CENTRO DE INFORMÁTICA

DA UNIVERSIDADE NOVA DE LISBOA

IMPLICAÇÕES DAS LINGUAGENS MODERNAS DE PROGRAMAÇÃO NOS PROJECTOS DE SOFTWARE

Madalena Quirino

José Augusto Legatheaux Martins

(APRESENTAÇÃO NO 1º CONGRESSO DE INFORMÁTICA)

CIUNL-4/80

IMPLICAÇÕES DAS LINGUAGENS DE PROGRAMAÇÃO MODERNAS NOS PROJECTOS DE SOFTWARE

Por: Madalena Quirino e
José Augusto Legatheaux Martins
Departamento de Informática da
Faculdade de Ciências e Tecnologia da
Universidade Nova de Lisboa

SUMÁRIO

As estatísticas sobre gestão de projectos de software têm vindo a demonstrar que os custos de debbug e de manutenção representam uma componente com tendência para absorver a maioria do investimento necessário à realização do projecto. Esta tendência é ainda agravada com o recente aumento da dimensão, complexidade e necessidade de segurança dos sistemas a implementar.

Nesta apresentação abordam-se alguns conceitos de programação introduzidos nas linguagens modernas e com a intenção de solucionar os problemas atrás mensionados, nomeadamente tipos de dados e módulos. Será dada enfase às linguagens da família Pascal e seus descendentes com incidência na linguagem ADA.

Por outro lado certas aplicações que exigem o acesso à máquina concreta, ou que tenham que tomar em consideração aspectos de processamento paralelo não podiam em termos clássicos ser programadas em código de alto nível. Ilustrar-se-á como a linguagem ADA é um exemplo de linguagem que permite construir programas com acesso a esses recursos.

1 - QUALIDADE DE SOFTWARE E PROJECTO DE SOFTWARE

Os últimos anos têm sido caracterizados por uma subida extraordinária da potência de cálculo ao mesmo tempo que os preços, se não têm descido, têm pelo menos tendido para uma certa estabilidade. De qualquer forma, com o mesmo investimento é hoje em dia possível adquirir equipamento muito mais potente.

Esta relativa "banalização" da informática fez explodir o número e dimensão das aplicações. O software de sistema adquiriu uma complexidade muito elevada. É hoje relativamente frequente encontrar, mesmo em minis, sistemas de operação permitindo multiprogramação, acesso a sistemas de tratamento de ficheiros, comunicação através de redes, sistemas de bases de dados. etc.

O software de sistema ou de aplicação passou assim, de peça suplementar e gratuita fornecida com o hardware, a componente previligiada e muitas vezes tão cara como a anterior.

O aumento da complexidade do software, a subida do preço da mão de obra e do seu nível de preparação, fazem com que hoje em dia a actividade de desenvolvimento e manutenção de software seja extremamente dispendiosa.

Um recente estudo realizado pelo Departamento de defesa dos E.U.A. concluiu que o investimento feito no desenvolvimento de software pelos seus departamentos se distribua aproximadamente da seguinte forma: (1)

a) 20% na fase de desenho e codificação

b) 50% na fase de testes

c) 30% na fase de manutensão

---

(1) É este estudo que está na base do projecto que levou este Departamento a encomendar o desenvolvimento da linguagem ADA, [4], [5].

Por outro lado estima-se que a correcção dos erros dos programas considerados já operacionais é cerca de 10 vezes mais dispendiosa do que a fase de teste.

A comparação destes números com os números colhidos uma década atrás, mostra que a parcela a) diminui a sua importância relativa, enquanto a parcela b) cresceu.

Esta tendência a manter-se como que "afogaria" completamente o desenvolvimento de software, e torná-lo-ia uma componente com tendência para absorver exponencialmente o investimento, ao mesmo tempo que se tornaria cada vez mais impossível ter garantias quer do seu funcionamento correcto quer da estimativa do seu custo.

