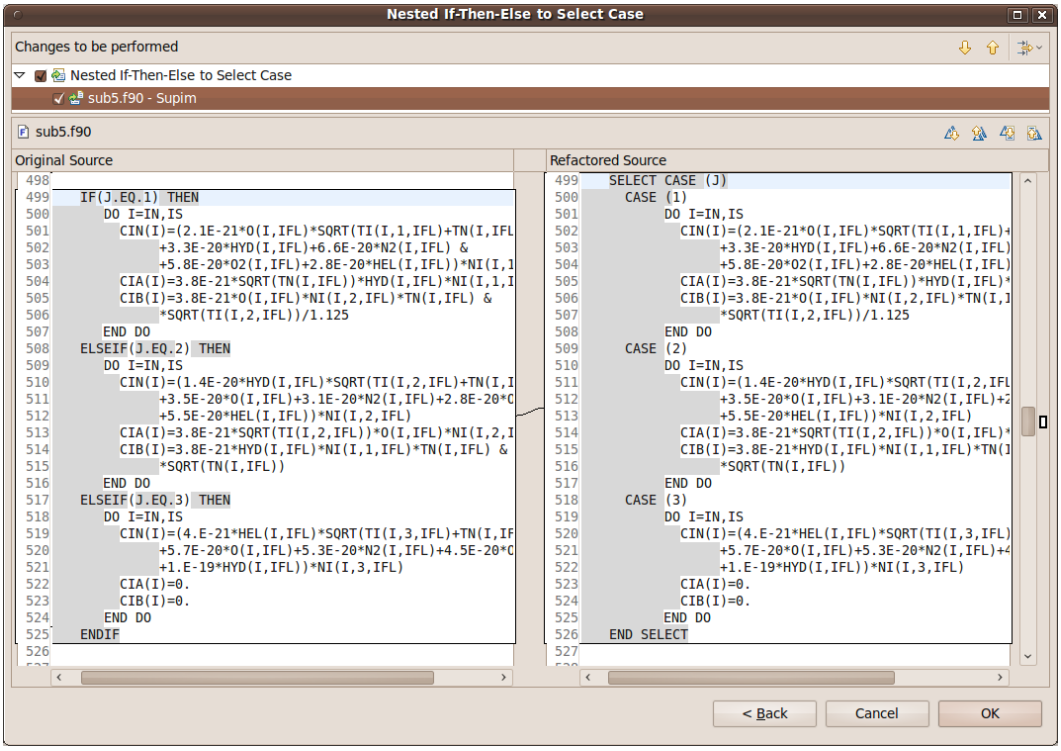


Figura 5.11: Substituindo a comparação da variável *J* por *SELECT CASE*

Com o uso dessa refatoração o código resultante ficou mais legível, sendo usada a construção mais adequada para a comparação de igualdade de valores de uma única variável. O resultado obtido é a evolução da construção usada para esse tipo de comparação do padrão Fortran 77 para o padrão correspondente do Fortran 90. Essa refatoração também não afeta o desempenho da aplicação, uma vez que apenas introduz diferenças sintáticas

