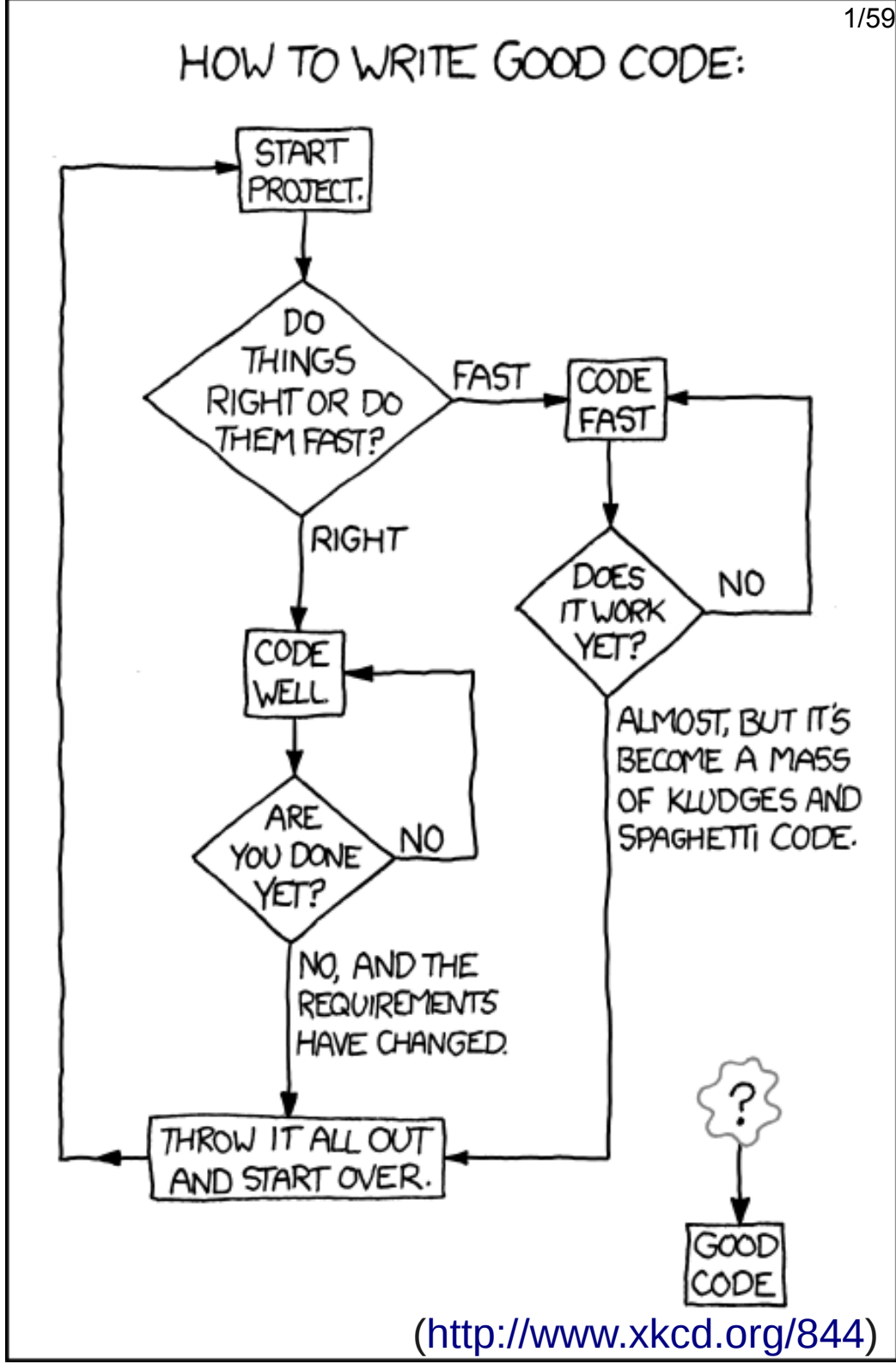
Programação em astronomia: indo além de loops e prints

2 - Organização

Paulo Penteado

http://www.ppenteado.net/pea



(http://www.xkcd.org/844)

Programa

1 – Slides em http://www.ppenteado.net/pea/pea01_linguagens.pdf
- Motivação
- Tópicos abordados
- Tópicos omitidos
- Opções e escolha de linguagens
- Uso de bibliotecas
- Referências

2 – Slides em http://www.ppenteado.net/pea/pea01_organizacao.pdf
- Organização de código
- Documentação
- IDEs
- Debug
- Unit testing

3 – Slides em http://www.ppenteado.net/pea/pea02_variaveis.pdf
- Tipos de variáveis
- Representações de números e suas conseqüências
- Ponteiros
- Estruturas
- Objetos

Programa

4 – Slides em http://www.ppenteado.net/pea/pea03_conteiners.pdf
- Contêiners
- Arrays
- Listas
- Mapas
- Outros contêiners
- Vetorização
- Escolha de contêiners

5 – Slides em http://www.ppenteado.net/pea/pea04_strings_io.pdf
- Strings
- Expressões regulares
- Arquivos

Organização de código

Um dos fatores mais importantes para determinar a qualidade de um software:

- Quanto tempo (e sofrimento) ele consome para ser escrito / editado / analisado
- A robustez
- A reusabilidade

Código-fonte deve ser compreensível por pessoas (não só pelo compilador / interpretador).

Documentação é também uma parte importante, mesmo sendo não executável.

Só porque o programa compila / executa não significa que esteja seja adequado:

- O compilador / interpretador tem memória perfeita e muito maior que a das pessoas, por isso consegue não se perder, e manter a contabilidade de tudo que está acontecendo.

- Particularmente relevante quando o código for ser lido no futuro muito distante (daqui a um mês), quando nem o autor se lembra como ele funciona.

Um programa que executa pode não fazer a coisa certa:

- Inspecionar o código é uma importante forma de saber se o resultado está certo.

Organização de código

Código mal organizado pode também ser pouco eficiente. Exs:

- Gerando cópias desnecessárias de grandes volumes de dados
- Usando mais memória que o necessário
- Acessando disco / memória de forma desordenada (mais lenta que sequencial)
- Reexecutando linhas desnecessariamente.
- **Muito mais tempo é consumindo o escrevendo, testando, consertando e modificando.**

Uso de contêiners, objetos e vetorização são importantes para melhorar a organização de código. (discutidos individualmente, em outras aulas)

Organização de código - estruturação

Estruturação significa dividir o código em unidades (rotinas) menores.

Cada tarefa bem definida fica em uma rotina separada.

O programa todo se torna hierarquizado.

Motivos para estruturação:

- Facilita inspeção, teste, e edição: **nenhuma unidade é longa demais.**
- Unidades podem ser facilmente reaproveitadas.
- Sabe-se que o código de cada unidade está relacionado apenas ao que ela faz; não há linhas de outras tarefas misturadas no meio.
- Há menos variáveis em escopo: diminui a confusão, o uso de memória, e facilita a edição. Uma variável (local) não vai afetar código fora da unidade.
- **Unidades podem ser facilmente trocadas.**
- **Unidades podem ser testadas individualmente**, em geral de forma muito mais fácil e robusta que quando tudo fica misturado em uma longa rotina.
- Facilita colaboração, quando diferentes autores trabalham em diferentes unidades.

