ISSN 1809-5860

PROGRAMAÇÃO ORIENTADA A OBJETOS EM FORTRAN

André Teófilo Beck[1] & Felipe Alexander Vargas Bazán[2]

Resumo

Este artigo apresenta conceitos fundamentais de programação orientada a objetos (OO) em FORTRAN. Em geral, os usuários de FORTRAN não estão familiarizados com estes conceitos, pois os compiladores desta linguagem não possuíam suporte para programação OO até o recente lançamento da versão 11.1 do compilador Intel Visual FORTRAN. Este compilador suporta a maioria das características de orientação a objetos do padrão FORTRAN 2003, permitindo a atualização de práticas de programação com esta importante metodologia. O principal objetivo do presente artigo é mostrar que o FORTRAN pode ser utilizado em um nível de abstração muito maior do que se constata na prática (em particular, na engenharia), utilizando conceitos de programação OO. O artigo apresenta o estado da arte da programação OO em padrões e compiladores de FORTRAN e discute conceitos de abstração de dados, encapsulamento e proteção de informação, classes e objetos. Os conceitos são apresentados independentemente de linguagem de programação, mas a implementação dos mesmos é ilustrada no FORTRAN 90/95/2003. A construção de classes polimórficas, por extensão de tipo e por hereditariedade, é ilustrada utilizando o novo compilador da Intel. Adicionalmente, o artigo mostra que é possível a emulação de polimorfismo em compiladores mais antigos e no FORTRAN 90/95, através do uso apropriado de ponteiros. Os conceitos são ilustrados em um exemplo acadêmico e didático, envolvendo um sistema de gerenciamento universitário, que manipula pessoas, alunos, professores, disciplinas e datas.

Palavras-chave: Programação de computadores. Programação orientada a objetos. Programação OO. FORTRAN.

OBJECT ORIENTED PROGRAMMING IN FORTRAN

Abstract

This paper presents fundamental concepts for object oriented programming (OOP) in FORTRAN. In general, FORTRAN users are not familiar with these concepts since FORTRAN compilers had no support for OOP until the recent release of Intel Visual FORTRAN compiler version 11.1. This compiler supports most object oriented features of FORTRAN 2003 standard, allowing us to update programming practices with this important methodology. The main purpose of this paper is to show that FORTRAN can be used with a level of abstraction much higher than observed in practice (in particular, in engineering), by using OOP concepts. The article presents the state of the art in FORTRAN compilers and standards with respect to OOP and discusses concepts of data abstraction, encapsulation and information hiding, classes and objects. Concepts are presented independently of programming language, but their implementation is illustrated in FORTRAN 90/95/2003. Construction of polymorphic classes, by type-extension and inheritance, is illustrated using the new Intel compiler. In addition, the article shows that polymorphism can be emulated in older compilers and in FORTRAN 90/95, by the appropriate use of pointers. Concepts are illustrated in an academic and didactic example, involving a university management system, which manipulates persons, students, professors, courses and dates.

Keywords: Computer programming. Object oriented programming. OO programming. FORTRAN.

[1] Professor do Departamento de Engenharia de Estruturas da EESC-USP, atbeck@sc.usp.br
[2] Pós-doutorando do Departamento de Engenharia de Estruturas da EESC-USP, favb@sc.usp.br

1 INTRODUÇÃO

Este artigo tem por objetivo difundir conceitos de programação orientada a objetos (programação OO) aos usuários da linguagem FORTRAN. Tais conceitos permitirão melhor proveito dos recursos de programação OO introduzidos pelo padrão FORTRAN 2003 e implementados recentemente no compilador Intel Visual FORTRAN versão 11.1 (INTEL, 2009). O artigo mostra que a linguagem pode ser usada em um nível de abstração muito maior do que se constata, na prática, no meio científico (em particular, nas engenharias).

Ao ser criado por John Backus nos anos 50, o FORTRAN – *FORmula TRANslator* – foi considerado uma linguagem de alto nível, em comparação com as linguagens de máquina existentes na época (por exemplo, *assembly*). Por esta razão, o FORTRAN teve rápida disseminação e conquistou milhares de usuários. O FORTRAN teve papel fundamental em tornar a programação de computadores acessível ao meio científico. Em consequência, um número enorme de rotinas e programas foi desenvolvido nesta linguagem, ao longo de muitos anos. O FORTRAN é, até hoje, extremamente popular no meio científico.

A partir da sua criação, usuários do FORTRAN desenvolveram vários "dialetos" da linguagem. Com o objetivo de padronizar a linguagem, surge em 1966 o primeiro padrão (*standard*) de uma linguagem de programação: o FORTRAN 66. Mais tarde, em 1977, foi criado o *standard* FORTRAN 77 para padronizar a linguagem novamente, incorporando alguns dos muitos recursos desenvolvidos por diferentes grupos. Durante este desenvolvimento, o FORTRAN sempre permaneceu próximo e explorou muito bem os recursos de *hardware*, sendo, até hoje, uma das linguagens de execução mais rápida.

Nas décadas de 60 a 80, novas linguagens de programação surgiram. Várias destas superaram o FORTRAN em termos de recursos, em especial aqueles que permitem um nível de programação mais abstrato. O padrão FORTRAN 90 (ADAMS et al., 1992) surgiu para, mais do que uniformizar práticas existentes, incorporar ao FORTRAN recursos existentes em outras linguagens. Isto significa que o padrão (o papel) passou a especificar recursos que deveriam ser implementados nos compiladores. O FORTRAN 90 representou uma grande revisão. As principais novidades introduzidas foram: operações com matrizes, muitas funções intrínsecas, alocação dinâmica de matrizes, *user-defined data types*, módulos, *module procedures*, *operator overloading*, interfaces, operação com ponteiros e programação em formato livre (*free form*). Constata-se que muitos destes recursos não são utilizados, por exemplo, por estudantes de engenharia que programam em FORTRAN. Neste artigo, alguns destes recursos, necessários para implementar a programação OO, são abordados. O padrão FORTRAN 95 (ADAMS et al., 1997) não incorporou grandes recursos à linguagem, mas a tornou compatível com o HPF – *High Performance FORTRAN* – para processamento paralelo.

O padrão FORTRAN 2003 (ADAMS et al., 2009) introduziu vários conceitos de programação OO, como *type extension*, a palavra chave CLASS, recursos para hereditariedade e polimorfismo, e *type-bound procedures*, conforme veremos neste artigo. Durante muitos anos, este padrão esteve à frente dos compiladores, pois até recentemente não havia compiladores FORTRAN com suporte aos recursos introduzidos pelo padrão FORTRAN 2003. Em outubro de 2009, a Intel lançou o compilador Intel Visual FORTRAN versão 11.1 (INTEL, 2009), que oferece suporte a grande parte (mas nem todos) dos recursos de orientação a objetos do FORTRAN 2003. Até então, alguns recursos de programação OO podiam ser emulados em compiladores mais antigos, usando recursos do FORTRAN 90/95 (AKIN, 2003; DECYK et al., 1997; DECYK et al., 1998).

O principal objetivo deste artigo é difundir conceitos de programação orientada a objetos aos usuários da linguagem FORTRAN. No entanto, é importante destacar que os recursos de orientação a objetos em FORTRAN são bastante limitados em comparação com os recursos de outras linguagens. Linguagens modernas como C++, Java e Python, possibilitam o uso de herança múltipla, classes bases virtuais, acoplamento dinâmico seletivo (métodos virtuais) e eficiente (tabelas de ponteiros de

métodos virtuais), encapsulamento com membros protegidos (além de privados e públicos) e definição de funções, métodos e classes paramétricas (*templates*), entre outros recursos inexistentes em FORTRAN. Portanto, o presente artigo pode servir como uma ponte para a migração do FORTRAN para uma linguagem mais moderna. Para tanto, o artigo apresenta os principais conceitos de orientação a objetos, e mostra como estes podem ser implementados em FORTRAN. Uma vez compreendido o conceito e as limitações do FORTRAN, o usuário poderá mais facilmente fazer a migração para outra linguagem mais moderna. Comparações com outras linguagens não são realizadas.

Este artigo está disposto da seguinte maneira. A Seção 2 apresenta definição, objetivos e componentes da programação OO. A Seção 3 descreve o conceito de abstração de dados. Na Seção 4, são apresentados os conceitos de classe e objeto. Na Seção 5, é ilustrada a construção de classes por composição, utilizando o FORTRAN 90/95/2003. A Seção 6 apresenta a construção de classes polimórficas utilizando hereditariedade segundo o padrão FORTRAN 2003. A Seção 7 mostra como classes polimórficas podem ser criadas por "emulação", utilizando o padrão FORTRAN 90/95. Na Seção 8, são apresentadas as conclusões.