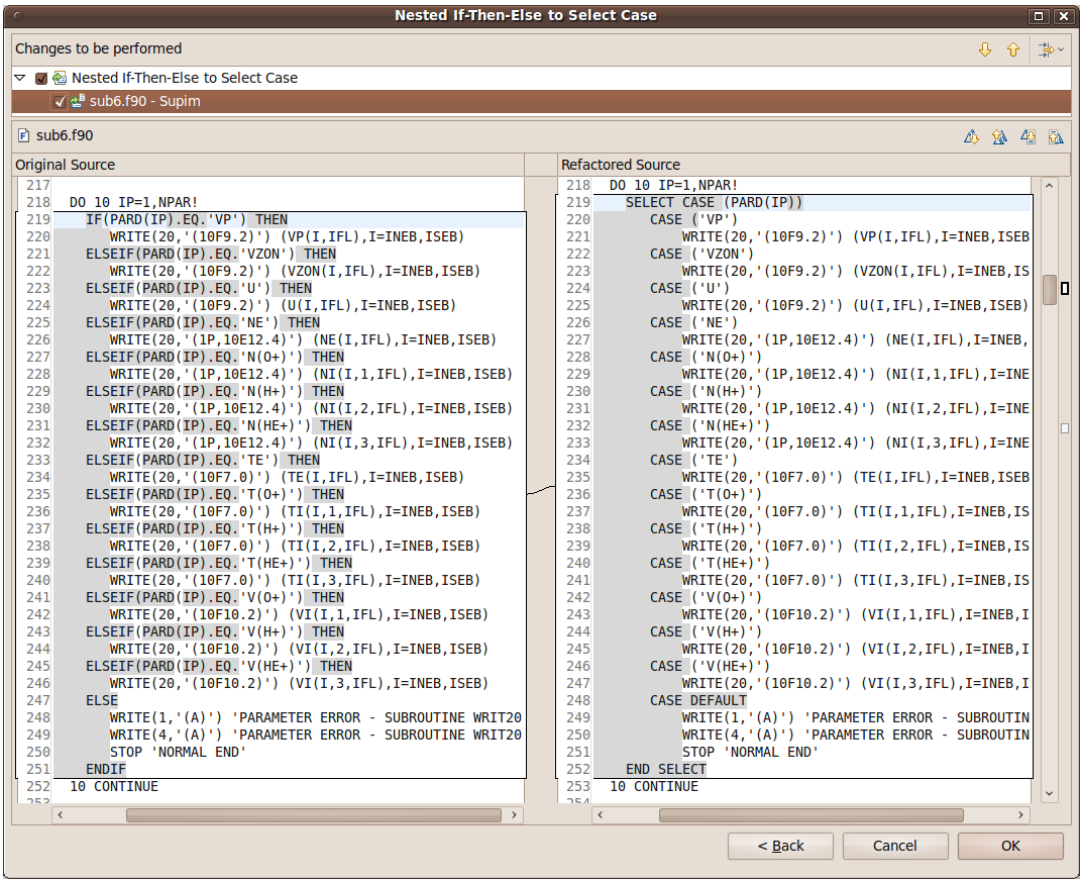


Figura 5.12: Substituindo a comparação da variável *PARD(IP)* por *SELECT CASE*

no código-fonte, adequando a construção da comparação ao padrão do Fortran 90, não modificando o fluxo de execução do código.

5.1.4 Converter laços *DO* em construções *FORALL*

Para avaliar a refatoração *Replace Do Loop by Forall*, foi gerada a aplicação MULT-MAT, que é um pequeno código que faz a multiplicação de duas matrizes (MA = MB*MC). Esse tipo de multiplicação é frequentemente usada em aplicações científicas. O código gerado é simples, mas demanda um grande processamento computacional para realizar a multiplicação de duas matrizes de tamanho 1000 x 1000. Com a construção *FORALL* é possível paralelizar a execução de um dos laços de repetição do código, podendo obter-se ganho de desempenho na execução do mesmo. O código-fonte da aplicação pode ser observado na Figura 5.13.

Analisando o código, observou-se que o terceiro laço *DO* poderia ser substituído pela construção *FORALL*, uma vez que as regras dos índices do *FORALL* são satisfeitas para esse laço. Com essa substituição, a execução do laço poderia ser paralelizada, havendo

```
PROGRAM multmat
   INTEGER, PARAMETER :: n = 1000
   REAL :: a(n,n), b(n,n), c(n,n)
   INTEGER :: i, j, k
   DO i=1, n
      DO j=1, n
         b(i,j) = i+j
         c(i,j) = i*j
      END DO
   END DO
   ! Realiza a multiplicação
   DO i = 1, n
      DO j = 1, n
         DO k = 1, n
            a(i,k) = a(i,k) + b(i,j) * c(j,k)
         END DO
      END DO
   END DO
END PROGRAM multmat
```

Figura 5.13: Código para multiplicar duas matrizes

chances de aumentar o desempenho de execução da aplicação. Observe que não seria possível substituir os três laços *DO* por *FORALL*, pois no lado esquerdo da atribuição da matriz *a* aparecem apenas os índices *i* e *k*. Caso na atribuição fossem usados os índices *j* e *k*, por exemplo, o segundo e terceiro laços *DO* poderiam ser substituídos por *FORALL* (pois estariam satisfeitas as condições dos índices para o uso da construção *FORALL*). A Figura 5.14 mostra o uso da refatoração *Replace Do Loop by Forall* na aplicação.

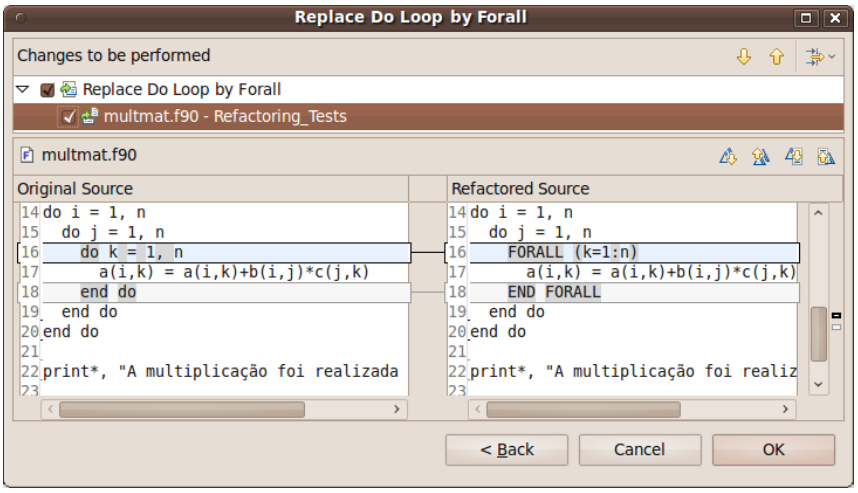


Figura 5.14: Substituindo um laço *DO* pela construção *FORALL*

Após compilar e executar o código-fonte refatorado, observou-se que os resultados obtidos na multiplicação das duas matrizes foram idênticos aos resultados obtidos antes da refatoração. Também foram realizados experimentos para avaliar o impacto da refatoração no desempenho da aplicação. Cada versão do código (original e refatorado)