Organização de código - estruturação (exemplo negativo)

Um exemplo péssimo (caso real em uso em Astronomia):

Uma rotina (Fortran) de ~4500 linhas, onde as primeiras ~350 são declarações de variáveis, além de variáveis vindo de 12 módulos externos:

- Até compilador e debugger têm dificuldade de lidar com ela.
- Como alguém pode entender tudo que a rotina faz, onde começa e termina cada pedaço do que ela faz, e onde são usadas as centenas de variáveis?
- Quanta memória seria economizada se não houvesse tantas variáveis em escopo simultaneamente?
- Como decidir que nome usar para criar uma nova variável?
- Ao o editar, como saber se um conjunto de 600 linhas que faz uma tarefa pode ser substituído por outra implementação da mesma tarefa?
  - Como saber se no meio destas 600 linhas não há linhas fazendo coisas que nada têm a ver com esta tarefa?
  - Como saber quais são todas as variáveis usadas nessas 600 linhas?
- Destas ~4500 linhas, há só ~1800 com comentários (não é incomum, em código bem organizado, haver mais linhas de documentação do que de código).

Organização de código – estruturação (exemplo negativo)

Neste exemplo, o código segue o padrão:

```
subroutine whatever(arg1, arg2, arg3, ...., arg75)
(centenas de linhas declarando variáveis, quase todas locais)
```

Se possível (em Fortran não é) as declarações devem estar próximas do uso das variáveis, para indicar melhor a que se referem. E quando possível, usando contêiners para diminuir o número de variáveis.

```
(uma linha de comentário dizendo que tarefa é feita adiante):
!Read parameters from file
```

Com freqüência, explicação de menos para código de mais a seguir.

```
(dezenas de linhas de código, presumivelmente para ler o arquivo)
```

Só dá para supor *que todas estas linhas só sirvam para ler o arquivo:* ***não há garantia****.*

Mas em geral elas não só lêem o arquivo. Também alocam variáveis, processam parâmetros, e usam variáveis temporárias.

Como trocar esta parte do código, quando se muda o formato do arquivo?

- *Não dá para recortar este trecho e trocar por outro, sem o risco de quebrar outras partes.*
- *Não dá para copiar este trecho e usar em outro programa (não se sabe quais são as variáveis locais, quais são recebidas do que veio antes, e quais são passadas adiante).*

```
!Read the spectra to process
(dezenas de linhas, presumivelmente para ler o outro arquivo)
(...)
(o mesmo padrão repetido dezenas de vezes até o final)
```

Organização de código – estruturação (como seria melhor)

Como definir a estrutura para o programa?

Antes de começar, é necessário ter uma idéia clara, para cada unidade, de:

1) O que ela precisa receber de entrada
2) O que ela deve fazer
3) O que ela deve retornar a quem a chamou

(1) e (3) constituem a interface da rotina.

(2) é o corpo da rotina – **o que** ela faz, em geral através do uso de outras rotinas (que determinam **como** é feita a tarefa).

(1), (2) e (3) formam uma **especificação** da rotina: tudo que é necessário dizer para a programar.

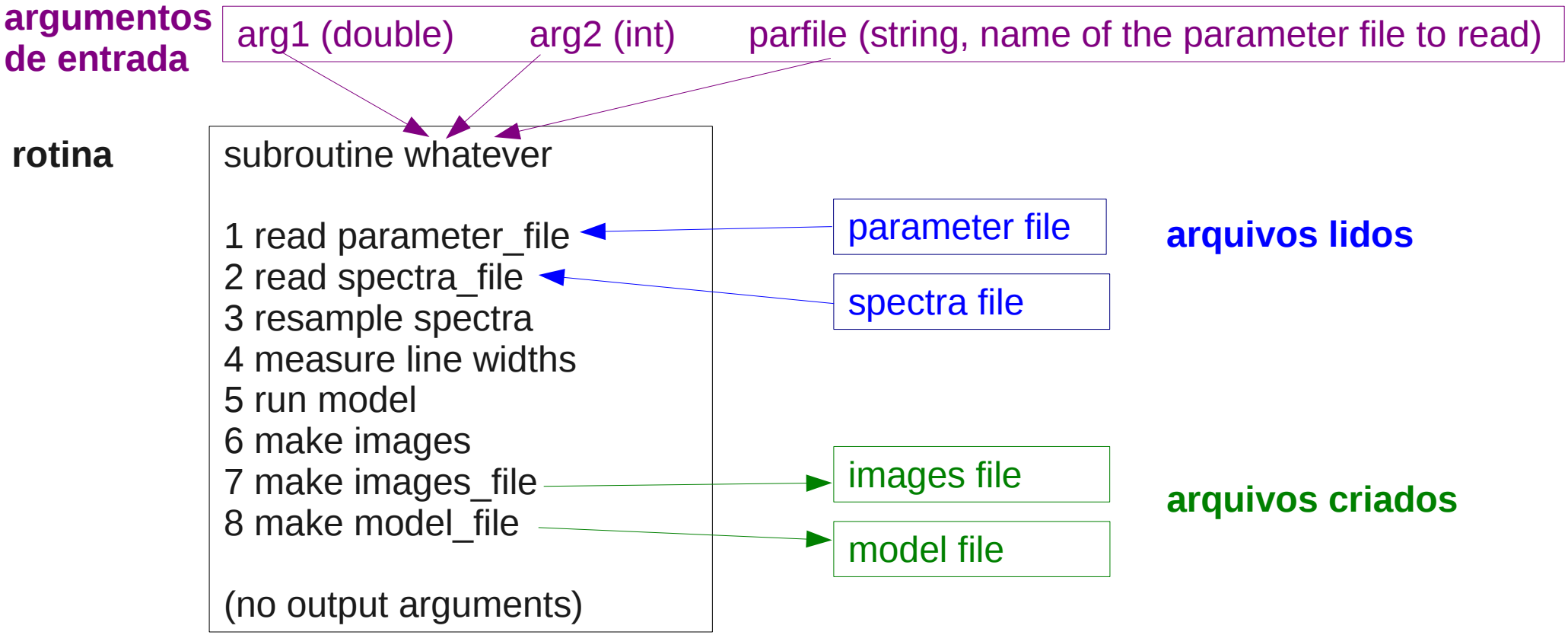
Quem programa a rotina não precisa ser quem a especificou. Mesmo que seja, é importante definir os 3 pontos acima de forma clara, antes de começar a escrever o código.

Freqüentemente é recomendado escrever estes 3 pontos para cada rotina a ser criada, e/ou desenhar um fluxograma de como o programa todo vai ser organizado.