Como o artigo lida com palavras em inglês, instruções de compilador e excertos de código, a seguinte notação é utilizada:

- *english words* – palavras em inglês aparecem em itálico.
- `CODE INSTRUCTION` – instruções de programação e fragmentos de programa aparecem exclusivamente em maiúsculas, em inglês, com fonte Courier New reduzida (tamanho 9).
- `LARGE CODE PORTIONS` – grandes porções de programa aparecem em um quadro, com numeração de figura. Instruções de programação seguem a simbologia acima, com letras maiúsculas e fonte Courier New reduzida. Nomes de variáveis em programas aparecem em inglês, de forma compatível com as palavras-chave do compilador. Comentários dos programas estão em português.

2 PROGRAMAÇÃO OO: DEFINIÇÃO, OBJETIVOS E COMPONENTES

A filosofia de orientação a objetos está baseada na construção de software como uma coleção estruturada de classes. O foco do desenvolvimento não está nas tarefas que o software deve executar, mas sim nos objetos (do mundo real ou não) que este software deve manipular. Em outras palavras, o foco do desenvolvimento não está no que o programa faz, mas sim em que objetos ele manipula.

Os principais objetivos da programação OO são expansibilidade, reusabilidade e compatibilidade. Por expansibilidade entende-se a capacidade de adaptar ou ampliar o software devido a mudanças nas especificações. Reusabilidade é a capacidade de utilizar programas ou elementos de software previamente programados na construção de novas aplicações. Compatibilidade é a capacidade de combinar elementos de software desenvolvidos por várias partes. Além destes objetivos, são fatores internos (percebidos apenas por aqueles que participam do desenvolvimento) a legibilidade do código e a capacidade de desenvolver grandes programas de forma lógica e objetiva.

A programação orientada a objetos possui 4 aspectos fundamentais: abstração de dados, classes, hereditariedade e polimorfismo. Abstração de dados consiste em extrair as características essenciais dos objetos a serem manipulados pelo programa. Uma classe consiste na implementação destas características em uma unidade de software independente, equipada com a estrutura de dados e as funções necessárias para manipular objetos. Um objeto, por sua vez, é uma porção de informação criada em tempo de execução a partir de uma classe. O desenvolvimento da unidade (classe) está baseado no encapsulamento e na proteção de informação (*encapsulation* e *information hiding*). Logo, aspectos essenciais do comportamento do objeto são visíveis fora da classe, mas

detalhes da implementação ficam escondidos. Hereditariedade é a propriedade a partir da qual (sub-) classes específicas podem ser criadas herdando as características de classes mais gerais. A hereditariedade dá origem a famílias de classes e ao polimorfismo. Polimorfismo é a capacidade de desenvolver software que manipule, genericamente, qualquer objeto originário de uma família de classes. Isto inclui a construção de software capaz de manipular objetos originários de classes desenvolvidas a *posteriori*.

3 ABSTRAÇÃO DE DADOS

Um aspecto fundamental da programação OO é a identificação e caracterização dos objetos (do mundo real ou não) que o programa a ser desenvolvido deve manipular. Chama-se de **abstração de dados** a capacidade de extrair as características essenciais destes objetos. As características são definidas a partir do comportamento; a implementação é secundária e fica escondida. A descrição dos objetos deve ser tal que qualquer usuário seja capaz de utilizar partes do programa, sem conhecer os detalhes internos de como foram programadas. Utiliza-se o termo *Abstract Data Type* (ADT), mas deve-se chamar a atenção que esta nomenclatura não corresponde a um tipo de dado do compilador FORTRAN. Abstração de dados é um conceito independente de linguagem; portanto, pode ser aplicada a qualquer linguagem de programação.

A descrição apropriada de um ADT deve ser precisa e não ambígua, ser tão completa quanto necessária e não deve especificar mais do que o necessário (MEYER, 2000). O foco deve estar nas operações a serem executadas sobre o objeto. A fim de realizar uma especificação independente de implementação, é apropriado utilizar um sistema de assinaturas, conforme ilustrado no exemplo a seguir.

3.1 Especificação de um ADT `CALENDAR_DATE`

Qualquer programa que manipule datas pode fazer uso de um ADT `CALENDAR_DATE`. A descrição deste ADT deve ser tal que sua utilização independa do sistema de datas preferido pelo usuário, isto é, que funcione tanto no sistema DIA/MÊS/ANO como no sistema MÊS/DIA/ANO. O exemplo é simples, mas serve ao propósito de ilustrar o conceito.

- **Especificação do type:** `TYPE(CALENDAR_DATE) DATE`
- **Listagem das funções:**

```
Para QQ I:inteiro, R:real, L:lógica e DATE:TYPE(CALENDAR_DATE)
CREATE:      → DATE
SET_DAY:     DATE x I → DATE      GET_DAY:      DATE → I
SET_MONTH:   DATE x I → DATE      GET_MONTH:    DATE → I
SET_YEAR:    DATE x I → DATE      GET_YEAR:     DATE → I
DATE1_LESS_THAN_DATE2:    DATE x DATE → L
DATE1_GREATER_THAN_DATE2: DATE x DATE → L
DATE_DIFFERENCE:          DATE x DATE → DATE
CONVERT_TO_DAYS:          DATE → I
PRINT_DATE_DMY_FORMAT:    DATE → Ø
PRINT_DATE_MDY_FORMAT:    DATE → Ø
```

- **Axiomas:**

```
Para QQ I:inteiro, R:real, L:lógica e DATE:TYPE(CALENDAR_DATE)
DATE = CREATE_DATE()
SET_DAY(DATE,I)   = DATE          GET_DAY(DATE)   = I
SET_MONTH(DATE,I) = DATE          GET_MONTH(DATE) = I
SET_YEAR(DATE,I)  = DATE          GET_YEAR(DATE)  = I
DATE1_LESS_THAN_DATE2(DATE,DATE) = L
DATE1_GREATER_THAN_DATE2(DATE,DATE) = L
DATE_DIFFERENCE (DATE,DATE) = DATE
```

```
CONVERT_TO_YEARS(DATE) = R
PRINT_DATE_DMY_FORMAT(DATE)
PRINT_DATE_MDY_FORMAT(DATE)
```

- **Pré-condicionadores:**

```
SET_DAY(DATE,I)     requer: 1 ≤ I ≤ número de dias de cada mês
SET_MONTH(DATE,I)   requer: 1 ≤ I ≤ 12
SET_YEAR(DATE,I)    requer: 1 ≤ I  ?
```

Na listagem das funções acima, utiliza-se uma nomenclatura de assinaturas, independente de linguagem de programação, semelhante àquela utilizada em funções na matemática. Na listagem acima, as funções onde `DATE` aparece à direita da seta são funções de criação, ou seja, produzem uma instância de `DATE` a partir de outro *type* ou de nenhum argumento. Quando `DATE` aparece à esquerda da seta, trata-se de uma função de interrogação (*query*), que retorna propriedades de uma ou mais instâncias de `DATE`. Funções em que `DATE` aparece em ambos os lados são funções de comando (ação), que alteram determinada instância de `DATE`.

Pode-se observar que a descrição acima é feita para funcionar corretamente, independentemente do formato em que o usuário estiver trabalhando, i.e., DIA/MÊS/ANO ou MÊS/DIA/ANO. Como contra-exemplo, qualquer função que permitisse a especificação simultânea dos três parâmetros, por exemplo, `CREATE_DATE(DIA,MÊS,ANO)=DATE`, daria margem a erros em tempo de execução se os parâmetros DIA/MÊS fossem trocados.

3.2 Implementação de DATA TYPES em FORTRAN

O conceito de abstração de dados apresentado é, por definição, independente de linguagem de programação. Nesta seção, a título de exemplo, é ilustrada uma possível implementação do ADT `CALENDAR_DATE` em FORTRAN.