Tempos de Execução Aplicação Original	**Tempos de Execução Aplicação Refatorada**
17,048s	14,505s
16,997s	14,457s
16,877s	14,482s
17,025s	14,431s
17,084s	14,483s
Tempo médio:	**Tempo médio:**
17,006s	14,472s
	Diferença do original:
	2,534s

Tabela 5.1: Tempos de execução da aplicação original e da aplicação refatorada

foi executada cinco vezes, tendo seu tempo de execução medido, para obter a média de tempos de execução de cada versão da aplicação, como pode ser observado na Tabela 5.1.

Com base nos dados apresentados na Tabela 5.1, pode-se constatar que houve melhora no desempenho de execução da aplicação refatorada. O *speedup* em um processador com 2 núcleos foi de 1,18 (59,00% de eficiência). O ganho de desempenho não foi muito grande, mas o resultado serve para exemplificar a possibilidade de aumento de desempenho em aplicações que usam a construção *FORALL* para a manipulação de matrizes.

5.2 Discussão

Com a aplicação das refatorações nos aplicativos vistos anteriormente, foram obtidos códigos mais bem estruturados, e com padrões mais recentes da linguagem de programação, e todos eles permaneceram funcionando corretamente após serem refatorados.

Em sua maioria, as refatorações implementadas não afetam o desempenho das aplicações, pois atuam apenas com mudanças na sintaxe do código-fonte. Apenas a refatoração *Replace Do Loop by Forall* é capaz de alterar o desempenho de um aplicativo, e o experimento realizado com ela mostrou uma melhora no desempenho da aplicação refatorada.

Também é preciso registrar que refatorações automatizadas agilizam e dão maior segurança ao programador. O uso da ferramenta Photran agrega a funcionalidade do programador pré-visualizar o código refatorado e decidir ou não por sua utilização, facilitando a atividade de programação e de refatoração de software.

6 CONCLUSÃO

Nesta dissertação, foi explorada a utilização de técnicas de refatoração sobre aplicações escritas em linguagem Fortran, objetivando melhorar o projeto de código, assim como a legibilidade do mesmo, através da evolução dos padrões da linguagem de programação utilizados em aplicações legadas. Para isso, foi proposto um catálogo de refatorações para a evolução de programas em linguagem Fortran.

O catálogo de refatorações proposto neste trabalho conta com um conjunto de dez refatorações para a evolução de programas escritos em Fortran. Um subconjunto de seis refatorações contidas no catálogo foi automatizado e integrado à ferramenta Photran, um *plugin* do Eclipse que oferece recursos para o ciclo de desenvolvimento de aplicações Fortran e que oferece um *framework* que permite estender funcionalidades de refatoração para essa linguagem de programação. A utilização do suporte desse tipo de ferramenta reduz o risco de erros e inconsistências, além de reduzir também o trabalho e o custo de desenvolvimento e manutenção de software. No Photran, as refatorações automatizadas foram nomeadas como: *Extract Subroutine or Function to Module*, *Move Subroutine or Function to Module*, *Transform to Derived Data Type*, *Add Variable to Derived Data Type*, *Nested If-Then-Else to Select Case* e *Replace Do Loop by Forall*.

A contribuição científica deste trabalho consiste na geração de um catálogo de refatorações que permitem migrar construções de uma versão de Fortran para outra. Cada refatoração do catálogo conta com uma lista de passos bem definidos para sua aplicação, possibilitando que programadores possam usá-las para melhorar o projeto de códigos existentes. As seis refatorações implementadas no Photran já foram enviadas ao comitê de avaliação da ferramenta para que sejam adicionadas em sua versão oficial. Até o momento da defesa deste trabalho, as refatorações ainda estavam em fase de avaliação para serem aceitas pelo comitê.

As refatorações automatizadas no Photran foram validadas em aplicações escritas em Fortran, tendo resultados satisfatórios. Com a aplicação das refatorações *Extract Subroutine or Function to Module*, *Move Subroutine or Function to Module*, *Transform to Derived Data Type*, *Add Variable to Derived Data Type* e *Nested If-Then-Else to Select Case* foram obtidos códigos mais legíveis, adequando-os aos novos padrões da linguagem Fortran. Com a aplicação da refatoração *Replace Do Loop by Forall*, além de se obter um código com um padrão mais atual da linguagem, ganhou-se desempenho na execução da aplicação refatorada, pois com o uso de construções *FORALL*, é possível paralelizar os laços que envolvem operações de vetores e matrizes quando a arquitetura utilizada é multiprocessada.