Organização de código – estruturação (como seria melhor)

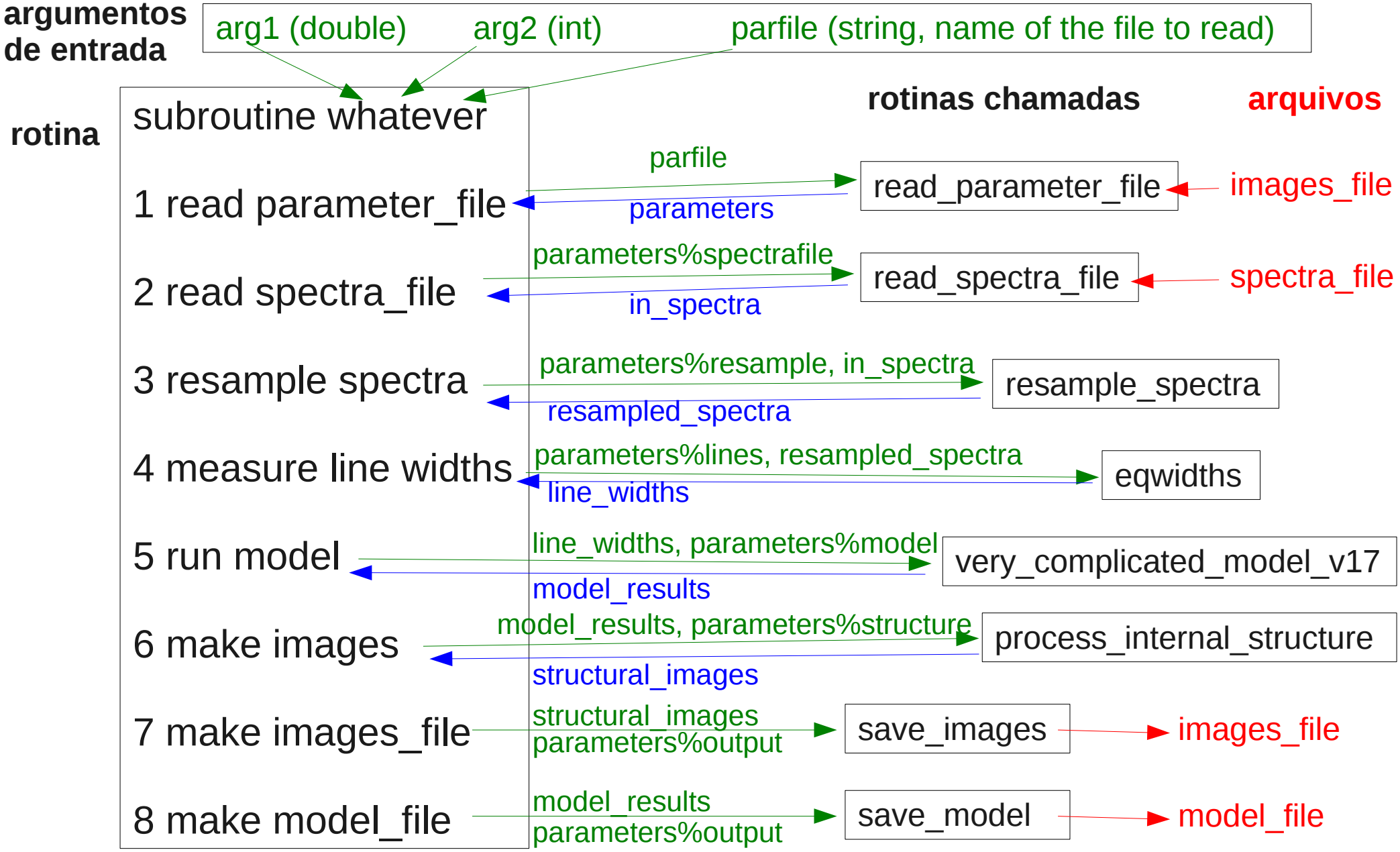
Um diagrama para pensar (visualizar) o que a rotina faz:

- Mostra tudo que a rotina faz (1-8).
- Não mostra como 1-8 são feitos, que não importa agora.
- Mostra como ela se relaciona com o mundo (argumentos de entrada e saída, arquivos lidos e escritos).



Organização de código – estruturação (como deveria ser)

Como fazer cada passo? Chamando uma rotina para cada (e cada uma destas pode chamar outras, para fazer partes específicas).



Organização de código – estruturação (como seria melhor)

```
subroutine whatever(arg1, arg2, parfile=parfile)
(várias linhas de documentação)
(só há 9 variáveis a declarar)

!Read parameters from file
call subroutine read_parameter_file(parfile, parameters)

!Read the spectra to process
call subroutine read_spectra_file(parameters%spectrafile, in_spectra)

!Resample the input spectra to another resolution
resampled_spectra=resample_spectra(in_spectra, paramters%resample)

!Measure the equivalent width of the relevant lines in the spectra
line_widths=eqwidths(resampled_spectra, parameters%lines)

!Use the super fancy model that will answer everything
call subroutine very_complicated_model_v17(parameters%model, line_widths,
model_results)

!Make images showing the structure of the object calculated by the model
call subroutine process_internal_structure(model_results, parameters
%structure, structural_images)

!Write these nice results to files
call subroutine save_images(structural_images, parameters%output)
call subroutine save_model(model_results, parameters%output)
end subroutine whatever
```

Organização de código – estruturação (como seria melhor)

A versão estruturada, é fácil de ser inspecionada:

- Há só 19 linhas de código.
- Há documentação suficiente.
- Há só 9 variáveis em escopo, sendo 6 locais.

Fica fácil ver que a rotina só faz 6 tarefas:

1) Lê parâmetros (1 rotina)
2) Lê arquivos (1 rotina)
3) Processa os dados (2 rotinas)
4) Calcula os modelos (1 rotina)
5) Gera visualizações (2 rotinas)
6) Salva resultados (1 rotina)

A complexidade de cada uma das 6 tarefas não fica exposta, nem misturada entre tarefas.

A complexidade está escondida (*encapsulada*) em 8 rotinas que são chamadas.

Fica claro como dados são passados entre tarefas.

Organização de código – estruturação (como seria melhor)

Fica muito melhor inspecionar / editar / testar o que é feito em cada uma das 8 rotinas:

- O código de cada uma não se mistura com o código das outras.
- Não há variáveis das outras, que não interessam, presentes.
- Em cada uma fica explícito o que são as dependências externas (as variáveis que aparecem na interface), e as variáveis locais não são expostas.
- Cada rotina pode ser escrita e testada individualmente.

É fácil reaproveitar uma destas rotinas para outros programas.

É fácil substituir uma destas rotinas por outra que faça algo diferente:

- Não é mais necessário reamostrar o espectro? Troca-se a chamada de rotina
  `resampled_spectra=resample_spectra(in_spectra, paramters%resample)`
  Por
  `resampled_spectra=in_spectra`.
- É melhor reamostrar usando outro método? Troca-se pelo uso de outra rotina.

Organização de código - interfaces

Interfaces têm grande importância sobre a organização do código.

Só com interfaces organizadas uma rotina é usável, sem sofrimento.