É perante este estado de coisas que a Engenharia de Software tem estabelecido pricípios e recomendado métodos que as linguagens de programação modernas refletam já tanto no que se refere a mecanismos de estruturação de programas como de dados, de modo a permitir modificar drásticamente, quer o processo da produção de software, quer de uma forma geral a sua qualidade.

Software de qualidade implica a utilização de linguagens de alto nível. Um compilador de Cobol equivalia em 1963 a um trabalho de 15 homens ano e a cerca de 40.000 instruções escritas em Assembler. Pelo menos 5 homens trabalhavam nele e nenhum deles o conseguia compreender ou conhecer na totalidade. Um compilador de Concurrent Pascal, linguagem bastante mais potente que o Cobol, foi desenvolvido em 1974 por 2 homens em 14 homens.mês. A utilização da linguagem Pascal para escrita do compilador reduzia-o a 8.300 linhas de código Pascal e tornava-o compreensível por uma só pessoa. [2]

Algumas das características que permitem classificar o software segundo a sua qualidade são:

a) Simplicidade

Um bom programa deve ser simples de compreender. Ele deve ser decomposto em parcelas bem definidas - módulos - realizando tarefas específicas. Estas parcelas devem ser suficientemente pequenas e bem definidas para que sejam fácimente compreendidas.

Para estudar e desenhar o programa estudam-se primeiro, as suas componentes. Uma componente é bem definida pela tarefa que executa. A forma como a executa pode ser em certa fase do desenho ignorada. O estudo do programa no seu conjunto é o estudo da forma como as suas componentes interagem. Pode-se assim controlar o programa ignorando os seus detalhes.

A linguagem deve portanto permitir esta decomposição e deve permitir a ignorância dos detalhes, ou seja deve impedir que os módulos afectem a forma de funcionamento interno uns dos outros (Ver por exemplo o conceito de módulo do MODULA).

Os programas clássicos ao permitirem que as componentes se afectem umas às outras não permitem este tipo de estruturação.

Se a decomposição do programa for clara, ela reflete o próprio raciocínio daquele que o escreveu. A linguagem deve também poder tornar esta forma de "documentação" visível e simples.

b) Fiabilidade

Compreender lógicamente um programa é meio caminho andado para que o possamos fazer certo, sem erros e sem grandes problemas de manutenção.

Mas mesmo assim o programa pode sempre ter erros. Para combater os erros há várias técnicas possíveis:

- provas formais
- testes em tempo de compilação
- testes em tempo de execução
- testes sistemáticos

As demonstrações formais da correcção de programas, estão ainda numa fase muito atrazada e são em geral inaplicáveis para grandes projectos.

Os testes sistemáticos são a técnica mais corrente, mas qualquer programador sabe que são insuficientes porque há sempre uma nova situação ainda não testada que surge e com ela por vezes o erro.

Os testes em tempo de execução são em geral extremamente dispendiosos(Testar se um indice ultrapassa os limites de um array custa em geral cerca de 30 instruções máquina).

Os testes em tempo de compilação são portanto os mais adequados ao processo de construção de software.

É por isso que é fundamental que a linguagem seja definida de uma tal forma que o compilador possa controlar a coerencia do programa.

Veja-se por exemplo o papel do chamado "type-checking", o papel das declarações e o papel dos mecanismos de estruturações que a linguagem introduz.

Este aspecto, exige por exemplo, que durante o processo de teste uma alteração num ponto do programa não tenha possíveis repercussões noutras zonas e que se o tiver o compilador o controle.

c) Adaptabilidade

Um programa de grande dimensão, custa tanto a desenvolver que se

pretende em geral explorá-lo durante anos. Durante esse período de tempo novas adaptações vão surgir geralmente a cargo de outras pessoas. Um programa bem estruturado deve ser bem planeado e bem documentado de modo a facilitar o trabalho das futuras adaptações.

Nesse sentido, as suas componentes devem ser tão gerais quanto possível para evitar que as alterações sejam demasiado difíceis, sobretudo porque serão feitas numa altura menos propícia.