ADTs podem ser implementadas em FORTRAN através do uso de estruturas de dados ou *user-defined data types*. Este recurso permite criar estruturas de dados mais complexas do que os chamados *intrinsic data types*: inteiros, reais, reais de dupla precisão, variáveis lógicas, etc. Na sintaxe do FORTRAN, uma possível implementação do ADT `CALENDAR_DATE` é:

```
TYPE CALENDAR_DATE
  INTEGER :: DAY=1, MONTH=1, YEAR=1
END TYPE CALENDAR_DATE
```

As variáveis `DAY`, `MONTH` e `YEAR` são chamadas de atributos do *data type*. A declaração de um *data type* `CALENDAR_DATE` no programa principal ou em alguma sub-rotina cria uma instância deste *data type*. Uma variável para armazenar a data de nascimento de uma pessoa, por exemplo, é criada com a instrução:

```
TYPE(CALENDAR_DATE) :: BIRTH_DATE
```

No FORTRAN, os atributos do *data type* são acessados através do caracter %. Assim, o ano de nascimento desta pessoa é `BIRTH_DATE%YEAR`. Um vetor para armazenar a data de nascimento de 10 pessoas é criado como:

```
TYPE(CALENDAR_DATE) :: BIRTH_DATE(10)
```

O oitavo atributo do vetor `BIRTH_DATE` é acessado como `BIRTH_DATE(8)`. O ano de nascimento desta pessoa é `BIRTH_DATE(8)%YEAR`. O dia de nascimento das 10 pessoas é acessado com a sintaxe `BIRTH_DATE(1:10)%DAY`. O vetor `BIRTH_DATE` também pode ser criado com dimensão alocatável:

```
TYPE(CALENDAR_DATE),ALLOCATABLE :: BIRTH_DATE(:)
```

Os atributos de uma estrutura de dados também podem ser vetores. No compilador Compaq Visual FORTRAN, a partir da versão 6.6.0, estes atributos também podem ser alocatáveis. Como exemplo, considere uma estrutura de dados para armazenar as informações de uma pessoa, como nomes e data de nascimento. Para armazenar *strings* com o número exato de caracteres necessários, este *data type* poderia ser escrito como:

```
TYPE PERSON
  CHARACTER, ALLOCATABLE :: NAME(:),MIDDLENAME(:),SURNAME(:)
END TYPE PERSON
```

Uma instância deste *data type* é criada com a instrução:

```
TYPE(PERSON) :: PROFESSOR
```

Uma vez conhecido o número de caracteres de cada nome (através da leitura de uma *string* temporária, por exemplo), a instrução de alocação é feita para cada atributo:

```
ALLOCATE(PROFESSOR%NAME(5),PROFESSOR%MIDDLENAME(7),PROFESSOR%SURNAME(4))
```

A alocação de atributos pode ser realizada dentro de uma função de criação, pertencente a uma classe. Uma vez alocados os atributos, pode-se atribuir valores aos mesmos:

```
PROFESSOR%NAME(1:5)       = (/'A','N','D','R','E'/)
PROFESSOR%SURNAME(1:4)    = (/'B','E','C','K'/)
```

A estrutura de dados também pode ter dimensão alocatável. Um vetor alocatável de alunos é criado com a instrução:

```
TYPE(PERSON), ALLOCATABLE :: STUDENT(:)
```

Trinta instâncias deste *data type* (i.e., um vetor de 30 alunos) são criadas com a instrução:

```
ALLOCATE(STUDENT(30))
```

Obviamente, esta alocação deve ocorrer antes da alocação de qualquer um dos atributos. Se o nome do primeiro aluno tem 6 letras, a alocação do atributo fica:

```
ALLOCATE(STUDENT(1)%NAME(6))
```

A sintaxe clara e enxuta das estruturas de dados contribui significativamente para melhorar a legibilidade do código, bem como para construir programas complexos com alto nível de abstração.

3.3 Sobrecarga de operador (*operator overloading*)

A sobrecarga de operadores permite que operações (ou funções) envolvendo estruturas de dados mais complexas sejam definidas para os mesmos operadores existentes para os *intrinsic data types* (`+` , `-` , `<` , `>` , ≤ , `==` , `.EQ.` , `.LT.` , etc.). Como exemplo, para ordenar um conjunto de datas em ordem cronológica, ou simplesmente para comparar a ordem cronológica de duas datas, pode-se sobrecarregar os operadores < e > de forma a que operem em *data types* `CALENDAR_DATE`. Para isto, cria-se primeiramente a função que executa a comparação entre as datas e retorna uma variável lógica com o resultado da comparação. Uma função para efetuar a comparação `DATE1 < DATE2` é ilustrada na Figura 1. A sobrecarga de operador é feita a partir da instrução:

```
INTERFACE OPERATOR (.LT.)
  MODULE PROCEDURE DATE1_LESS_THAN_DATE2
END INTERFACE
```

Esta declaração deve estar contida dentro de um módulo, conforme ilustrado na Figura 3.

```
!-------------------------------------------------------------------------------
! VERIFICA SE A DATA1 É MENOR QUE A DATA2. RESULTADO É UMA VARIÁVEL LÓGICA ( V OU F)
!-------------------------------------------------------------------------------
FUNCTION DATE1_LESS_THAN_DATE2(DATE1,DATE2) RESULT(LESS)
  TYPE(CALENDAR_DATE), INTENT(IN) :: DATE1,DATE2
  LOGICAL :: LESS = .FALSE.
  IF(DATE1%YEAR.LT.DATE2%YEAR) THEN
    LESS = .TRUE.
  ELSE IF ((DATE1%YEAR.EQ.DATE2%YEAR).AND.(DATE1%MONTH.LT.DATE2%MONTH)) THEN
    LESS = .TRUE.
  ELSE IF ((DATE1%YEAR.EQ.DATE2%YEAR).AND.(DATE1%MONTH.EQ.DATE2%MONTH).AND. &
                                        (DATE1%DAY.LT.DATE2%DAY)) THEN
    LESS = .TRUE.
  ENDIF
END FUNCTION DATE1_LESS_THAN_DATE2
```

Figura 1 – Código FORTRAN ilustrando função para sobrecarga do operador < (menor que).

4 CLASSES E OBJETOS

4.1 Classes

Segundo MEYER (2000), a implementação de um ADT através de uma forma particular de representação e da codificação em linguagem de computador é conhecida como **classe**. Classes são construídas de forma a encapsular todas as variáveis, funções e sub-rotinas necessárias para manipular um objeto. No FORTRAN, o encapsulamento é feito utilizando módulos; o módulo é a única ferramenta disponível para criar blocos de instruções isolados do restante do programa.

Toda classe deve possuir uma parte pública e outra privada, conforme ilustrado na Figura 2. A parte pública, que deve ser conhecida pelos usuários da classe, corresponde às especificações do ADT. A parte secreta corresponde a uma escolha particular de representação e a detalhes da implementação. Com base na parte pública e na correta identificação das características do ADT, o usuário pode utilizar uma classe sem conhecer os detalhes de implementação. Em FORTRAN, por *default*, os atributos de um *user-defined data type*, bem como funções e sub-rotinas de uma classe, são públicos, a menos que se especifique o contrário. O conceito de *data hiding*, fundamental na programação OO, implica que atributos do *data type* sejam declarados como privados. Apenas as funções e sub-rotinas (mas nem todas) devem ser públicas. A Figura 3 ilustra uma implementação da classe `CALENDAR_DATE` com atributos privados.

Figura 2 – Ilustração das partes pública e secreta de uma classe.

Quando os atributos do *data type* são privados, não podem ser acessados de fora da classe. Assim, qualquer parte do programa principal que utilize o `TYPE(CALENDAR_DATE)` não poderá acessar

diretamente o dia através da instrução `DATE%DAY`. Isto significa que funções devem ser criadas especificamente com este fim, a exemplo da função `GET_DAY(DATE)=DAY` ilustrada na Figura 3. Isto implica a programação de várias funções que podem, a princípio, parecer desnecessárias. No entanto, estas funções irão permitir o polimorfismo das classes. Tais funções também podem aumentar o tempo de processamento; portanto nem sempre é adequado trabalhar com atributos privados.

```
!-------------------------------------------------------------------------------
! IMPLEMENTAÇÃO DE UM ABSTRACT DATA TYPE CALENDAR_DATE EM FORTRAN 90/95/2003
!-------------------------------------------------------------------------------
MODULE CLASS_CALENDAR_DATE
  IMPLICIT NONE
  TYPE, PUBLIC :: CALENDAR_DATE
    PRIVATE
    INTEGER :: DAY=1, MONTH=1, YEAR=0001
  END TYPE CALENDAR_DATE
  INTEGER, PARAMETER :: DAYS_PER_MONTH(12) = (/31,29,31,30,31,30,31,31,30,31,30,31/)
!-------------------------------------------------------------------------------
! SOBRECARGA DO OPERADOR 'MENOR QUE' OU .LT. PARA COMPARA DATAS
!-------------------------------------------------------------------------------
  INTERFACE OPERATOR (.LT.)
    MODULE PROCEDURE DATE1_LESS_THAN_DATE2
  END INTERFACE
                    ! SEGUEM OUTRAS DEFINIÇÕES DE SOBRECARGA DE OPERADOR
!-------------------------------------------------------------------------------
CONTAINS
!-------------------------------------------------------------------------------
! SET_DAY(CD,I) ! FUNÇÃO QUE ATRIBUI O NÚMERO DO DIA EM UMA DATA, VERIFICANDO
!-------------------------------------------------------------------------------
  SUBROUTINE SET_DAY(CD,X)
    INTEGER,INTENT(IN) :: X                    ! DIA A SER ATRIBUÍDO
    TYPE(CALENDAR_DATE),INTENT(INOUT) :: CD    ! DATA
    IF(X.LT.1 .OR. X.GT.DAYS_PER_MONTH(CD%MONTH)) THEN
      PRINT *, 'INVALID DAY!'
    ELSE
      CD%DAY = X
    ENDIF
  END SUBROUTINE SET_DAY
!-------------------------------------------------------------------------------
! I = GET_DAY(CD) ! FUNÇÃO QUE RETORNA O DIA DE UM OBJETO DATA
!-------------------------------------------------------------------------------
  FUNCTION GET_DAY(CD) RESULT(X)
    INTEGER X                                  ! DIA
    TYPE(CALENDAR_DATE),INTENT(IN) :: CD       ! DATA
    X = CD%DAY
  END FUNCTION GET_DAY
!-------------------------------------------------------------------------------
! DATE1_LESS_THAN_DATE2(DATE,DATE) = L
!-------------------------------------------------------------------------------
  FUNCTION DATE1_LESS_THAN_DATE2(DATE1,DATE2) RESULT(LESS)
                              ! IMPLEMENTAÇÃO CONFORME FIGURA 1
  END FUNCTION DATE1_LESS_THAN_DATE2
                              ! OUTRAS FUNÇÕES E SUB-ROTINAS
END MODULE CLASS_CALENDAR_DATE
```