Problemas comuns:

- Muitos argumentos posicionais. Exemplo (real, em Fortran):

```
      SUBROUTINE DISORT( NLYR, DTAUC, SSALB, PMOM, TEMPER, WVNMLO,
     &                   WVNMHI, USRTAU, NTAU, UTAU, NSTR, USRANG, NUMU,
     &                   UMU, NPHI, PHI, IBCND, FBEAM, UMU0, PHI0,
     &                   FISOT, LAMBER, ALBEDO, HL, BTEMP, TTEMP, TEMIS,
     &                   DELTAM, PLANK, ONLYFL, ACCUR, PRNT, HEADER,
     &                   MAXCLY, MAXULV, MAXUMU, MAXCMU, MAXPHI, RFLDIR,
     &                   RFLDN, FLUP, DFDT, UAVG, UU, U0U, ALBMED,
     &                   TRNMED )
```

47 argumentos posicionais (correspondência é feita apenas pela sua ordem).
É pior ainda: todos estes estão declarados estaticamente:

```
      PARAMETER ( MXCLY =6, MXULV = 5, MXCMU = 48, MXUMU = 10,
     &            MXPHI = 3, MI = MXCMU / 2, MI9M2 = 9*MI - 2,
     &            NNLYRI = MXCMU*MXCLY, MXSQT = 1000 )
```

Qualquer mudança nas dimensões exige que se mude os valores nesta rotina, e na que a chama, e recompilar.

Se as declarações não forem iguais nos dois lugares, a rotina pode até rodar, mas fará coisas erradas.

Organização de código - interfaces

Em linguagens estáticas, interfaces são particularmente trabalhosas: **Todos argumentos devem ter declarações compatíveis, na rotina e onde ela é usada:**

- Declarações copiadas entre lugares exigem manter as duas cópias sincronizadas.
- O problema é aliviado com módulos e tipos parametrizados (Fortran 90-2008, mais rudimentar) e *header files* (C, C++), *templates* (C++) e *genéricos* (Java).
- É possível fazer rotinas genéricas, que aceitam argumentos de tipos e tamanhos variáveis.
  - Mas é necessário ou lidar com eles explicitamente (só praticável para casos muito simples) ou usar *templates / genéricos* (maior complexidade).

Linguagens dinâmicas facilitam muito a organização, pelo trabalho muito pequeno imposto pela interface de uma rotina:

- Não há declarações.
- Não há problemas com tipos / tamanhos.
- Nos casos em que a rotina depende de tipos / tamanhos, ela pode o testar, por *introspecção* (rotinas que dão informações sobre as variáveis).
- Keywords (argumentos associados por nome), números variáveis de argumentos, manipulação dos conjuntos de argumentos, e defaults são feitos facilmente, em grande parte por introspecção.

Organização de código - interfaces

É comum uma rotina depender de muitas variáveis:

- Para definir o que vai fazer
- Para retornar resultados.

Ex: um modelo complicado pode depender de dezenas de parâmetros.

Como reduzir o número de argumentos passados (posicionais ou não)?

- Alguns usam variáveis globais (*global* / *common*), que são variáveis visíveis em todas as rotinas dentro de um programa. **Que em geral é uma solução ruim:**
  - Variáveis globais fazem com que uma rotina dependa do ambiente externo de uma forma que não aparece em sua interface. Variáveis passam “por baixo do pano”.
  - Módulos (Fortran) e *includes* (inclusão de arquivos dentro de outros) são menos ruins, pois tudo que é passado escondido está em um mesmo lugar (o módulo ou arquivo incluído): **Coisas ficam escondidas, mas são mais fáceis de encontrar.**

Organização de código - interfaces

Como melhorar a passagem de muitos parâmetros?

Duas soluções (não mutuamente exclusivas):

- **Contêiners** (arrays, estruturas, listas, mapas, objetos):
  - ➔ Carregam vários valores necessários, de forma organizada por ordem (listas, arrays) ou nome (estruturas, mapas, objetos).
  - ➔ O número de variáveis é pequeno, e estas são organizadas.

- **Argumentos associados por nome (keywords):**
  - No lugar de
    ```
    call subroutine(t, p, x, y, z, r, ...)
    ```
    Se usa
    ```
    call subroutine(x=x, y=y, z=z, r=r, temperature=t,
    pressure=p, ...)
    ```
  - **Principalmente úteis para argumentos opcionais**
  - Argumentos de entrada (substituídos por defaults quando não fornecidos)
  - Argumentos de saída (que nem sempre interessam; se não foram usados na chamada da rotina, esta pode pular as partes que geram estas saídas).

Organização de código - interfaces

Voltando ao exemplo anterior, a interface origina (Fortran)

```
      SUBROUTINE DISORT( NLYR, DTAUC, SSALB, PMOM, TEMPER, WVNMLO,
     &                   WVNMHI, USRTAU, NTAU, UTAU, NSTR, USRANG, NUMU,
     &                   UMU, NPHI, PHI, IBCND, FBEAM, UMU0, PHI0,
     &                   FISOT, LAMBER, ALBEDO, HL, BTEMP, TTEMP, TEMIS,
     &                   DELTAM, PLANK, ONLYFL, ACCUR, PRNT, HEADER,
     &                   MAXCLY, MAXULV, MAXUMU, MAXCMU, MAXPHI, RFLDIR,
     &                   RFLDN, FLUP, DFDT, UAVG, UU, U0U, ALBMED,
     &                   TRNMED )
```

Foi substituída por apenas (ainda Fortran)

```
subroutine disort_pp_run(dsv,dsr)
```

Sem alocações estáticas (dimensões declaradas no código fonte).

Todas as dimensões são dinâmicas (determinadas quando o programa executa).

Organização de código - interfaces

A nova interface carrega as mesmas informações de entrada e saída, em duas estruturas. O seu uso fica apenas:

```
dsv%nlayers=nlayers
dsv%degree=degree
dsv%naz=naz
```

As dimensões do modelo são colocadas na estrutura dsv

```
call disortset(dsv,dsr) !aloca os componentes e ajusta os defaults em dsv,dsr
```

Estruturação do código: esta parte foi colocada em outra rotina (46 linhas), que apenas define os defaults para os 48 parâmetros (acessados por nome, obviamente)

```
dsv%tau=tau
dsv%ssa=ssa
dsv%pmom=pmom
dsv%inccos=inccos(1)
dsv%az=azang

write(6,*) 'calling disort'
```

Os 5 parâmetros que não vão ser default são colocados na estrutura dsv

```
call disort_pp_run(dsv,dsr)
```

A rotina é chamada, com apenas 2 variáveis:
dsv para argumentos de entrada
dsr para os de saída

Organização de código - interfaces

O mesmo tipo de algoritmo deste exemplo em Fortran, com uma interface implementada em IDL, usando objetos:

Cria o objeto (que vem com defaults para muitos parâmetros)

```
op=pprtdiscreteordinatesfull(nlayers=nlayers,degree=degree,azang=azang)

op.setboundaryc,inccos=inccos
```