Há que no entanto tomar cuidado de não fazer um monstro, nem sacrificar demasiado a eficiência.

d) Portabilidade

Quando se desenvolve um programa é desejável que ele possa correr em diferentes máquinas. Uma forma de conseguir este aspecto é reduzir os aspectos dependentes da máquina e do seu sistema de operação a módulos bem definidos.

A linguagem deve ter uma definição independente da máquina. Para isso a definição da linguagem tem de ser completa para não deixar ao implementador do compilador, a opção de seguir esta ou aquela forma.

O assembler é a antitese da portabilidade.

e) Eficiencia

A eficiência reduz os custos da computação. Dado o estado actual dos custos das diferentes componentes de um sistema informático, é extremamente perigoso sacrificar os outros critérios a este.

As linguagens modernas tendem a ter um processo de compilação extremamente moroso, mas a produzirem um código mais eficiente. Métodos extre

mamente poderosos de optimização têm sido desenvolvidos com vista a uma melhor eficiencia.

No entanto como em programas complexos as partes significativas sob o ponto de vista de eficiência são restritas (10 a 20%) é nessas que se deve investir na optimização.

2 - MODELOS DA REALIDADE - TIPOS DE DADOS-DETEÇÃO DE ERROS

A actividade da resolução de problemas através do computador é no seu essencial uma actividade de modelização da realidade através de um modelo abstracto da mesma, adequado para o problema em causa. Em seguida para esse modelo, que poderá ser mais ou menos formalizado, é escolhida uma representação adequada na linguagem que se está a utilizar e nela são expressas as operações que se pretendem realizar.

Existem vários problemas inerentes a esta actividade:

a) Verificação da adequação do modelo ao problema em causa. Este problema ultrapassa o quadro do desenvolvimento de software.

b) Verificação de que o modelo está bem representado através da linguagem e que a sua expressão na linguagem de programação é a correspondente ao modelo idealizado.

Ainda que estas duas fases estejam muito interligadas, e ainda que não sejam geralmente explícitas quer verbalmente quer matemáticamente, é sempre através deste meio que a programação de desenvolve, quer na programação de sistema, quer nas aplicações mais matematizadas, quer na gestão, na medicina, etc. quer ainda no limite, em toda a actividade humana da resolução de problemas com auxílio de instrumentos de cálculo (manuais ou automáticos).

Uma das formas mais importantes em que o desenho das linguagens prdgramação influencia fortemente a segunda fase atrás referida é através do conceito de tipo de dados. Daremos em seguida alguns exemplos que ilustram as questões levantadas sem pretendermos levar às últimas consequências toda a potência do conceito ou todas as suas implicações na programação.

Um tipo de dados é um conjunto de valores munido de operações. Esse conjunto de valores pode ser de valores simples, ou de valores que são constituídos à custa de outros conjuntos com eventuais relações entre eles (tipo de dados estruturado). Os tipos de dados podem por sua vez ser pré-definidos (exemplo: inteiros, reais, etc.) ou definidos pelo próprio utilizador.

Tomemos o seguinte exemplo em Pascal, [3]:

```
Type data = Record
                dia: 1.. 31;
                mês: (Jan, Fev, Mar, Abr, Mai, Jun, Jul, Ago,
                      Set, Out, Nov, Dez):
                ano: 1900..2000
            end
```

Acabamos de definir um tipo de dados estruturado cujos valores são ternos ordenados de três conjuntos. O primeiro, dia, pode tomar valores inteiros entre 1 e 31. O segundo, mês, pode tomar valores no conjunto definido em extensão:

{Jan, Fev, Mar, Abr, Mai, Jun, Jul, Ago, Set, Out, Nov, Dez}

e o terceiro, ano, pode tomar valores inteiros entre 1900 e 2000.