Figura 3 – Implementação da classe `CALENDAR_DATE` em FORTRAN 90/95/2003.

4.2 Objetos

Objetos são instâncias de classes, criados em tempo de execução a partir de "fôrmas" que são as próprias classes. Portanto, toda a programação OO é, na verdade, baseada em ADTs e classes; os objetos só são criados (diz-se que são *instanciados*) em tempo de execução. A relação entre ADTs, classes e objetos é ilustrada na Figura 4. Os objetos instanciados em tempo de execução equivalem aos objetos utilizados na definição dos *abstract data types* (ADTs).

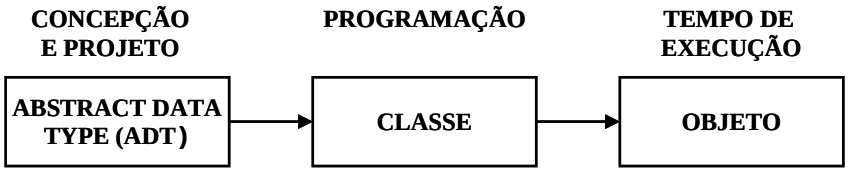

Figura 4 – Relação “temporal” entre ADTs, classes e objetos.

5 CONSTRUÇÃO DE CLASSES POR COMPOSIÇÃO

Há duas maneiras de construir classes a partir de outras classes: composição e hereditariedade. A hereditariedade é descrita na Seção 6.

Na composição, uma classe já existente é utilizada na construção de uma nova classe. Desta forma, a nova classe pode utilizar todos os atributos e métodos da classe existente, como se fosse um cliente desta. A título de exemplo, a Figura 5 ilustra a construção de uma classe `PERSON` utilizando o *data type* `CALENDAR_DATE`, cuja implementação foi ilustrada na Figura 3.

No exemplo ilustrado, a classe `PERSON` é cliente da classe `CALENDAR_DATE`, pois utiliza métodos desta para manipular datas. A instrução `USE CLASS_CALENDAR_DATE` permite a utilização dos elementos desta classe. Para comparar a idade de duas pessoas, a classe `PERSON` pode utilizar diretamente a função `DATE1_LESS_THAN_DATE2` ou o operador sobrecarregado “<” (ilustrado na Figura 3). A data de nascimento de uma pessoa é definida utilizando as funções `SET_` da classe `CALENDAR_DATE` (Figura 3).

No exemplo, a função que atualiza a idade de uma pessoa, `UPDATE_AGE`, faz uso de funções da classe `CALENDAR_DATE`, na linha destacada em negrito (Figura 5). Uma delas é a função diferença entre datas, sobrecarregada no operador de subtração “–”, que resulta também em um *data type* data (diferença em anos, meses e dias). A outra é a função `CONVERT_TO_YEARS`, que converte uma data (na verdade, a diferença entre duas datas) em anos. A sintaxe torna-se legível: a linha em negrito mostra explicitamente que a idade de uma pessoa é calculada em anos inteiros, a partir da diferença entre a data atual e a data de nascimento desta pessoa.

As funções `GET_PERSON_FULL_NAME` e `GET_PERSON_ID`, mostradas na Figura 5 a título de ilustração, são utilizadas mais adiante neste artigo.

6 HEREDITARIEDADE E POLIMORFISMO NO FORTRAN 2003

Um dos recursos mais importantes de qualquer linguagem OO é a hereditariedade (MEYER, 2000). Através deste recurso, é possível criar classes derivadas (filhos) que herdam as suas principais características da classe base (classe pai). Linguagens OO são capazes de criar automaticamente classes polimórficas, com base na extensão de *data types*. O polimorfismo permite a um programa manipular, de forma genérica, objetos instanciados a partir de diferentes classes, que tenham sido ou que venham a ser geradas a partir da extensão de uma mesma classe geral. O polimorfismo é fundamental para obter expansibilidade e reusabilidade de partes do programa.

Hereditariedade e polimorfismo foram incorporados ao FORTRAN através do padrão FORTRAN 2003. No entanto, até recentemente, não existia compilador comercial com suporte a estes recursos de linguagem. Apenas em outubro de 2009, foi lançado o compilador Intel Visual FORTRAN versão 11.1 (INTEL, 2009), com suporte a estes recursos. O exemplo apresentado nesta seção deve ser executado neste compilador ou em versão mais recente.

```
!-------------------------------------------------------------------------------
! CLASS PERSON – IMPLEMENTAÇÃO DE UMA CLASSE PESSOA (FORTRAN 90/95/2003)
!-------------------------------------------------------------------------------
MODULE CLASS_PERSON
USE CLASS_CALENDAR_DATE
IMPLICIT NONE
!-------------------------------------------------------------------------------
  INTEGER, PARAMETER :: NS = 20
  CHARACTER(NS), PARAMETER :: BLANK = '                    '
  TYPE PERSON
    PRIVATE
    CHARACTER(NS) ::       NAME = BLANK     ! NOME PRÓPRIO OU PRIMEIRO NOME
    CHARACTER(NS) :: MIDDLENAME = BLANK     ! NOME DO MEIO
    CHARACTER(NS) ::    SURNAME = BLANK     ! SOBRENOME OU NOME DE FAMÍLIA
    INTEGER       :: CPF = 0         ! NÚMERO DE CADASTRO DE PESSOA FÍSICA
    INTEGER       ::  RG = 0         ! NÚMERO DE REGISTRO GERAL
    INTEGER       :: AGE = 0         ! IDADE
    TYPE(CALENDAR_DATE) :: DOB       ! DATA DE NASCIMENTO (ABREVIADO DE DATE OF BIRTH)
  END TYPE PERSON
!-------------------------------------------------------------------------------
CONTAINS
!-------------------------------------------------------------------------------
! UPDATE_AGE(PE) – DETERMINA A IDADE DE UMA PESSOA A PARTIR DA DATA ATUAL
!-------------------------------------------------------------------------------
  SUBROUTINE UPDATE_AGE(PE)
    USE DFLIB
    TYPE(PERSON)        PE
    TYPE(CALENDAR_DATE) TODAY
    INTEGER(2)          D,M,Y
    CALL GETDAT(Y,M,D)  ! GETDAT: FUNÇÃO BIBLIOTECA DFLIB RETORNA ANO, MÊS E DIA ATUAL
    CALL SET_YEAR(TODAY,Y)
    CALL SET_MONTH(TODAY,M)
    CALL SET_DAY(TODAY,D)
    PE%AGE = INT(CONVERT_TO_YEARS(TODAY-PE%DOB))
  END SUBROUTINE UPDATE_AGE
!-------------------------------------------------------------------------------
! C = GET_PERSON_FULL_NAME(PE) ! RETORNA UMA STRING COM O NOME COMPLETO
!-------------------------------------------------------------------------------
  FUNCTION GET_PERSON_FULL_NAME(PE) RESULT(FULL_NAME)
    TYPE(PERSON) PE
    CHARACTER(3*NS)   FULL_NAME
    IF(PE&MIDDLENAME.EQ.BLANK) THEN
      FULL_NAME = TRIM(PE%NAME)//' '//TRIM(PE%SURNAME)//BLANK//BLANK
    ELSE
      FULL_NAME = TRIM(PE%NAME)//' '//TRIM(PE%MIDDLENAME)//' '//TRIM(PE%SURNAME)
    ENDIF
  END FUNCTION GET_PERSON_FULL_NAME
!-------------------------------------------------------------------------------
! I = GET_PERSON_ID(PE)        ! RETORNA NÚMERO DE IDENTIFICAÇÃO DA PESSOA (RG)
!-------------------------------------------------------------------------------
  FUNCTION GET_PERSON_ID(PE) RESULT(ID)
    TYPE(PERSON) PE
    INTEGER ID
    ID = PE%RG
  END FUNCTION GET_PERSON_ID
END MODULE CLASS_PERSON
```

Figura 5 – Ilustração de composição: construção de classe `PERSON` utilizando `CALENDAR_DATE`.