Atribui o valor de um parâmetro

```
for i=0,nwn-1 do begin
  print,'wav: ',strcompress(string(i+1,'/',nwn),/rem)

  tau=a.v.tautot[*,0]
  ssa=a.v.ssa[*,0]
  pmom=a.v.phase[*,*,0]

  op.setatmosphere,opt=tau,ssa=ssa,phasem=pmom

  op.run

  flux[*,i]=op.flux

endfor
```

Atribui o valor de outros parâmetros (que variam com comprimento de onda, por isso estão no loop)

Executa o modelo

Guarda o resultado para este comprimento de onda

Organização de código - exemplo

Outro exemplo simples (IDL), mostrando uso de argumentos opcionais e defaults

A documentação, muito mais longa que a função, foi omitida

```
function pp_gauss_from_fwhm,x,fwhm=fwhm,sigma=sigma
compile_opt idl2, logical_predicate

;Defaults
;Get sigma from fwhm, if fwhm was provided

fsfac=(2d0*sqrt(2d0*alog(2d0)))
sigma=n_elements(fwhm eq 1) ? sigma : 1d0/fsfac
sigma=n_elements(fwhm eq 1) ? fwhm/fsfac : sigma

;Evaluate the Gaussian of standard deviation sigma and mean 0 at the
coordinates given in x

ret=exp(-(x^2)/(2d0*sigma^2d0))
ret/=(sigma*sqrt(2d0*!dpi))

return,ret
end
```

Organização de código - exemplo

Padrão exigido pela ESA para software de análise e processamento de dados do Gaia escrito em Java:

- Documentação (formato javadoc)
- Unit tests (executados automaticamente todo dia)
- Os métodos (rotinas) não podem exceder:
  - 30 linhas de código
  - 5 argumentos
  - Profundidade 5
  - Complexidade ciclomática 10
- As classes não podem exceder:
  - 12 atributos
  - 20 métodos
  - Herança de até 5 classes

Todo o software é mantido em um repositório de controle de versão, com um gerenciador que executa os testes diariamente e gera relatórios da conformidade das classes (Hudson).

Organização de código - controle de fluxo

Não é mito: *GOTO nunca deve ser usado*

October 1995

Classic of the month

Go To Statement Considered Harmful

Edsger W. Dijkstra

Reprinted from *Communications of the ACM*, Vol. 11, No. 3, March 1968, pp. 147-148.

Organização de código - controle de fluxo

Não é mito: *GOTO nunca deve ser usado*

http://www.xkcd.org/292

Organização de código - controle de fluxo

Não é mito: *GOTO nunca deve ser usado*

É uma construção de baixo nível, remanescente de código de máquina:

- Só computadores (com memória perfeita e grande) são capazes de manter a contabilidade do fluxo usando saltos arbitrários (*gotos*).
- Tornam o fluxo do programa **arbitrário, e não estruturado.**
- Para entender o código fica necessário encontrar, lembrar de, e considerar **todas** as ocorrências de *gotos*.
- Editar e mover blocos fica muito difícil.

Sua impropriedade em linguagens de alto nível, para pessoas lerem, é conhecida há muito tempo (Dijikstra 1968).

Não existe em muitas linguagens modernas (Java, Python).

Nas linguagens atuais em que existe, se deve a apenas duas possibilidades:

- Manter compatibilidade com código antigo.
- Para ser usado por por software que gera código (que não tem o objetivo de ser compreendido por pessoas).

Organização de código - controle de fluxo

A maior parte dos usos de *goto* por programadores hoje se deve a não considerar as estruturas de controle mais apropriadas.

Estruturas de controle normalmente disponíveis (exs. com sintaxe de IDL):

1. if .. then .. else

```
if (some_condition) then begin
    (...)
else begin
    (...)
endelse
```

A forma mais importante de controle, executando condicionalmente um bloco.

Para decidir sobre que valor atribuir a uma variável entre duas opções, um *if .. then .. else* pode ser substituído pelo operador ternário de atribuição condicional (*?:*):

```
if a then b=c else b=d
```

É equivalente a

```
b=a ? c : d
```

Comum para defaults:

```
arg=n_elements(arg) ne 0 ? arg : 1.0
```

Organização de código - controle de fluxo

2. Loops *for* – executam o bloco um número predeterminado de vezes

Ex: Imprimir os números pares de 0 a 10, inclusive.

```
for i=0,10,2 do begin
   print,i
endfor
```

i variando de 0 a 10, com passos de tamanho 2
em algumas linguagens (IDL, C, C++, Java, Python) o contador não precisa ser inteiro

3. Loops em contêiners (*foreach*)

el recebe o valor de cada elemento do contêiner (não precisa ser escalar), e *i* o índice do elemento

```
v=[1.8,17.5,49.23]
foreach el, v, i, do begin
   print, 'index 'i, ' element ',el
endforeach
```

Resultado:

```
index             0 element       1.80000
index             1 element       17.5000
index             2 element       49.2300
```

Estritamente, o que Python chama *for* é na verdade um *foreach.*

Organização de código - controle de fluxo

4. Loops *while* – repete o bloco enquanto (se) uma condição é verdadeira

```
i=0
while (i le 10) do begin
    print,i
    i++
endwhile
```

Se i tivesse um valor maior que 10 ao chegar a esta linha pela primeira vez, o loop nunca executaria

i++ é equivalente a *i+=1*, que é equivalente a *i=i+1*.
Também há -- (*i--* é equivalente a *i-=1*, equivalente a *i=i-1*).
E há */=*, **=*, *^=*, *>=*, *<=*.

Loops *for* costumam ser uma abreviação de um *while* como acima (um for pode não ser executado, se a condição dele nunca é verdadeira: *for i=10,9*).

A condição do *while* é livre: poderia ser qualquer teste, mesmo sem haver contadores.

Exemplo menos artificial: ler um arquivo sem saber quantas linhas ele tem

```
file_contents=list()                    ;make an empty list
str=''                                  ;make an empty string
openr,unit,'some_file.txt',/get_lun     ;open the file on unit
while (not eof(unit)) do begin          ;runs the block if not at end of file*
    readf,unit,str                      ;read a line as a string
    file_contents.add,str               ;add the line to the file_contents list
endwhile
```

**eof(unit)*, de *End Of File,* retorna 1 (verdadeiro) se a unidade *unit* está no final do arquivo, 0 (falso) em caso contrário.

Organização de código - controle de fluxo

5. Loops *until* – repete o bloco até que uma condição se torne falsa

```
i=0
repeat begin
    print,i
    i++
endrep until (i gt 10)
```

Como o teste é feito no final, um *until* é sempre executado pelo menos uma vez.

A escolha entre *while* e *until* depende de ser mais natural testar no começo ou no final do loop.

6. return - sai da rotina atual (retornando um valor, se for uma função):

```
function dot_product,x,y
    nx=n_elements(x)
    ny=n_elements(y)
    if ((nx eq 0) || (ny eq 0) || (nx ne ny)) then return, !null
    return,total(x*y)
end
```

Dois pontos de saída: se x, y não são adequados, retorna um valor nulo e não tenta calcular o produto

Organização de código - controle de fluxo

7. *case* - seleciona uma opção a executar dentre várias:

```
case (a) of
    1: begin
        (...)
        end
    2: begin
        (...)
        end
    else: begin
        print, 'invalid a'
        (...)
    end
endcase
```

Se *a* for 1, será executado o primeiro bloco, se for 2, o segundo, etc.