Como trabalhos futuros, pretende-se automatizar a refatoração *Converter laços DO em laços DO CONCURRENT* assim que estiver disponível o reconhecimento da sintaxe do comando *DO CONCURRENT* no Photran, e enviá-la ao comitê de avaliação da ferramenta para que seja adicionada em sua versão oficial.

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. New York: Springer Publishing Company, Inc., 2008.

ALI, S.; MAQBOOL, O. Monitoring software evolution using multiple types of changes. **International Conference on Emerging Technologies**, Islamabad, Pakistan, p.410–415, 2009.

ANSI. **Fortran 77 Standard – ISO/IEC ISO 1539**:1980, information technology - programming languages. Disponível em: `http://www.fortran.com/fortran/F77_std/rjcnf.html`. Acesso em: abril de 2011.

ANSI. **Fortran 90 Standard – ISO/IEC 1539**:1991, information technology - programming languages. Disponível em: `ftp://ftp.nag.co.uk/sc22wg5/N001-N1100/N692.pdf`. Acesso em: abril de 2011.

ANSI. **Fortran 95 Standard – ISO/IEC 1539-1**:1997, information technology - programming languages. Disponível em: `http://j3-fortran.org/doc/standing/archive/007/97-007r2/pdf/97-007r2.pdf`. Acesso em: abril de 2011.

ANSI. **Fortran 2003 Standard – ISO/IEC 1539-1**:2004(e), information technology - programming languages. Disponível em: `http://www.j3-fortran.org/doc/year/04/04-007.pdf`. Acesso em: abril de 2011.

ANSI. **Fortran 2008 Standard – ISO/IEC FDIS 1539-1**:2010, information technology - programming languages. Disponível em: `ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1830.pdf`. Acesso em: abril de 2011.

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

ASHBY, J. V.; REID, J. K. **Migrating a Scientific Application from MPI to Coarrays**. Harwell Oxford: STFC Rutherford Appleton Laboratory, 2008.

ASTELS, D. Refactoring with UML. In: THIRD INTERNATIONAL CONFERENCE ON EXTREME PROGRAMMING AND FLEXIBLE PROCESSES IN SOFTWARE ENGINEERING, 2002, Alghero, Italia. **Anais...** Springer-Verlag, 2002. p.67–70.

BACKUS, J. The history of FORTRAN I, II, and III. **SIGPLAN Not.**, New York, NY, USA, v.13, p.165–180, August 1978.

BAILEY, G. **SUPIM - Sheffield University Plasmasphere Ionosphere Model**. Disponível em: `http://gbailey.staff.shef.ac.uk/supim.html`. Acesso em: abril de 2011.

BEATON, W.; RIVIERES, J. des. **Eclipse Platform Technical Overview**. Ottawa, Ontario, Canada: 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.

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

BONIATI, B. B.; CHARÃO, A. S.; OLIVEIRA STEIN, B. de; RISSETTI, G.; PIVETA, E. K. Automated refactorings for high performance Fortran programmes. **Int. J. High Performance Systems Architecture**, Geneva, v.3, p.98–109, 2011.

BONIATI, B. B.; RISSETTI, G.; CHARÃO, A. S.; PIVETA, E. K. Extensões para Refatoração de Código Fortran no Eclipse. In: FÓRUM INTERNACIONAL DE SOFTWARE LIVRE – XI WORKSHOP SOBRE SOFTWARE LIVRE, 11., 2010, Porto Alegre, Brasil. **Anais...** Consórcio Editorial Livre, 2010. p.74–79.

BURKARDT, J. **ASA159 - Random Generation of a Table**. Disponível em: `http://orion.math.iastate.edu/burkardt/f_src/asa159/asa159.html`. Acesso em: abril de 2011.

CHEN, N.; OVERBEY, J. **Photran 7.0 Developer's Guide**. [S.l.: s.n.], 2010.

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

DE, V. **A Foundation for Refactoring Fortran 90 in Eclipse**. 2004. Dissertação (Mestrado) — University of Illinois, Urbana-Champaign, EUA.

DI PENTA, M.; NETELER, M.; ANTONIOL, G.; MERLO, E. A language-independent software renovation framework. **J. Syst. Softw.**, New York, NY, USA, v.77, p.225–240, September 2005.

DRAGAN-CHIRILA, J. **Integrating a Fortran 90 Parser into an Eclipse Environment**. 2004. Dissertação (Mestrado) — University of Illinois, Urbana-Champaign, EUA.

ECLIPSE.ORG. **Photran - An Integrated Development Environment for Fortran**. Disponível em: `http://www.eclipse.org/photran/`. Acesso em: abril de 2011.