Em seguida poder-se-iam declarar por exemplo:

```
Var  data 1, data 2 : data;
     I, J, K: integer;
```

Algumas instruções válidas seriam:

```
data 1. dia := 3;
data 2. mês := Jan;
data 1. ano := 1979;
```

(dia da data 1 recebe o valor 3, mês de data 2 o valor Jan, e ano da data 1 o valor 1979)

No entanto a instrução:

data 1. ano:= 1844 daria erro em tempo de compilação pois

1844 ∉ 1900 ... 2000

Por outro lado data 1. ano:= I seria acompanhado de um teste em tempo de execução para verificar se o valor de I estava em 1900 .. 2000.

Torna-se assim claro como a linguagem não só permite exprimir com clareza o modelo escolhido para representar grosseiramente um aspecto da realidade, como o compilador se torna um poderoso auxiliar para verificar da correcção da utilização desse modelo.

Tomemos outro exemplo mais complexo onde interveem outros tipos de dados definidos pelo utilizador:

```
type  número-do-empregado = 1..100;
type  características = (eficiente, preguiçoso, inteligente, me-
                         diano, burro, bem-apresentado, mal-apre-
                         sentado);
type  funcionários = array [número-do-empregado] of
```

```
record
        nome: array [1..30] of char;
        vencimento: real;
        classificação: set of características;
        admissão: data
end:
```

Uma operação sobre este tipo de dados seria por exemplo:

```
Procedure Promoção (var empregados: funcionários; factor-de-promoção:
                    características; aumento: real);
    var I: número-do-empregado;
    begin
        for I: = first (número-do-empregado) to
                 last  (número-do-empregado) do
            if    factor-de-promoção in
            empregados [I]. Classificação then
            empregados[I]. Vencimento : =
            empregado [I]. Vencimento + aumento
    end;
```

A escrita está num estilo informal, não totalmente conforme à sintaxe de Pascal.

Em tempo de compilação e/ou execução são verificadas todas as utilizações correctas ou incorrectas do tipo de dados funcionários.

Exemplos de erros seriam referências ao funcionário nº 1010 ou consideração de características de classificação fora do conjunto.
{eficiênte, preguiçoso, inteligente, mediano, burro, bem-apresentado, mal--apresentado}

O exemplo ilustra quanto a nós toda a clareza e a elegância com que a linguagem permite trabalhar.

As linguagens desta família permitem exprimir outros tipos de dados muito complexos e com vários outros tipos de estruturação.

Para terminarmos apresentamos um exemplo extremamente curioso permitido pela linguagem ADA:

```
type maçãs is new integer;
type laranjas is new integer;
     A: maçãs; B: laranjas; C: integer;

A: = 0;  B: = 0;
A: = A+A; (maçãs podem ser somadas com maçãs)
I: = A+B; (Erro! maçãs não podem ser somadas com laranjas) (1)
I: = integer (A) + integer (B); (no entanto o total de laranjas e
     maçãs pode ser calculado forçando a conversão).
```

3 - CONCEITO DE MÓDULO - PACKAGES - EXTENSIBILIDADE DA LINGUAGEM

Neste capítulo os conceitos serão todos ilustrados com a linguagem ADA.

Um package em ADA é uma implementação do conceito de módulo, que permite substituir com vantagem, quer o papel dos common em FORTRAN, quer dos packages convencionais em várias linguagens, quer ainda as extensões directamente reconhecidas em fase de compilação às linguagens para manipulação de bases de dados, plotters, Input/Output especial, etc.

---

(1) Um compilador jocoso poderia então escrever uma mensagem de erro assim:

"Como já lhe ensinaram na 1ª. classe maçãs não podem ser somadas a laranjas".

Um package pode encapsular apenas informação ou informação e tipos, pode definir tipos, ou pode até encapsular informações, tipos, operações, outros módulos, sub-programas, etc. O módulo serve para pôr à disposição do seu utilizador um conjunto de conceitos e operações de que ele não necessita em geral conhecer nem aceder a todo o pormenor interno.