A variável não precisa ser inteira: pode ser comparada com strings, reais, ou qualquer outra coisa.

Organização de código - controle de fluxo

Condições mais complicadas podem ser testadas com *case* “ao contrário”:

```
CASE 1 OF

   (X GT 0) AND (X LE 50): Y = 12 * X + 5

   (X GT 50) AND (X LE 100): Y = 13 * X + 4

   (X LE 200): BEGIN
     Y = 14 * X - 5
     Z = X + Y
   END

ELSE: PRINT, 'X has an illegal value.'

ENDCASE
```

(exemplo tirado da documentação do IDL)

Organização de código - controle de fluxo

8. *switch* - seleciona uma opção a executar dentre várias, mas continuando nas opções seguintes à selecionada:

```
PRO ex_switch, x
   SWITCH x OF
      1: PRINT, 'one'
      2: PRINT, 'two'
      3: PRINT, 'three'
      4: BEGIN
         PRINT, 'four'
         BREAK
        END
      ELSE: BEGIN
         PRINT, 'You entered: ', x
         PRINT, 'Please enter a value between 1 and 4'
         END
   ENDSWITCH
END
```

Executando

```
ex_switch, 2
```

O resultado seria

```
two
three
four
```

(exemplo tirado da documentação do IDL)

Organização de código - controle de fluxo

9. *break* - sai do bloco (*for, foreach, while, repeat, case, switch*) atual, continuando da linha seguinte ao bloco.

10. *continue* pula o restante do loop (*for, foreach, while, repeat*) atual, continuando da primeira linha da próxima iteração (se houver) do loop.

•Em algumas linguagens, como C, C++, Java, pode ser usado também em case e switch.

***return, break e continue* são os mais comuns recursos a usar para mudar o fluxo de execução dependendo de um teste:**

Exemplo (artificial): encontrar quantos dos primeiros elementos de um vetor somam 20, sem contar os negativos:

```
v=[1,-7,-2,3,-5,2,9,7,5,4,8,12,-5]
sum=0
foreach el, v, i do begin

  if (el lt 0) then continue ;ignore negative numbers, going to the next

  sum+=el

  if (sum ge 20) then break ;get out if 20 is reached in the sum

endforeach
print, 'the first ', i, ' elements, not counting negatives, add to ', sum
```

Organização de código – problemas com literais* - exs. reais

(IDL)

```
dlambda   = fwhm / 17D0
interlam  = lammin + dlambda * DINDGEN( LONG( (lammax-lammin)/dlambda+1 ))
interflux = INTERPOL( flux, lam, interlam )
fwhm_pix = fwhm / dlambda
window   = FIX( 17 * fwhm_pix )
```

Por que 17? Só mencionado em comentário 30 linhas acima
Em quantos lugares é necessário mudar o código para usar outro valor?
17 ou 17d0?
Uma variável deveria ter sido definida.

(Fortran)

```
data pi/3.14159/
```

Precisão muito ruim; em geral há uma função (*pi()*) ou constante (*!pi*) para isso. Ou, ***pi=acos(-1d0)***.

```
open(19,file='rtinput2-iso',status='unknown'
(...)
do q=1, nlay
   write(19,222) q, dtau(q), ssalb(q)
   end do
(...)
close(19)
```

Além de ser um literal perdido no meio do código (linha 171), é uma dependência externa, fixa no código.

Além de ser um literal repetido (deveria ser uma variável), é algo de escopo global (a unidade aberta). Assume que o 19 estará livre.

Se esta rotina é usada em outro lugar, o 19 pode já estar ocupado.

*constantes que aparecem *literalmente* dentro do código: **17d0**, **5**, **'some_name'**, etc.

Organização de código – problemas com literais e unidades

Geralmente, literais devem:

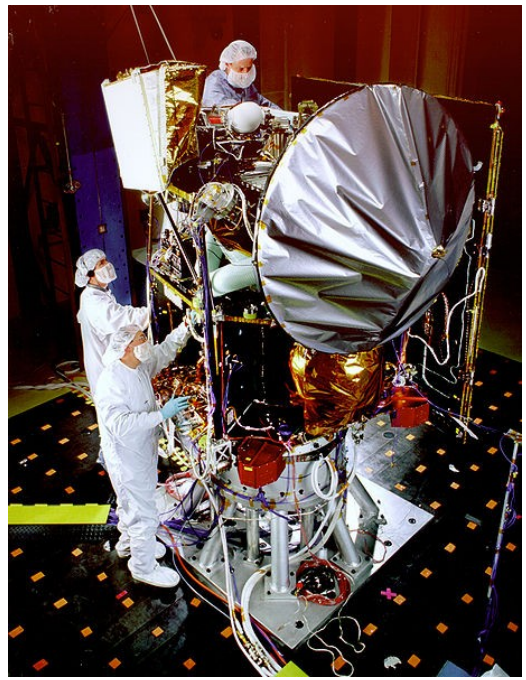
- ser atribuídos a variáveis (ponto único de edição),
- ser explicados com comentários,
- não ser usados para definir recursos globais, que podem não estar disponíveis (ex: número de unidade)

É comum ser desejável que todas as variáveis com literais sejam definidas no começo do código, localizando todas as dependências (de parâmetros, externas) onde são facilmente encontradas.

Se o valor não é adimensional (literal ou não), comentários devem informar as unidades.

Foi a causa da perda da Mars Climate Orbiter:

The metric/US customary units mix-up that destroyed the craft was caused by a human error in the software development. The thrusters on the spacecraft, which were intended to control its rate of rotation, were controlled by a computer that underestimated the effect of the thrusters by a factor of 4.45. This is the ratio between a pound force (the standard unit of force in the United States customary units system) and a newton (the standard unit in the metric system).



É raro, mas há softwares (em geral, classes) que contabilizam as unidades de variáveis e suas transformações em operações matemáticas.