ECLIPSE.ORG. **Eclipse C/C++ Development Tooling - CDT**. Disponível em: `http://www.eclipse.org/cdt/`. Acesso em: abril de 2011.

EIPE, R. M. **Extending Eclipse to create an IDE plugin for a new language with FORTRAN as a case study**. 2004. Dissertação (Mestrado) — University of Illinois, Urbana-Champaign, EUA.

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

FOWLER, M.; BECK, K.; BRANT, J.; OPDYKE, W.; ROBERTS, D. **Refactoring**: improving the design of existing code. Boston, Massachusetts: 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.

GODFREY, M. W.; GERMAN, D. M. The past, present, and future of software evolution. **Frontiers of Software Maintenance**, Waterloo, ON, p.129–138, 2008.

GREENFIELD, M. N. History of FORTRAN standardization. In: JUNE 7-10, 1982, NATIONAL COMPUTER CONFERENCE, 1982, New York, NY, USA. **Proceedings...** ACM, 1982. p.817–824. (AFIPS '82).

GRISWOLD, W. G.; NOTKIN, D. Automated Assistance for Program Restructuring. **ACM Transactions on Software Engineering and Methodology**, New York, NY, USA, v.2, n.3, p.228–269, 1993.

JONAS, J. S. **Fortran Statement Card**. Disponível em: `http://ferretronix.com/march/computer_cards/4tran_statement_thumb.jpg`. Acesso em: abril de 2011.

JONES, J. Abstract Syntax Tree Implementation Idioms. In: CONFERENCE ON PATTERN LANGUAGES OF PROGRAMS (PLOP2003), 10., 2003, Allerton Park in Monticello, IL. **Proceedings...** The Hillside Group: Inc., 2003.

KERIEVSKY, J. **Refactoring to Patterns**. Boston, Massachusetts: Addison-Wesley Professional, 2004.

KOELBEL, C. H.; ZOSEL, M. E. **The High Performance FORTRAN Handbook**. Cambridge, MA, USA: MIT Press, 1993.

KOFFMAN, E. B.; FRIEDMAN, F. L. **FORTRAN**. Boston, Massachusetts: Addison-Wesley, 1996.

LANO, K.; MALIK, N. Mapping Procedural Patterns to Object-Oriented Design Patterns. **Automated Software Engg.**, Hingham, MA, USA, v.6, p.265–289, July 1999.

LEHMAN, M. M.; RAMIL, J. F.; WERNICK, P. D.; PERRY, D. E.; TURSKI, W. M. Metrics and Laws of Software Evolution - The Nineties View. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE METRICS, 4., 1997, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 1997. p.20–.

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.

MARTICORENA, R. **Analysis and definition of a language independent refactoring catalog**. Porto, Portugal: In 17th Conference on Advanced Information Systems Engineering (CAiSE 05). Doctoral Consortium, 2005.

MELTON, H.; TEMPERO, E. Identifying refactoring opportunities by identifying dependency cycles. In: AUSTRALASIAN COMPUTER SCIENCE CONFERENCE - VOLUME 48, 29., 2006, Darlinghurst, Australia, Australia. **Proceedings...** Australian Computer Society: Inc., 2006. p.35–41. (ACSC '06).

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

MILLER, A. **Convert f77 to f90 (to_f90.f90)**. Disponível em: `http://jblevins.org/mirror/amiller/to_f90.f90`. Acesso em: abril de 2011.

MPI FORUM, C. MPI: a message passing interface. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 1993., 1993, New York, NY, USA. **Proceedings...** ACM, 1993. p.878–883. (Supercomputing '93).

MÉNDEZ, M.; OVERBEY, J.; GARRIDO, A.; TINETTI, F.; JOHNSON, R. A Catalog and Classification of Fortran Refactorings. In: ARGENTINE SYMPOSIUM ON SOFT-

WARE ENGINEERING (ASSE 2010), 11., 2010, Buenos Aires, Argentina. **Anais...** SADIO Sociedad Argentina de Informática, 2010.

MÉNDEZ, M.; OVERBEY, J.; GARRIDO, A.; TINETTI, F.; JOHNSON, R. Refactorización en Código Fortran Heredado (In Spanish). In: XVI CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN (CACIC 2010), 2010, Buenos Aires, Argentina. **Anais...** CACIC, 2010.

NYHOFF, L.; LEESTMA, S. **Fortran 90 for Engineers and Scientists**. New Jersey, USA: Prentice-Hall, 1997.

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

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. L. **Virtual Program Graph**. Disponível em: `http://jeff.over.bz/software/vpg/doc/`. Acesso em: abril de 2011.

OVERBEY, J. L.; NEGARA, S.; JOHNSON, R. E. Refactoring and the evolution of Fortran. In: ICSE WORKSHOP ON SOFTWARE ENGINEERING FOR COMPUTATIONAL SCIENCE AND ENGINEERING, 2009., 2009, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2009. p.28–34. (SECSE '09).