Considere a especificação da classe `PERSON` na Figura 5. Para construir um programa de gestão acadêmica, é importante distinguir dois tipos fundamentais de pessoas: professores e alunos. Ambos são pessoas; portanto, têm nome, sobrenome, documento de identidade, data de nascimento, etc. No entanto, professores e alunos são tipos particulares de pessoas, ao menos no que se refere ao ambiente acadêmico. Um aluno, por exemplo, terá associados a si um curso (de graduação, por exemplo), um número de identidade estudantil, uma data de entrada no curso, uma data de saída (formatura) e um histórico escolar contendo informações sobre as disciplinas cursadas, como ano,

semestre e conceito final. Um professor, por outro lado, terá associados a si uma data de contratação, salário, dados bancários (para pagamento de salário), informação sobre dependentes, etc.

Professores e alunos são tipos particulares de pessoas; portanto, classes `STUDENT` e `PROFESSOR` podem ser construídas com base na classe `PERSON` (classe pai). A Figura 6 ilustra a construção da classe `STUDENT` por hereditariedade, no FORTRAN 2003, usando a palavra chave `EXTENDS`. Ao "estender" a classe `PERSON`, a classe `STUDENT` herda seus atributos e métodos. Portanto, objetos da classe `STUDENT` passam a ter atributos `NAME`, `MIDDLENAME`, `SURNAME`, `AGE`, etc. A idade de um aluno, por exemplo, é o atributo `STUDENT%AGE`. De forma semelhante ao ilustrado na Figura 6, pode ser construída uma classe `PROFESSOR`.

```
!-------------------------------------------------------------------------------
! IMPLEMENTAÇÃO DA CLASSE STUDENT EM FORTRAN 2003 (INTEL VISUAL FORTRAN VERSÃO 11.1.048).
! POR HEREDITARIEDADE, O TYPE STUDENT HERDA ATRIBUTOS E MÉTODOS DA CLASSE PERSON.
! POR COMPOSIÇÃO, O TYPE STUDENT USA ATRIBUTOS E MÉTODOS DA CLASSE CALENDAR_DATE.
!-------------------------------------------------------------------------------
MODULE CLASS_STUDENT
  USE CLASS_PERSON
! USE CLASS_TRACK_RECORD ! CLASSE QUE IMPLEMENTA OS CAMPOS DO HISTÓRICO ESCOLAR COM
! ANO/SEMESTRE, FREQÜÊNCIA, CONCEITO OBTIDO, RESULTADO FINAL (APROV. OU REPROV.)
  IMPLICIT NONE
  TYPE, PUBLIC, EXTENDS(PERSON) :: STUDENT
    PRIVATE
    INTEGER             :: ID = 1       ! NÚMERO DA IDENTIDADE ESTUDANTIL
    TYPE(CALENDAR_DATE) :: ENROLMENT    ! DATA DE MATRÍCULA NO CURSO
    TYPE(CALENDAR_DATE) :: CONCLUSION   ! DATA DE CONCLUSÃO DO CURSO
!   TYPE(TRACK_RECORD)  :: TR(50)       ! HISTÓRICO DE ATÉ 50 DISCIPLINAS CURSADAS
  END TYPE STUDENT
!-------------------------------------------------------------------------------
  CONTAINS
!-------------------------------------------------------------------------------
! I = GET_STUDENT_ID(ST) ! RETORNA NÚMERO DE IDENTIDADE ESTUDANTIL DO ALUNO
!-------------------------------------------------------------------------------
  INTEGER FUNCTION GET_STUDENT_ID(ST) RESULT(ID)
    TYPE(STUDENT) :: ST
    ID = ST%ID
  END FUNCTION GET_STUDENT_ID
END MODULE CLASS_STUDENT
```

Figura 6 – Implementação da classe `STUDENT` por *type extension* usando FORTRAN 2003.

O potencial de uma classe está no polimorfismo obtido pela construção *type extension*. Um programa pode ser escrito para manipular pessoas, de forma genérica, sejam elas alunos, professores ou dependentes destes. Naturalmente, há algumas funções que são específicas de cada uma das classes especializadas. Seguindo o exemplo, apenas alunos recebem notas e apenas professores recebem salário. No entanto, ainda como exemplo, considere um programa de gestão da biblioteca. A biblioteca empresta livros para alunos, para professores ou para dependentes destes (pessoas em geral). Portanto, um usuário da biblioteca pode pertencer a qualquer uma das classes discutidas. No programa de gestão da biblioteca, um usuário pode ser declarado como:

```
CLASS(PERSON),POINTER :: USER
```

A notação do FORTRAN deixa claro que `USER` é um ponteiro. A palavra chave `CLASS`, neste caso, indica que este ponteiro poderá apontar para objetos da classe `PERSON` ou de qualquer uma de suas classes derivadas. Portanto, com uma declaração dos possíveis alvos deste ponteiro:

```
TYPE(PERSON ),   TARGET :: PE
TYPE(STUDENT),   TARGET :: ST
TYPE(PROFESSOR), TARGET :: PR
```

o programa poderá, em tempo de execução, decidir para que tipo de usuário da biblioteca o ponteiro `USER` deve apontar. Por exemplo, `USER=>ST` cria uma associação, em tempo de execução, a um objeto da classe `STUDENT`. Diz-se que `PERSON` é o tipo declarado (*declared data type*) da variável `USER`, e que `STUDENT` e `PROFESSOR` são os tipos dinâmicos (*dynamic data types*).

O programa de gestão da biblioteca é escrito de forma a manipular usuários genericamente, sejam eles `PERSON`, `STUDENT`, `PROFESSOR` ou mesmo um *data type* criado *a posteriori* (por exemplo, um funcionário). Para isto, torna-se necessária uma pequena "correção" do código apresentado na Figura 5 (que está no padrão FORTRAN 90/95): na definição das funções da classe `PERSON`, o código `TYPE(PERSON)` deve ser substituído por `CLASS(PERSON)`. Isto permite que todas as classes derivadas façam uso das funções da classe pai (`UPDATE_AGE`, `GET_PERSON_FULL_NAME`, `GET_PERSON_ID`, etc.). Portanto, o nome completo do usuário genérico `USER` acima é obtido com a instrução `GET_PERSON_FULL_NAME(USER)`. Para o compilador FORTRAN (Intel versão 11.1), as funções `UPDATE_AGE` e `GET_PERSON_FULL_NAME` são automaticamente polimórficas, pois podem operar com argumentos da classe `PERSON` ou de qualquer uma das sub-classes derivadas.

As funções da classe pai que operam com argumentos `CLASS(PERSON)` podem ser utilizadas diretamente pelas classes derivadas. No entanto, em situações específicas, pode ser necessário redefinir o comportamento de determinadas funções para algumas das classes derivadas. Um aluno, por exemplo, possui suas identificações de pessoa (RG e CPF) e o número de identificação estudantil. Para um programa de gestão acadêmica, o número de identificação estudantil pode ser mais relevante do que o RG. Portanto, poderia ser necessário reformular a função `GET_PERSON_ID`, para retornar o número de identificação estudantil quando o argumento for da classe `STUDENT`. Este procedimento é conhecido como *procedure overriding* (sobre-escrita de função): ao criar a (sub-)classe derivada por *type extension*, algumas funções são re-escritas, utilizando o mesmo nome, mas operando explicitamente com argumentos da classe derivada. As Figuras 7 e 8 ilustram trechos de código que deveriam ser modificados na construção das classes `PERSON` e `STUDENT`, respectivamente, para possibilitar a sobre-escrita de funções. É importante observar nestas figuras que os argumentos *dummy* das funções sobre-escritas devem coincidir não apenas na posição mas também no nome. No exemplo, o nome `PE` é utilizado como argumento da função `GET_ID` nas classes `PERSON`, `STUDENT` e, eventualmente, `PROFESSOR`.

Alternativamente, pode ser utilizada a construção `SELECT TYPE` (apenas FORTRAN 2003). A Figura 9 ilustra este procedimento para o caso da função `GET_PERSON_ID`. A instrução `SELECT TYPE` é capaz de identificar o *data type* do argumento, fazendo, explicitamente, a seleção da função apropriada para cada argumento. Nota-se que, neste caso, a função `GET_PERSON_ID` deve ser re-escrita, com nomes diferentes, para cada um dos possíveis argumentos. Quando o recurso *procedure overriding* é utilizado, as funções são re-escritas com o mesmo nome (por exemplo, `GET_ID`) e o polimorfismo é realizado, de forma automática, pelo compilador.