Organização de código – programming pearls

http://users.erols.com/blilly/programming/Programming_Pearls.html

If you can't write it down in English, you can't code it.
Peter Halpern, Brooklyn, New York

If the code and the comments disagree, then both are probably wrong.
Norm Schryer, Bell Labs

If you have too many special cases, you are doing it wrong.
Craig Zerouni, Computer FX Ltd., London, England

Get your data structures correct first, and the rest of the program will write itself.
David Jones, Assen, The Netherlands

Don't make the user provide information that the system already knows.
Rick Lemons, Cardinal Data Systems

Less than 10% of the code has to do with the ostensible purpose of the system; the rest deals with input-output, data validation, data structure maintenance, and other housekeeping.
Mary Shaw, Carnegie-MellonUniversity

Don't write a new program if one already does more or less what you want. And if you must write a program, use existing code to do as much of the work as possible.
Richard Hill, Hewlett-Packard S.A., Geneva, Switzerland

Whenever possible, steal code.
Tom Duff, Bell Labs

Organização de código – programming pearls

http://users.erols.com/blilly/programming/Programming_Pearls.html

Make it work first before you make it work fast.
Bruce Whiteside, Woodridge, Illinois

Translating a working program to a new language or system takes ten percent of the original development time or manpower or cost.
Douglas W Jones, University of Iowa

The fastest algorithm can frequently be replaced by one that is almost as fast and much easier to understand.
Douglas W Jones, University of Iowa

In non-I/O-bound programs, less than four per cent of a program generally accounts for more than half of its running time.
Don Knuth, Stanford University

Before optimizing, use a profiler to locate the "hot spots" of the program.
Mike Morton, Boston, Massachusetts

[One Page Principle] A {specification, design, procedure, test plan} that will not fit on one page of 8.5-by-11 inch paper cannot be understood.
Mark Ardis, Wang Institute

Documentação

É essencial! Não é um detalhe para só considerar depois.

Faz a diferença entre uma rotina ser ou não compreensível, reusável, verificável e útil.

Há dois tipos importantes de documentação:

1. Para explicar o código - **voltada a quem lê o código-fonte** (especialmente o próprio autor), para entender como ele funciona, e por que foi feito daquela forma.

- **Código-fonte deve ser compreensível para pessoas (não só para o compilador/interpretador).**
- **Sempre que escrever uma linha / conjunto de linhas não triviais, escreva na hora um comentário os explicando.**

2. Para explicar o uso do código - **voltada aos usuários** (inclusive o próprio autor), para informar para que serve e como usar aquela rotina.

- Em geral vem um pouco depois na criação do código, quando este se torna menos experimental e suas características estão melhor definidas. Por isso é freqüentemente negligenciada.

Se o código mudar, mude os comentários: comentários não atuais / errados são piores que sua ausência.

Documentação

Apesar de negligenciada, a documentação que explica o propósito e uso do código costuma ser a mais necessária:

- Pode ser lida não só por pessoas, mas também por software, que as usa para criar documentação bem formatada (HTML, LaTeX, PDF, com links para referências), e para as ajudas de contexto das IDEs.

- Para outros usuários do código, é a parte mais importante. Decide se a rotina interessa ou não:
  - Informa o objetivo do código e **o que** ele faz (**não como** ele o faz).
  - Informa se há dependências em outras coisas, e limitações de seu uso.

- Para saber como usar a rotina:
  - Explica todos os argumentos / keywords: seu propósito, o que contém, seu tipo / tamanho, obrigatoriedade, se é entrada e/ou saída.
  - Explica as formas como aquela rotina pode ser usada, e para quê.
  - **Dá exemplos de uso** – uma das partes mais importantes, principalmente mostrando de forma completa como usar a rotina, e que resultado ela deveria gerar quando usada daquela forma.

Principais sistemas de processamento de documentação:

- JavaDoc, IDLDoc, Doxygen (C, C++, Fortran, Java Python e muitas outras), Sphinx (Python).

Documentação - exemplos

Uma rotina muito simples, onde a documentação para seu uso é maior que o código.

Documentação escrita como comentários para serem processados pelo IDLDoc (que gera HTML ou LaTeX).

Resultado HTML:

http://www.ppenteado.net/idl/pp_lib/doc/index.html

```
;+
; :Description:
;    Evaluates a normalized Lorentzian distribution of mean zero and the provided
;    width at the provided locations (`x`). The width by the FWHM (`fwhm`).
;
; :Params:
;    x in, required
;      The locations where the Gaussian is to be evaluated.
;
; :Keywords:
;    fwhm : in, optional, default=1d0
;      Specifies the width of the Gaussian, as its Full Width Half Maximum (FWHM).
;
; :Examples:
;
;   Make a domain where the Lorentzian is to be evaluated::
;
;     nx=201
;     x=(dindgen(nx)/(nx-1d0)-0.5d0)*5d0
;
;   Make a Lorentzian with fwhm=1 and plot it::
;
;     yl=pp_lorentz_from_fwhm(x,fwhm=1d0)
;     pl=plot(x,yl,color='blue',name='Lorentzian, FWHM=1d0',thick=2.)
;
;   Now compare with a Gaussian (made with `pp_gauss_from_fwhm`)::
;
;     yg=pp_gauss_from_fwhm(x,fwhm=1d0)
;     pg=plot(x,yg,color='red',name='Gaussian, FWHM=1d0',thick=2.,/over)
;     l=legend(target=[pg,pl],position=[0.5,0.5]) ;Identify the two lines
;
;   Save the result into the file shown below::
;
;       pl.save,'pp_lorentz_from_fwhm.png',resolution=100
;
;   .. image:: pp_lorentz_from_fwhm.png
;
;
;-
function pp_lorentz_from_fwhm,x,fwhm=fwhm
compile_opt idl2, logical_predicate
;Defaults
fwhm=(n_elements(fwhm) eq 1) ? fwhm : 1d0
;Get gamma (hwhm) from fwhm
gamma=fwhm/2d0

;Evaluate the Lorentzian at each provided coordinate x
ret=(1d0/!dpi)*(gamma/(x^2+gamma^2))
return,ret
end
```

Documentação – programming pearls

http://users.erols.com/blilly/programming/Programming_Pearls.html

When explaining a command, or language feature, or hardware widget, first describe the problem it is designed to solve.
David Martin, Norristown, Pennsylvania

The job's not over until the paperwork's done.
Anon

IDEs

Interactive Develpment Environments

Existem para todas as linguagens (inclusive coisas como LaTeX, HTML, CSS).

Provêm um ambiente integrado para

- edição
- compilação / empacotamento (quando existe)
- execução
- debug
- inspeção de arquivos (não só código-fonte)
- acesso a documentação
- profiling (medida de performance) - em algumas linguagens
- uso interativo – em algumas linguagens
- construção de interfaces gráficas - em algumas linguagens

Ferramenta muito útil, com freqüência ignorada (quando se usa, no lugar de uma IDE, a combinação editor de texto + linha de comando).

Algumas IDEs são multilinguagem e multiplataforma, oferecendo um ambiente uniforme - especialmente o Eclipse, que entende (nativamente ou com plugins) quase qualquer linguagem.

Algumas já são fornecidas juntamente com o compilador / interpretador, outras são independentes.