Exemplo:

```
package RATIONAL-NUMBER is
type RATIONAL is
    record
        NUMERATOR   : INTEGER;
        DENOMINATOR : INTEGER range 1.. INTER'LAST;
    end record;
function " = " (X,Y : RATIONAL) return BOOLEAN;
function " + " (X,Y : RATIONAL) return RATIONAL;
function " * " (X,Y : RATIONAL) return RATIONAL;
-- Note: " = " hides predefined equality for RATIONAL operands
end;
package body RATIONAL-NUMBER is
procedure SAME-DENOMINATOR (X,Y;in out RATIONAL) is
begin
   reduces X and Y to the same denominator
end;
function " = "(X,Y : RATIONAL) return BOOLEAN is
   U,V : RATIONAL;
begin
   U : = X;
   V : = Y;
   SAME-DENOMINATOR (U,V);
   return U. NUMERATOR = V. NUMERATOR;
```

```
    end " = ";
    function " + " (X,Y: RATIONAL) return RATIONAL is ... end " + ";
    function " * " (X,Y: RATIONAL) return RATIONAL is ::: end " * ";
end RATIONAL-NUMBERS;
```

Neste caso por exemplo o módulo define o tipo número racional. Sobre o tipo número racional existem todas as operações do tipo record. Por outro lado foram redefinidas as operações =, + e * para esse tipo. Ou seja, o utilizador pode fazer:

```
X 1, X2: RATIONAL-NUMBER;
X 1. NUMERATOR : =1;
X 1. DENOMINATOR : =2;
X 2 : = (4, 8);
if X1 = X2 then .....
X1 *  X2 + X2
```

Repare-se que o utilizador não conhece nada sobre a forma como as operações =, * e + estão implementadas, ele apenas sabe que as pode usar visto que elas figuram na parte de especificação do módulo.

O utilizador conhece no entanto os pormenores estruturais do tipo RATIONAL-NUMBER, isto é, que ele é um record com 2 campos NUMERATOR e DENOMINATOR.

A representação de 1 tipo pode no entanto ser escondida ao utilizador. Neste caso ele apenas pode utilizar sobre o tipo as operações que o módulo define e as operações standard de todos os tipos.

Exemplo:

```
Package KEY-MANAGER is
    type KEY is private;
```

```
        NULL-KEY : constant KEY;
        Procedure GET-KEY (K: out KEY);
        function " <" (X,Y : KEY) return BOOLEAN;
    Private
        type KEY is new INTEGER range 0..INTEGER'LAST;
        NULL-KEY : constant KEY : = 0;
    end
```

A esta parte de especificação do package seguir-se-ia o seu corpo com a parte de implementação. Neste caso o utilizador apenas poderia declarar variáveis do tipo KEY e usar sobre elas as operações GET-KEY, < e as operações standard para todos os tipos: assignement e teste de igualdade. Ele não poderia sequer usar o facto de a representação interna de KEY ser por inteiro.

É também possível impedir o uso de todas as operações, a não ser as especificadas pelo módulo.

Através de um package pode-se por exemplo desenhar todo um software de escrita no Plotter, o utilizador apenas tem acesso à parte de especificação do software isto é, ele sabe que existem por exemplo os tipos: PONTO, RECTA, CARACTERÍSTICAS-DO-TRAÇADO. etc. e as operações SET-PEN, MOVE-PEN,etc. Ele não sabe nada sobre a implementação nem pode aceder aos seus promenores internos.

Estes módulos uma vez construídos e testados podem ser guardados em bibliotecas da instalação. Uma instrução especial permite ao utilizador especificar à compilação, quais os packages da biblioteca que quer usar. Podem-se assim construir extenções à linguagem, equivalentes nas linguagens clássicas ao grau de segurança e clareza que se obtêm quando elas são introduzidas pelo compilador, como sucede geralmente com o COBOL ou FORTRAN para ba-

ses de dados.

Este tipo de desenvolvimento de software é possível pois ADA, permite compilação separada de módulos, sub-programas, etc.

4 - TEMPO REAL -CONCORRÊNCIA -

ADA é um exemplo de linguagem que permite exprimir processos concorrentes, isto é, módulos executando-se pseudo-pararelamente ou em multiprogramação, e a sua respectiva sincronização.