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...** ACM, 2005.

PATEFIELD, W. M. Algorithm AS 159: an efficient method of generating rxc tables with given row and column totals. **Applied Statistics**, Oxford, UK, v.30, n.1, p.91–97, 1981.

PGI, T. P. G. **PGI Fortran Workstation**. Disponível em: `http://www.pgroup.com/products/pgiworkstation.htm`. Acesso em: maio de 2011.

PIDAPARTHI, S.; LUKER, P.; ZEDAN, H. **Reengineering procedural software to object-oriented software using design transformations and resource usage matrix**. Chichester, USA: Horwood Publishing, Ltd., 1999. p.182–197.

PIVETA, E.; ARAÚJO, J.; PIMENTA, M.; MOREIRA, A.; GUERREIRO, P.; PRICE, R. T. Searching for Opportunities of Refactoring Sequences: reducing the search space. In: ANNUAL IEEE INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, 2008., 2008, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2008. p.319–326.

PIVETA, E. K. **Improving the search for refactoring opportunities on object-oriented and aspect-oriented software**. 2009. Tese de Doutorado — Universidade Federal do Rio Grande do Sul, Porto Alegre - Rio Grande do Sul, Brasil.

PIVETA, E.; PIMENTA, M.; ARAÚJO, J.; MOREIRA, A.; GUERREIRO, P.; PRICE, R. T. Representing refactoring opportunities. In: ACM SYMPOSIUM ON APPLIED COMPUTING, 2009., 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.1867–1872. (SAC '09).

PIZKA, M.; JUERGENS, E. Automating Language Evolution. In: IN PROCEEDINGS OF FIRST JOINT IEEE/IFIP SYMPOSIUM ON THEORETICAL ASPECTS OF SOFTWARE ENGINEERING, 2007. **Anais...** IEEE Computer Society Press, 2007. p.305–315.

POLYHEDRON. **plusFORT**. Disponível em: `http://www.polyhedron.com/pf-plusfort0html`. Acesso em: maio de 2011.

PRESS, W. H.; TEUKOLSKY, S. A.; FLANNERY, B. P.; VETTERLING, W. T. **Numerical Recipes in FORTRAN**: the art of scientific computing. 2nd.ed. New York, NY, USA: Cambridge University Press, 1992.

QUINLAN, D.; LIAO, C.; PANAS, T.; MATZKE, R.; SCHORDAN, M.; VUDUC, R.; YI, Q. **ROSE Compiler Infrastructure**. Disponível em: `http://www.rosecompiler.org/`. Acesso em: maio de 2011.

RAMSDEN, S.; LIN, F. **Fortran 90 - A Conversion Course for Fortran 77 Programmers**. Manchester, UK: Manchester Computing Centre, University of Manchester, 1995.

RATZINGER, J.; SIGMUND, T.; VORBURGER, P.; GALL, H. Mining Software Evolution to Predict Refactoring. **Empirical Software Engineering and Measurement, International Symposium on**, Los Alamitos, CA, USA, v.0, p.354–363, 2007.

REID, J. The new features of Fortran 2003. **SIGPLAN Fortran Forum**, New York, NY, USA, v.26, p.10–33, April 2007.

REID, J. The new features of Fortran 2008. **SIGPLAN Fortran Forum**, New York, NY, USA, v.27, p.8–21, August 2008.

RISSETTI, G.; CHARÃO, A. S.; BONIATI, B. B. Incorporação de Novas Refatorações para Linguagem Fortran no IDE Eclipse. In: X ESCOLA REGIONAL DE ALTO DESEMPENHO, 2010, Passo Fundo, Brasil. **Anais...** SBC, 2010. p.233–236.

ROBBES, R.; LANZA, M.; LUNGU, M. An approach to software evolution based on semantic change. In: FUNDAMENTAL APPROACHES TO SOFTWARE ENGINEERING, 10., 2007, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2007. p.27–41. (FASE'07).

ROYCHOUDHURY, S. A language-independent approach to software maintenance using grammar adapters. In: COMPANION TO THE 19TH ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS, 2004, New York, NY, USA. **Anais...** ACM, 2004. p.314–315. (OOPSLA '04).

SEREBRENIK, A.; SCHRIJVERS, T.; DEMOEN, B. Improving prolog programs: refactoring for prolog. **Theory Pract. Log. Program.**, New York, v.8, p.201–215, 2008.

SOUZA, J. R. de. **Modelagem Ionosférica em baixas latitudes no Brasil**. 1997. Tese de Doutorado — Instituto Nacional de Pesquisas Espaciais.

TICHELAAR, S.; DUCASSE, S.; DEMEYER, S.; NIERSTRASZ, O. A Meta-Model for Language-Independent Refactoring. In: INTERNATIONAL SYMPOSIUM ON PRINCIPLES OF SOFTWARE EVOLUTION, 2000, Kanazawa, Japan. **Anais...** IEEE Computer Society, 2000. p.154–164.