Outra novidade do FORTRAN 2003 é que métodos (funções ou sub-rotinas) também podem ser atributos do *user-defined data type*. Métodos atributos de determinada classe são declarados usando a palavra chave `CONTAINS` dentro da definição do *type* (Figuras 7 e 8). No exemplo ilustrado, a função `GET_PERSON_ID` seria acessada, na forma tradicional, como `I=GET_PERSON_ID(PE)`. A referida novidade, aliada ao recurso de sobre-escrita de funções, permite a sintaxe mais natural e mais compacta `I=PE%GET_ID`.

```
!-------------------------------------------------------------------------------
! CLASS PERSON – IMPLEMENTAÇÃO DE UMA CLASSE PESSOA (FORTRAN 90/95/2003)
!-------------------------------------------------------------------------------
MODULE CLASS_PERSON
USE CLASS_CALENDAR_DATE
IMPLICIT NONE
!-------------------------------------------------------------------------------
  INTEGER, PARAMETER :: NS = 20
  CHARACTER(NS), PARAMETER :: BLANK = '                    '
  TYPE PERSON
    PRIVATE
    CHARACTER(NS) ::       NAME = BLANK     ! NOME PRÓPRIO OU PRIMEIRO NOME
    CHARACTER(NS) :: MIDDLENAME = BLANK     ! NOME DO MEIO
    CHARACTER(NS) ::    SURNAME = BLANK     ! SOBRENOME OU NOME DE FAMÍLIA
    INTEGER       :: CPF = 0         ! NÚMERO DE CADASTRO DE PESSOA FÍSICA
    INTEGER       ::  RG = 0         ! NÚMERO DE REGISTRO GERAL
    INTEGER       :: AGE = 0         ! IDADE
    TYPE(CALENDAR_DATE) :: DOB       ! DATA DE NASCIMENTO (ABREVIADO DE DATE OF BIRTH)
    CONTAINS
      PROCEDURE,PUBLIC :: GET_ID
  END TYPE PERSON
  PRIVATE :: GET_ID
!-------------------------------------------------------------------------------
CONTAINS
!-------------------------------------------------------------------------------
! SEGUE CÓDIGO IGUAL AO DA FIGURA 5
!-------------------------------------------------------------------------------
! I = GET_ID(PE)        ! RETORNA NÚMERO DE IDENTIFICAÇÃO DA PESSOA (RG)
!-------------------------------------------------------------------------------
  FUNCTION GET_ID(PE) RESULT(ID)
    CLASS(PERSON) PE
    INTEGER ID
    ID = PE%RG
  END FUNCTION GET_ID
END MODULE CLASS_PERSON
```

Figura 7 – Classe PERSON para sobre-escrita de funções (alterações em relação à Figura 5 em negrito).

```
!-------------------------------------------------------------------------------
! IMPLEMENTAÇÃO DA CLASSE STUDENT EM FORTRAN 2003 (INTEL VISUAL FORTRAN VERSÃO 11.1.048).
! POR HEREDITARIEDADE, O TYPE STUDENT HERDA ATRIBUTOS E MÉTODOS DA CLASSE PERSON.
! POR COMPOSIÇÃO, O TYPE STUDENT USA ATRIBUTOS E MÉTODOS DA CLASSE CALENDAR_DATE.
!-------------------------------------------------------------------------------
MODULE CLASS_STUDENT
  USE CLASS_PERSON
! USE CLASS_TRACK_RECORD ! CLASSE QUE IMPLEMENTA OS CAMPOS DO HISTÓRICO ESCOLAR COM
! ANO/SEMESTRE, FREQÜÊNCIA, CONCEITO OBTIDO, RESULTADO FINAL (APROV. OU REPROV.)
  IMPLICIT NONE
  TYPE, PUBLIC, EXTENDS(PERSON) :: STUDENT
    PRIVATE
    INTEGER             :: ID = 1       ! NÚMERO DA IDENTIDADE ESTUDANTIL
    TYPE(CALENDAR_DATE) :: ENROLMENT    ! DATA DE MATRÍCULA NO CURSO
    TYPE(CALENDAR_DATE) :: CONCLUSION   ! DATA DE CONCLUSÃO DO CURSO
!   TYPE(TRACK_RECORD)  :: TR(50)       ! HISTÓRICO DE ATÉ 50 DISCIPLINAS CURSADAS
    CONTAINS
      PROCEDURE,PUBLIC :: GET_ID
  END TYPE STUDENT
  PRIVATE :: GET_ID
!-------------------------------------------------------------------------------
  CONTAINS
!-------------------------------------------------------------------------------
! I = GET_ID(PE) ! RETORNA NÚMERO DE IDENTIDADE ESTUDANTIL DO ALUNO
!-------------------------------------------------------------------------------
  INTEGER FUNCTION GET_ID(PE) RESULT(ID)
    CLASS(STUDENT) :: PE
    ID = PE%ID
  END FUNCTION GET_ID
END MODULE CLASS_STUDENT
```

Figura 8 – Classe STUDENT para sobre-escrita de funções (alterações em relação à Figura 6 em negrito).

```
!-------------------------------------------------------------------------------------
! IMPLEMENTAÇÃO DA CLASSE PERSON EM FORTRAN 2003 (INTEL VISUAL FORTRAN VERSÃO 11.1.048).
! P = PONTEIRO PARA TYPE PERSON OU PARA QUALQUER UM DOS DOS SEUS TYPES DERIVADOS.
! AS FUNÇÕES GET_TYPE E GET_ID PODERÃO SER IMPLEMENTADAS POR SOBRE-ESCRITA (OVERRIDING)
! EM NOVAS VERSÕES DO COMPILADOR.
!-------------------------------------------------------------------------------------
MODULE CLASS_PERSON_POINTER
  USE CLASS_PERSON
  USE CLASS_STUDENT
  USE CLASS_PROFESSOR
  IMPLICIT NONE
  TYPE :: PERSON_POINTER
    CLASS(PERSON), POINTER :: P
  END TYPE PERSON_POINTER
!-------------------------------------------------------------------------------------
  CONTAINS
!-------------------------------------------------------------------------------------
! C = GET_TYPE(PE) ! RETORNA STRING COM IDENTIFICAÇÃO DO TIPO DE PESSOA
!-------------------------------------------------------------------------------------
  CHARACTER(LEN=12) FUNCTION GET_PERSON_TYPE(X) RESULT(NAME)
    CLASS(PERSON)   X
    SELECT TYPE(X)
    TYPE IS (PROFESSOR)
      NAME = '   PROFESSOR'
    TYPE IS (STUDENT)
      NAME = '     STUDENT'
    CLASS IS (PERSON)
      NAME = 'PESSOA COMUM'
    END SELECT
  END FUNCTION GET_PERSON_TYPE
!-------------------------------------------------------------------------------------
! I = GET_ID(PR) ! RETORNA NÚMERO DE IDENTIFICAÇÃO DE PESSOA, ALUNO OU PROFESSOR
!-------------------------------------------------------------------------------------
  INTEGER FUNCTION GET_ID(X) RESULT(ID)
    CLASS(PERSON) :: X
    SELECT TYPE(X)
    TYPE IS (PROFESSOR)
      ID = GET_PROFESSOR_ID(X)
    TYPE IS (STUDENT)
      ID = GET_STUDENT_ID(X)
    CLASS IS (PERSON)
      ID = GET_PERSON_ID(X)
    END SELECT
  END FUNCTION GET_ID
END MODULE CLASS_PERSON_POINTER
```

Figura 9 – Complementação da classe `PERSON` usando FORTRAN 2003.