Damos em seguida um exemplo retirado de [5] .

Trata-se de um processo que se executa concorrentemente com outros processos e permite a estes comunicarem através de mensagens por exemplo do tipo:

```
Type message is ...;

task MAILBOX is
entry SEND (INMAIL: in message):
entry RECEIVE (OUTMAIL; out message);
  end;
  task body MAILBOX is
      buffer: message;
  begin
      loop
          accept SEND (INMAIL : in message) do
                  buffer : = INMAIL;
          end;
          accept RECEIVE (OUTMAIL: out message) do
                  OUTMAIL : = Buffer;
```

```
        end;
    end loop;
end MAILBOX;
```

Trata-se de um exemplo em que vários processos concorrentes podem enviar mensagens a vários processos receptores. No exemplo, extremamente simples, assim que o processo enviar uma mensagem, chamando a entrada SEND, todos os processos que tentarem enviar outras mensagens ficam retidos em fila de espera, enquanto esta não for recebida por um processo receptor que execute um RECEIVE. Isto é, assegura-se apenas que vários processos que enviam mensagens e vários processos que recebem sejam sincronizados de tal forma que os processos que recebem, recebam pela mesma ordem em que as mensagens são enviadas.

Um problema destes (caso particular do problema dos n produtores e m consumidores) só é possível de programar em linguagens de programação convencionais tendo acesso a System call's, geralmente só programáveis em assembler e portanto sem qualquer espécie de verificação nem em tempo de compilação nem em tempo de execução, quer da coerência quer da correcta utilização dos system call's.

5 - A POSSIBILIDADE PRÁTICA DE ULTRAPASSAR O BINÓMIO FORTRAN/COBOL

a) Os desejos e as realidades

Passámos em revista alguns dos últimos avanços quer na concepção, quer nas ferramentas de software.

A situação na prática, quer no nosso país, quer mesmo em outros países de maior capacidade tecnológica, é caracterizada pela má qualidade do software e dos métodos utilizados na sua produção. É frequente que nu-

ma empresa, ou numa instituição pública, a parcela do tempo máquina e do tempo de trabalho dos profissionais da informática destinada aos testes e à manutenção do software seja da ordem dos 50% a 60% e muitas vezes ainda superior.

As razões deste estado de coisas têm várias origens. Sem intenção de lhes darmos pesos relativos, focaremos algumas delas:

a - 1) A tradição de utilizar apenas o Cobol e o Fortran como linguagens base. Em geral esta opção é baseada em critérios de rentabilidade que talvez não resistam a uma análise mais profunda. De qualquer forma, os utilizadores são quase sempre conduzidos a esta opção, por pressões meramente comerciais e sem tomar em consideração critérios de rentabilidade a mais longo prazo.

a - 2) O mercado e a utilização das linguagens estão extremamente condicionados pelo apoio que um fabricante ou um governo dão a esta ou aquela linguagem.

a - 3) O investimento realizado pelas empresas e pelos fabricantes em software é de tal forma elevado, que cria todo o circulo vicioso: Não se investe noutras linguagens devido aos elevados custos de reconversão e por esse mesmo motivo cria-se um movimento em bola de neve que o torna cada vez mais difícil ainda, apesar de a linguagem se ir revelando cada vez mais inadequada.

a - 4) Desconhecimento generalizado de que existem associações de utilizadores de computadores de quase todas as grandes marcas. Através destas associações é muitas vezes possível adquirir software com boas características a preços extremamente baratos, tornando assim viável a experimentação de outras metodologias e linguagens. É claro que este argumento não é válido num contexto onde não houver a mini

ma capacidade de instalar e manter esse software.

a - 5) Uma proverbial má preparação dos profissionais. A grande maioria dos quais se forma em cursos rápidos de alguns meses, dados pelos fabricantes ou por escolas particulares. Esta formação é em geral extremamente dirigida para uma "produtividade" imediata e assim todos os seus complementos serão conseguidos de uma forma geralmente muito empírica.

b - Conhecer as linguagens de alto nível é extremamente importante

Muitos informáticos mais pragmáticos podem afirmar: "as questões levantadas são interessantes mas não têm aplicação possível nos meus trabalhos".