TIETZMANN, D.; RISSETTI, G.; CHARÃO, A. S.; PIVETA, E. K.; PETRY, A.; SOUZA, J. Refatorações para Melhoria da Legibilidade de Código Fortran. In: FÓRUM

INTERNACIONAL DE SOFTWARE LIVRE – XII WORKSHOP SOBRE SOFTWARE LIVRE, 12., 2011, Porto Alegre, Brasil. **Anais...** Consórcio Editorial Livre, 2011. p.–.

TOURWÉ, T.; MENS, T. Identifying Refactoring Opportunities Using Logic Meta Programming. In: SEVENTH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, 2003, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2003. p.91–.

TOURWÉ, T.; MENS, T. A Survey of Software Refactoring. **IEEE Transactions on Software Engineering**, Los Alamitos, CA, USA, v.30, n.2, p.126–139, 2004.

WLOKA, J.; HIRSCHFELD, R.; HÄNSEL, J. Tool-supported refactoring of aspect-oriented programs. In: ASPECT-ORIENTED SOFTWARE DEVELOPMENT, 7., 2008, New York, NY, USA. **Proceedings...** ACM, 2008. p.132–143. (AOSD '08).

XU, L.; BUTLER, G. Cascaded Refactoring for Framework Development and Evolution. **Software Engineering Conference, Australian**, Los Alamitos, CA, USA, v.0, p.319–330, 2006.

YOKOMORI, R.; SIY, H.; YOSHIDA, N.; NORO, M.; INOUE, K. Measuring the effects of aspect-oriented refactoring on component relationships: two case studies. In: ASPECT-ORIENTED SOFTWARE DEVELOPMENT, 2011, New York, NY, USA. **Proceedings...** ACM, 2011. p.215–226. (AOSD '11).

ZHAO, Q.; WANG, H.; FENG, G.; LU, X. Software Evolution Method Considering Software Historical Behavior. **International Conference on Internet Computing in Science and Engineering**, Los Alamitos, CA, USA, v.0, p.36–41, 2009.

ZOU, Y.; KONTOGIANNIS, K. A Framework for Migrating Procedural Code to Object-Oriented Platforms. In: EIGHTH ASIA-PACIFIC ON SOFTWARE ENGINEERING CONFERENCE, 2001, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2001. p.390–. (APSEC '01).

ZOU, Y.; KONTOGIANNIS, K. Incremental Transformation of Procedural Systems to Object Oriented Platforms. In: ANNUAL INTERNATIONAL CONFERENCE ON COMPUTER SOFTWARE AND APPLICATIONS, 27., 2003, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2003. p.290–. (COMPSAC '03).

APÊNDICE A ÁRVORES DE CHAMADAS

Durante o desenvolvimento deste trabalho foi proposta e implementada no Photran uma refatoração para introduzir árvores de chamadas para cada sub-rotina do código-fonte das aplicações, sendo nomeada no Photran como *Introduce Call Tree*. Essa refatoração também já foi submetida ao comitê de avaliação do Photran para ser disponibilizada em sua versão oficial, e foi descrita em um trabalho publicado no WSL 2011 (Workshop Sobre Software Livre), evento integrado ao FISL 2011 (Fórum Internacional de Software Livre) (TIETZMANN et al., 2011).

A.1 *Introduce Call Tree*

Você possui um código com diversas sub-rotinas espalhadas, e deseja compreender melhor sua lógica de execução.

Introduza árvores de chamadas em cada sub-rotina do código-fonte, e saiba onde cada sub-rotina é usada.

A.1.1 Motivação

Chamadas a sub-rotinas são frequentes no código-fonte de programas em Fortran. Quando surge a necessidade de fazer alguma alteração na aplicação, o fato de se ter diversas chamadas de sub-rotinas espalhadas pelo código-fonte pode causar dúvidas e complicar o entendimento da lógica do programa. A introdução da árvore de chamadas nos comentários de cada sub-rotina permite saber antecipadamente quais são as invocações efetuadas em seu escopo.

A.1.2 Mecânica

A mecânica desta refatoração consiste em percorrer a AST do código-fonte em busca de chamadas a sub-rotinas. Quando uma chamada é encontrada, é verificado o nome

da sub-rotina chamada e o número da linha onde a mesma ocorre. Após coletar esses dados, eles são inseridos na forma de um comentário Fortran no início de cada escopo do código. Esta refatoração percorre todos os escopos do código-fonte e adiciona árvores de chamadas em cada um deles.

A.1.3 Avaliação

Para avaliar e validar o funcionamento dessa refatoração, ela foi aplicada no código SUPIM, usado na avaliação e validação de outras refatorações implementadas neste trabalho. A Figura A.1 mostra a introdução de uma árvore de chamadas na sub-rotina *apex* do código SUPIM, utilizando-se o Photran.