A Figura 10 ilustra um programa de gestão da biblioteca, construído para manipular, de forma genérica, pessoas, professores e/ou alunos. O programa utiliza um vetor de pessoas (`USER`) para armazenar as informações sobre os usuários. No entanto, um vetor declarado como `CLASS(PERSON),POINTER` não pode apontar para tipos distintos de dados. Esta limitação é contornada pela criação de um *data type* `PERSON_POINTER`, que pode apontar indistintamente para objetos da classe `PERSON` ou de qualquer sub-classe (Figura 9). Um vetor destes objetos (`USER(:)`), com dimensão alocatável, é criado no programa principal (Figura 10). Em tempo de execução, o programa principal obtém as quantidades de cada tipo de usuário, calcula o número total de usuários e faz a alocação dos vetores. A seguir, o programa atribui a cada atributo do vetor `USER` um dos *data types* disponíveis: `PERSON`, `PROFESSOR` ou `STUDENT`, conforme ilustrado. Esta atribuição é feita através de ponteiros, i.e., cada atributo do vetor `USER` aponta para um *data type* diferente. A partir deste ponto, o programa atua sobre o vetor `USER` de forma independente do *data type* dos atributos, sempre que se tratar de função aplicável à classe `PERSON`, ou seja, a pessoas em geral. Na Figura 10, observa-se que foi criado o ponteiro `PT`, declarado como `CLASS(PERSON),POINTER`, o qual é utilizado para acessar a função `GET_ID` como um atributo do *data type*.

```
!-------------------------------------------------------------------------------
! PROGRAMA USUÁRIOS DA BIBLIOTECA (FORTRAN 2003, COMP. INTEL VISUAL FORTRAN V. 11.1.048)
!-------------------------------------------------------------------------------
PROGRAM LIBRARY_USERS
  USE CLASS_PERSON_POINTER
  IMPLICIT NONE
  INTEGER I
  INTEGER N_USERS, N_PR, N_ST, N_PE
  TYPE(PERSON_POINTER),ALLOCATABLE :: USER(:)
  CLASS(PERSON),POINTER :: PT
  TYPE(PERSON),   TARGET,ALLOCATABLE :: PE(:)
  TYPE(STUDENT),  TARGET,ALLOCATABLE :: ST(:)
  TYPE(PROFESSOR),TARGET,ALLOCATABLE :: PR(:)
  ! N_USER = INQUIRE_NUMBER_OF_USERS() ! ROTINA QUE OBTÉM QTD. DE USUÁRIOS (NÃO IMPLEM.)
  N_PE = 1; N_ST = 2; N_PR = 1; N_USERS = N_PE + N_ST +N_PR
  ALLOCATE(USER(N_USERS), PE(N_PE), ST(N_ST), PR(N_PR))
!-------------------------------------------------------------------------------
! CRIA LIGAÇÃO (BINDING) AOS DIFERENTES DECLARED E DYNAMIC DATA TYPES
!-------------------------------------------------------------------------------
  USER(1)%P => ST(1)   ! 1º USUÁRIO É UM ALUNO
  USER(2)%P => PE(1)   ! 2º USUÁRIO É UMA PESSOA COMUM
  USER(3)%P => ST(2)   ! 3º USUÁRIO É UM ALUNO
  USER(4)%P => PR(1)   ! 4º USUÁRIO É UM PROFESSOR
!-------------------------------------------------------------------------------
! ATUA SOBRE O DATA TYPE POLIMÓRFICO INDEPENDENTEMENTE DO DECLARED E DYNAMIC DATA TYPES
!-------------------------------------------------------------------------------
  CALL SET_FULL_NAME(USER(1)%P,'GREGORIO ','DENER ','DONATO')
  CALL SET_FULL_NAME(USER(2)%P,'JOSE',' ','MONTEIRO')
  CALL SET_FULL_NAME(USER(3)%P,'LUI','CHENG','LIU')
  CALL SET_FULL_NAME(USER(4)%P,'ANDRE','TEOFILO','BECK')
  DO I=1,N_USERS
    CALL SET_DOB(USER(I)%P,9,4,1969+I)
    CALL UPDATE_AGE(USER(I)%P)
  ENDDO
  WRITE(*,10) ' '
  WRITE(*,10) 'USUARIOS DA BIBLIOTECA:
  WRITE(*,10) ' '
  WRITE(*,20) 'TIPO','NOME                  ','ID ','IDADE'
  DO I=1,N_USERS
    PT => USER(I)%P
    WRITE(*,100) GET_PERSON_TYPE(USER(I)%P),GET_FULL_NAME(USER(I)%P), &
                 PT%GET_ID(),GET_AGE(USER(I)%P)
    NULLIFY(PT)
  ENDDO
  PAUSE
!-------------------------------------------------------------------------------
  DEALLOCATE(USER, ST, PE, PR)
 10 FORMAT(A)
 20 FORMAT('  ',A12,' ',A22,' ',A3,' ',A5)
100 FORMAT('  ',A12,' ',A22,' ',I3,' ',I5)
END PROGRAM LIBRARY_USERS
```

Figura 10 – Programa ilustrando uso de classe polimórfica PERSON no FORTRAN 2003.

7 POLIMORFISMO POR EMULAÇÃO NO FORTRAN 90/95

Os recursos de hereditariedade e polimorfismo, obtidos no compilador Intel versão 11.1 através da palavra chave EXTENDS (*type extension*), não estão disponíveis em compiladores mais antigos nem fazem parte dos padrões FORTRAN 90/95. Portanto, pode-se dizer que não há recursos para programação orientada a objetos nestes padrões ou nos compiladores mais antigos. Ainda assim, existem maneiras de emular (imitar) polimorfismo nestes compiladores (AKIN, 2003; DECYK et al., 1997; DECYK et al., 1998).

Os exemplos da Seção 6 são reapresentados nesta seção, mas utilizando o padrão FORTRAN 90/95 no compilador Compaq Visual FORTRAN versão 6.6.0. Considerando a

disponibilidade de recursos de polimorfismo no novo compilador Intel, há duas razões para ilustrar a emulação destes recursos em compiladores mais antigos:

- atender àqueles usuários que não têm à disposição o novo compilador da Intel e
- mostrar aos usuários do FORTRAN em geral, de forma bastante explícita, como os novos recursos foram implementados, automaticamente, no novo compilador.

Esta ilustração deve ser bastante elucidativa para usuários familiares ao padrão FORTRAN 90/95. No entanto, ficará evidente que a emulação de polimorfismo em compiladores mais antigos requer um número significativamente maior de linhas de código.

Na falta de *type extension* e hereditariedade, o *derived data type* `STUDENT` deve ser criado exclusivamente por composição, conforme ilustrado na Figura 11. Neste caso, o *type* `STUDENT` não herda os atributos do pai; estes passam a ser atributos do *derived data type*:

```
STUDENT%PERSON%NAME
STUDENT%PERSON%MIDDLENAME
STUDENT%PERSON%SURNAME
```

A sintaxe não é tão natural como aquela obtida com o recurso *type extension*. O exemplo da Figura 11 ilustra apenas uma das muitas funções do *type* `STUDENT`. A função `GET_STUDENT_FULL_NAME` retorna uma *string* com o nome completo do aluno, fazendo uso da função `GET_PERSON_FULL_NAME` (ilustrada na Figura 5). Pode-se observar que a função `GET_STUDENT_FULL_NAME` serve apenas como uma máscara para chamar a função original `GET_PERSON_FULL_NAME` com o parâmetro correto.

```
!-------------------------------------------------------------------------------------
! IMPLEMENTAÇÃO DA CLASSE STUDENT EM FORTRAN 90/95 (COMPAQ VISUAL FORTRAN VERSÃO 6.6.0).
! POR COMPOSIÇÃO, O TYPE STUDENT USA ATRIBUTOS E MÉTODOS DAS CLASSES DATE E PERSON
!-------------------------------------------------------------------------------------
MODULE CLASS_STUDENT
  USE CLASS_PERSON
  IMPLICIT NONE
  TYPE STUDENT
    PRIVATE
    TYPE(PERSON)        :: PERSON
    INTEGER             :: ID = 1     ! NÚMERO DA IDENTIDADE ESTUDANTIL
    TYPE(CALENDAR_DATE) :: ENROLMENT  ! DATA DE MATRÍCULA NO CURSO
    TYPE(CALENDAR_DATE) :: CONCLUSION ! DATA DE CONCLUSÃO DO CURSO
  END TYPE STUDENT
!-------------------------------------------------------------------------------------
  CONTAINS
!-------------------------------------------------------------------------------------
! C = GET_STUDENT_FULL_NAME(ST) ! RETORNA O NOME COMPLETO
!-------------------------------------------------------------------------------------
  FUNCTION GET_STUDENT_FULL_NAME(ST) RESULT(FULL_NAME)
    TYPE(STUDENT) ST
    CHARACTER(3*NS) FULL_NAME
    FULL_NAME(1:3*NS) = GET_PERSON_FULL_NAME(ST%PERSON)
  END FUNCTION GET_STUDENT_FULL_NAME
END MODULE CLASS_STUDENT
```

Figura 11 – Implementação da classe `STUDENT` por composição usando FORTRAN 90/95.

O *data type* polimórfico `PERSON` (em verdade, `PERSON_POINTER`) é criado, no FORTRAN 90/95, com atributos ponteiros, conforme ilustrado na Figura 12. O *data type* `PERSON_POINTER` possui um atributo que permite “apontar” para cada um dos *data types* derivados. Em tempo de execução, a associação (*dynamic binding*) é feita em relação a um destes *data types* e desfeita em relação aos demais. A função `ASSOCIATE_TO_PERSON(PE)`, por exemplo, associa o *data type* polimórfico

PERSON_POINTER a um *data type* PERSON. A mesma função elimina qualquer eventual associação a outro *data type*, através da instrução NULLIFY().