Este ponto de vista, muito generalizado, não é totalmente correcto. Mesmo para as pessoas que trabalham em Assembler, o conhecimento das questões que tentamos levantar são de uma extrema importância.

Tentaremos apontar algumas razões para esse facto:

b - 1) A análise e primeira versão de um programa para resolver determinado problema pode ser mais fácilmente conseguido com linguagens de alto nível com o conceito de classe, módulo ou package.

Torna-se assim possível obter uma primeira versão da solução, estruturada, legível, bem definida e adaptável. Esta versão pode então ser traduzida manualmente para uma outra linguagem de programação como por exemplo Assembler, Fortran, Cobol, etc.

Esta aproximação facilita extraordinàriamente todo processo de programação e debbug de um programa e facilita a sua documentação e futuras alterações, pois estas são mais fácilmente estudadas sobre o modelo em código de alto nível.

b - 2) Outra via relativamente semelhante a esta e que também permite o aumento da produtividade do trabalho é a utilização dos chamados sistemas de ajuda à programação.

Estes programas ou conjunto de programas permitem, utilizando técnicas semelhantes às expostas, utilizar a capacidade do computador para ajudar o programador no seu trabalho.

Algumas das tarefas a que dão muita ajuda são:

- documentação
- formatação automática de programas
- geração automática de módulos standard numa certa gama de aplicações
- geração de programas, automáticamente, a partir de linguagens extremamente pequenas e definidas apenas para objectivos restritos ou em campos menos pragmáticos como por exemplo a transformação de programas.

b - 3) Tem-se reconhecido que é preferível usar como suporte de uma introdução à programação, uma linguagem de alto nível, pois estas permitem não só formar muito mais rápidamente o aluno na utilização de metodologias de programação mais avançada, como também introduzí-lo numa disciplina de raciocínio que tenha como suporte uma grande quantidade de conceitos semelhantes aos expostos.

Numa linguagem desse tipo, esses conceitos surgem naturalmente e lógicamente e não como novidades extremamente difíceis de percepcionar à primeira vista e que têm por obstáculo toda uma disciplina mental que lhes é oposta, como sucede com a maioria dos programadores.

Em seguida serão então ensinadas outras linguagens de nível mais baixo, mas tendo sempre presente o suporte metodológico anteriormente explicado ao aluno. [6]

b - 4) O actual estado de coisas na programação vai mudar sensívelmente nos próximos anos e mesmo nos contextos industriais o grau de exigência que se vai fazer dos programadores será muito maior. Para essa alteração concorrerá muito o facto de a informática estar nítidamente a ultrapassar a sua fase adolescente e a entrar na sua fase mais adulta. Projectos como o projecto DoD, desempenharão um papel fundamental nessa alteração da situação. As exigências que se farão aos profissionais da informática serão certamente muito superiores, sob pena de as ferramentas colocadas à sua disposição serem sub-aproveitadas.

6 - REFERÊNCIAS

[1]- José A. Legatheaux Martins
Prospectivas da Computação Científica
Seminário nº261 LNEC
Tema-3 Linguagens

[2]- Per Brinch Hansen
the Architecture of Concurrent Programs
Prentice Hall

[3]- K. Jensen, N. Wirth
Pascal user Manual and Report
Springer Verlag

[4]- Jean D. Ichbiah e outros

Preliminary ADA Reference Manual

ACM Sigplan Notices. Junho 1979

[5]- Peter Wegner

Programming with ADA: An Introduction

by means of Graduated Examples

ACM Sigplan Notices. Dezembro 1979

[6]- M. Quirino. L. Monteiro, Legatheaux Martins, Próspero dos Santos.

Educação em Informática

Comunicação a este Congresso (Área-3).