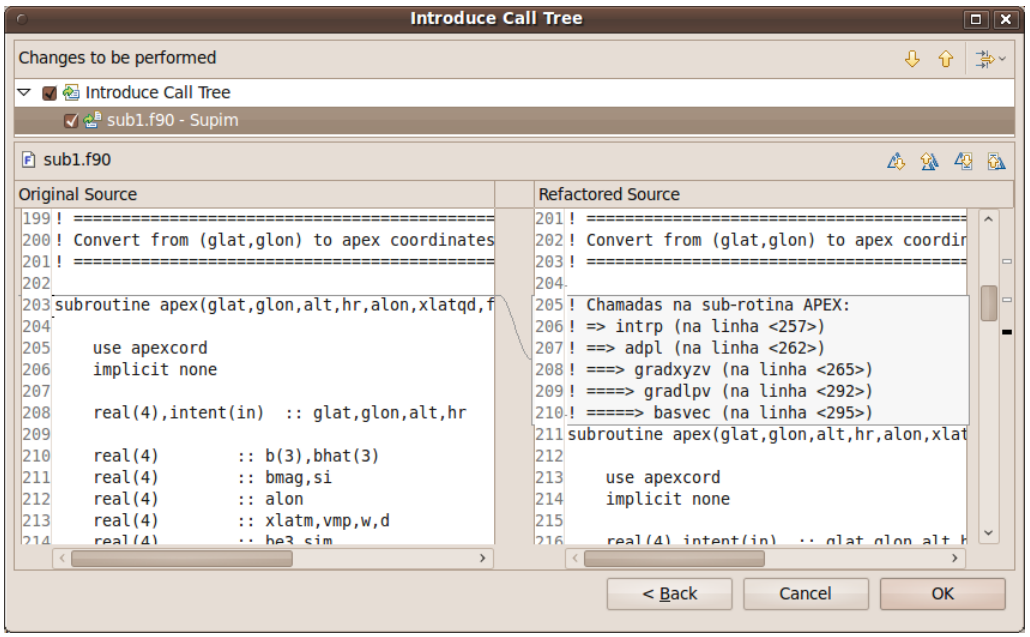


Figura A.1: Introduzindo árvore de chamadas na sub-rotina *apex*

Com a aplicação dessa refatoração no código SUPIM, foram identificadas 104 chamadas para sub-rotinas em todo o código-fonte. Foram adicionadas informações sobre as chamadas em todos os escopos onde as mesmas ocorreram, informando o nome da sub-rotina usada e a linha onde o uso ocorreu.

Assim, pode-se concluir que com o uso dessa refatoração, o código resultante apresenta novos comentários que facilitam o entendimento de sua lógica, sendo úteis em suas futuras rotinas de manutenção.

APÊNDICE B PUBLICAÇÕES

Durante o desenvolvimento deste trabalho foi publicado um artigo na ERAD 2010 (Escola Regional de Alto Desempenho), um artigo no WSL 2010 (Workshop Sobre Software Livre), evento integrado ao FISL 2010 (Fórum Internacional de Software Livre), e um artigo no WSL 2011 (Workshop Sobre Software Livre), evento integrado ao FISL 2011 (Fórum Internacional de Software Livre). Também foi publicado um artigo no *International Journal of High Performance Systems Architecture* (Qualis B1), apresentando conceitos, implementações e avaliações de refatorações automatizadas para Fortran com o uso do *plugin* Photran.

- Gustavo Rissetti; Andrea S. Charão; Bruno B. Boniati. **Incorporação de Novas Refatorações para Linguagem Fortran no IDE Eclipse**, X Escola Regional de Alto Desempenho (ERAD), 2010, Passo Fundo - RS. p. 233–236.

- Bruno B. Boniati; Gustavo Rissetti; Andrea S. Charão; Eduardo K. Piveta. **Extensões para Refatoração de Código Fortran no Eclipse**, 11º Fórum Internacional de Software Livre (FISL), 2010, Porto Alegre - RS. XI Workshop Sobre Software Livre. Porto Alegre - RS. p. 74–79.

- Dionatan Tietzmann; Gustavo Rissetti; Andrea S. Charão; Eduardo K. Piveta; Adriano Petry; Jonas Souza. **Refatorações para Melhoria da Legibilidade de Código Fortran**, 12º Fórum Internacional de Software Livre (FISL), 2011, Porto Alegre - RS. XII Workshop Sobre Software Livre. Porto Alegre - RS

- Bruno B. Boniati; Andrea S. Charão; Benhur de Oliveira Stein; Gustavo Rissetti; Eduardo K. Piveta. ***Automated Refactorings for High Performance Fortran Programs***, International Journal of High Performance Systems Architecture, v. 3, p. 98–109, Inderscience Publishers, Geneva, 2011.