Além das funções de associação, é necessário criar uma máscara para cada uma das funções da classe pai. A Figura 12 ilustra a função GET_FULL_NAME(PT), que, de acordo com o *dynamic data type* associado, chama a função adequada com os parâmetros corretos. Observe que esta função faz o papel da instrução SELECT TYPE do FORTRAN 2003 (Figura 9).

A sintaxe do programa LIBRARY_USERS, em FORTRAN 90/95, fica muito parecida com a versão para FORTRAN 2003. A maior diferença é a associação dinâmica em tempo de execução, que, neste caso, é feita através de rotinas de associação, como:

```
USER(I) = ASSOCIATE_TO_PERSON(PE(J))
```

A partir desta associação, o programa manipula genericamente objetos USER, independentemente do *data type* de seus atributos. Portanto, a maior parte das linhas de código adicionais, necessárias para "emular" o polimorfismo no FORTRAN 90/95, fica escondida na implementação das classes derivadas e da classe "genérica" PERSON_POINTER.

```
!-------------------------------------------------------------------------------------
! IMPLEMENTAÇÃO DA CLASSE "POLIMÓRFICA" PERSON_POINTER EM FORTRAN 90/95
!                                                    (COMPAQ VISUAL FORTRAN VERSÃO 6.6.0)
!-------------------------------------------------------------------------------------
MODULE CLASS_PERSON_POINTER
  USE CLASS_PERSON
  USE CLASS_STUDENT
  USE CLASS_PROFESSOR
  IMPLICIT NONE
  TYPE PERSON_POINTER
    PRIVATE
    TYPE(PERSON   ),POINTER :: PERSON    ! PONTEIRO PARA UM PERSON TYPE
    TYPE(STUDENT  ),POINTER :: STUDENT   ! PONTEIRO PARA UM STUDENT TYPE
    TYPE(PROFESSOR),POINTER :: PROFESSOR ! PONTEIRO PARA UM PROFESSOR TYPE
  END TYPE PERSON_POINTER
!-------------------------------------------------------------------------------------
  CONTAINS
!-------------------------------------------------------------------------------------
! PT = ASSOCIATE_TO_PERSON(PE) ! ASSOCIA O TYPE POLIMÓRFICO A UM TYPE PERSON
!-------------------------------------------------------------------------------------
  FUNCTION ASSOCIATE_TO_PERSON(PE) RESULT(PT)
    TYPE(PERSON),TARGET,INTENT(IN) :: PE
    TYPE(PERSON_POINTER)           :: PT
    PT%PERSON => PE                ! ASSOCIAÇÃO DE PONTEIRO PARA TIPO PESSOA
    NULLIFY(PT%STUDENT)
    NULLIFY(PT%PROFESSOR)
  END FUNCTION ASSOCIATE_TO_PERSON
!-------------------------------------------------------------------------------------
                        ! OUTRAS FUNÇÕES DE ASSOCIAÇÃO PARA STUDENT E PROFESSOR
!-------------------------------------------------------------------------------------
! C = GET_FULL_NAME(PT)  ! RETORNA UM STRING COM O NOME COMPLETO
!-------------------------------------------------------------------------------------
  FUNCTION GET_FULL_NAME(PT) RESULT(FULL_NAME)
    TYPE(PERSON_POINTER) :: PT
    CHARACTER(3*NS)         FULL_NAME
    IF(ASSOCIATED(PT%PERSON))   FULL_NAME(1:3*NS)=GET_PERSON_FULL_NAME(PT%PERSON)
    IF(ASSOCIATED(PT%STUDENT))  FULL_NAME(1:3*NS)=GET_STUDENT_FULL_NAME(PT%STUDENT)
    IF(ASSOCIATED(PT%PROFESSOR))FULL_NAME(1:3*NS)=GET_PROFESSOR_FULL_NAME(PT%PROFESSOR)
  END FUNCTION GET_FULL_NAME
END MODULE CLASS_PERSON_POINTER
```

Figura 12 – Implementação da classe "polimórfica" PERSON_POINTER em FORTRAN 90/95.

8 CONCLUSÕES

Este artigo apresentou alguns conceitos da programação orientada a objetos: abstração de dados, classes, objetos, encapsulamento e restrição de acesso a dados, polimorfismo e construção de classes por composição e hereditariedade, bem como ilustrou a implementação destes conceitos utilizando FORTRAN 90/95/2003. O artigo mostrou que recursos de programação orientada a objetos do padrão FORTRAN 2003 estão disponíveis no compilador Intel Visual FORTRAN versão 11.1, lançado recentemente. Com estes recursos, já é possível criar classes polimórficas e desenvolver programas FORTRAN segundo o paradigma de orientação a objetos.

O conceito de *abstract data types* é, por definição, independente de linguagem de programação. O FORTRAN possui o recurso de estruturas de dados (*user-defined data types*), que permite a implementação dos ADTs na construção de classes. O FORTRAN possui módulos, que permitem encapsular o conteúdo de uma classe, possibilitando a proteção de informação. Recursos específicos para a programação orientada a objetos, como hereditariedade e polimorfismo, foram introduzidos pelo padrão FORTRAN 2003 e estão disponíveis no compilador Intel Visual FORTRAN versão 11.1.

O artigo mostrou ainda que, através do uso de ponteiros e por composição, é possível imitar o comportamento de classes polimórficas no FORTRAN 90/95, ainda que ao custo de algumas linhas de programação adicionais. A imitação de polimorfismo através de ponteiros dá uma idéia explícita de como o polimorfismo automático foi implementado no novo compilador Intel.

É importante registrar que os conceitos apresentados neste artigo são perfeitamente aplicáveis na prática e já foram incorporados a um número de programas desenvolvidos pelo primeiro autor e por colaboradores. Isto inclui um programa de análise de confiabilidade estrutural, com treze mil linhas de código, desenvolvido em FORTRAN 90/95 e que já foi amplamente testado. As duas versões do programa de gestão acadêmica, utilizadas para ilustrar conceitos ao longo deste artigo, estão incompletas mas são perfeitamente funcionais. A versão FORTRAN 90/95 deste programa faz a gestão de turmas de pós-graduação e já possui duas mil e trezentas linhas de código.

A experiência dos autores mostra que, com os recursos disponíveis no compilador Intel Visual FORTRAN versão 11.1 e, mesmo com os recursos existentes no FORTRAN 90/95, é possível atingir os principais objetivos da programação orientada a objetos, isto é: expansibilidade, reusabilidade e compatibilidade dos programas desenvolvidos em FORTRAN. O paradigma de programação orientada a objetos permite desenvolver grandes programas computacionais, de forma clara, lógica e consistente.

9 AGRADECIMENTOS

Os autores agradecem ao CNPq e à CAPES pelo suporte financeiro a este projeto de pesquisa.

10 REFERÊNCIAS

ADAMS, J. C. et al. **Fortran 90 Handbook**: Complete ANSI/ISO Reference. New York: McGraw-Hill, 1992. 823 p. ISBN: 0-07-000406-4.

ADAMS, J. C. et al. **Fortran 95 Handbook**: Complete ANSI/ISO Reference. New York: MIT Press, 1997. ISBN: 0-262-51096-0.

ADAMS, J. C. et al. **The Fortran 2003 Handbook**: The Complete Syntax, Features and Procedures. London: Springer, 2009. 712 p. ISBN: 978-1-84628-378-9.

AKIN, E. **Object oriented programming via FORTRAN 90/95**. Cambridge: Cambridge University Press, 2003. ISBN: 0-521-52408-3.

CHAPMAN, S. J. **Fortran 95/2003 for Scientists and Engineers**. 3ª ed. Cambridge: McGraw-Hill, 2007. 982 p. ISBN: 0-07-319157-4.

DECYK, V. K.; NORTON, C. D.; SZYMANSKI, B. K. Expressing object-oriented concepts in Fortran 90. **ACM Fortran Forum**, v. 16, n. 1, p. 13-18, Abr., 1997.

DECYK, V. K.; NORTON, C. D.; SZYMANSKI, B. K. How to suppor inheritance and run-time polymorphism in Fortran 90. **Computer Physics Communications**, v. 115, n. 1, p. 9-17, 1998. ISSN: 0010-4655.

INTEL. **Intel® Visual Fortran Compiler Professional Edition 11.1 for Windows***: Instalation Guide and Release Notes. 2009.

MEYER, B. **Object-Oriented Software Construction**. 2ª ed. Upper Saddle River: Prentice Hall, 2000. 1254 p. ISBN: 0-13-629155-4.