IDEs

Principais exemplos:

- **Eclipse** - *quase qualquer plataforma e linguagem (mesmo obscuras como IDL e R)*

- MS Visual Studio - Windows, C, C++, VB, C#, e várias outras possíveis (só a versão Express, mais limitada, é gratuita)

- NetBeans - qualquer plataforma, Java, C, C++, e várias outras possíveis

- Xcode - Mac, C, C++, Objective C, Java, e várias outras possíveis

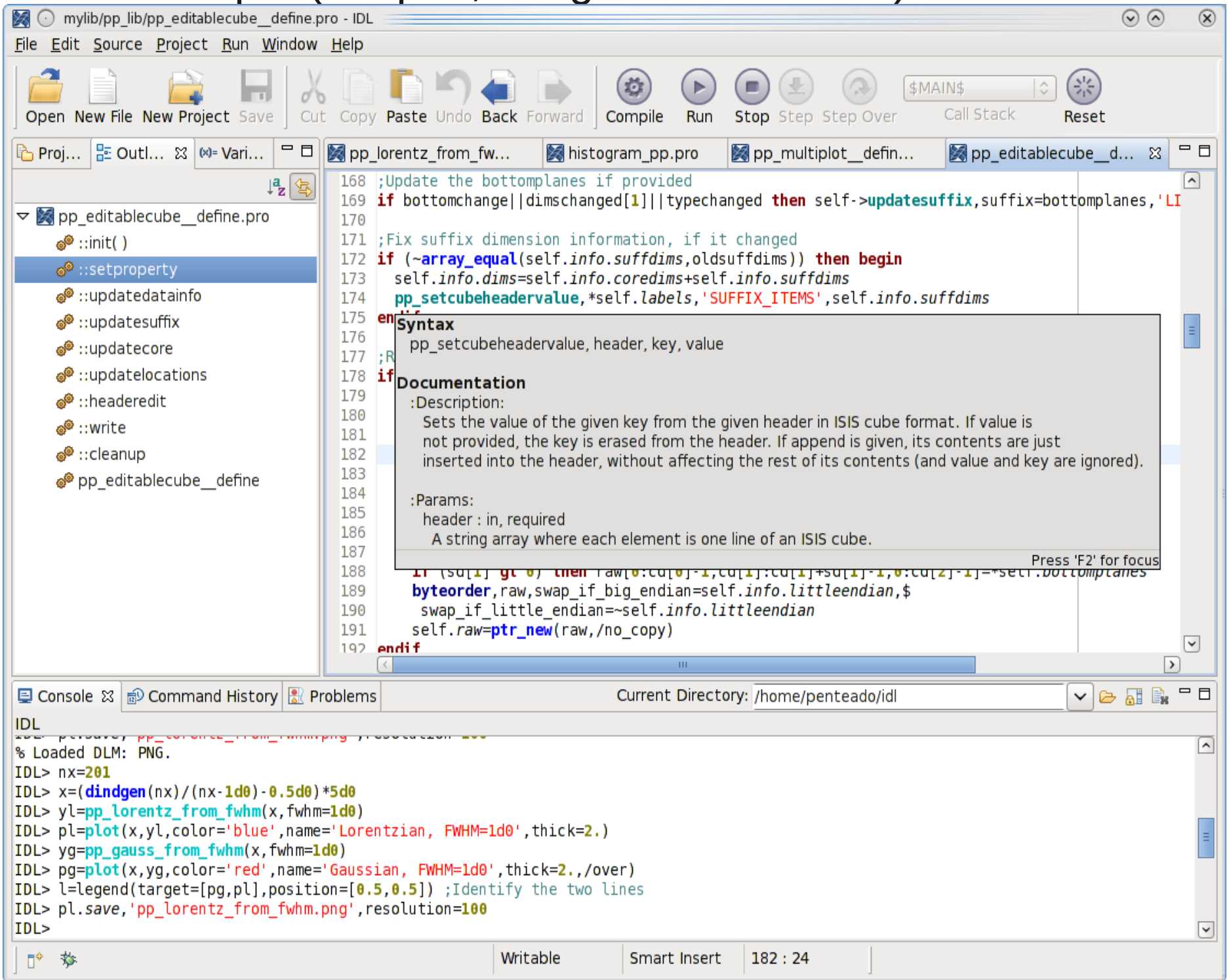
- Solaris Studio (antigo Sun Studio) - Linux, Unix, C, C++, Fortran

- C++ Builder (antigo Borland C++) - Windows, C, C++ (comercial)

Os editores nas IDEs costumam ser os mais capazes, por conhecer as linguagens:

- **S*yntax highlighting*** (cores diferentes para elementos diferentes, marcação de parênteses).
- **Ajuda de contexto** - acesso direto à documentação / código-fonte pelas ocorrências de nomes de rotinas no código, *auto-complete.*
- **Integrados ao debugger** – criação e uso de pontos de parada, consulta de valores de variáveis (mostrado adiante).
- Em algumas linguagens, **integrados ao compilador** – mostram ao vivo erros no código, de forma semelhante a corretores ortográficos (mas não só erros de sintaxe; podem mostrar coisas como tipos errados e erros nos argumentos de funções).

IDEs – exemplo (Eclipse, integrado ao IDL ≥7)



Debug – tipos de problemas

É comum que um programa não funcione da primeira vez que é usado.

Erros podem ser detectados em vários níveis. Começando do mais simples:

1 - Erros detectados pelo [editor / linha de comando / interpretador / compilador]:

- **erros de sintaxe**
- **nomes de [rotinas / variáveis / classes / tipos / campos de estrutura]**
- **tipos errados (em linguagens de tipos estáticos)**

Podem aparecer:

a) Interativamente, ao escrever o código.
b) Ao compilar / interpretar o código.

Simples de resolver; muitas vezes nem é considerado debug:

- **São sempre erros com o código e/ou ambiente de compilação/execução.** O programa nunca chegou a ser executado.
- **Não há a possibilidade de o erro ser pelo uso inadequado do programa (parâmetros/arquivos de entrada inválidos).**
- O [editor / linha de comando / interpretador / compilador] sabe onde está o problema.
- No caso de **(a)**, o erro é visto imediatamente quando é cometido (como com corretores ortográficos que marcam o texto ao ser escrito).

Em linguagens compiladas mais erros são detectáveis antes de executar o programa.

Em linguagens de tipos estáticos, ainda mais erros são detectados, porque mais coisas têm que ser especificadas.

Debug – tipos de problemas

A dificuldade maior costuma estar em erros que só se manifestam na hora da execução.

Começando dos mais simples:

2 - Programas que apresentam erros ao ser executados:
(**a**) Mostrando mensagens de erro (do programa, do interpretador, ou colocadas pelo compilador).
(**b**) Morrendo silenciosamente, sem informar erros.

Programas devem emitir mensagens de erro informativas, para que o usuário / programador tenha uma idéia do que deu errado. (**b**) é muito pior que (**a**).

Erros gerados pelo interpretador ou por código colocado pelo compilador, especialmente em linguagens dinâmicas **mostram com precisão (em que linha) acontece o problema**.

- Ex: tentativa de acesso um elemento além do final de um vetor.

Debug – tipos de problemas

3 - Programas que terminam de executar sem erros, mas que geram resultados errados.

Atenção ao procurar ajuda: muitos programadores os vêem da mesma forma que compiladores:

- Se o código é semanticamente correto, vão o considerar certo.
- Se ele faz ou não o que era pretendido é outro problema, que exige que o autor explique suas intenções.

Corrigir estes erros é muito facilitado por estruturar o programa:

- Pode-se testar unidades (rotinas) individuais
- Algumas unidades podem ser sabidamente corretas, eliminando locais possíveis para o erro
- Pode-se trocar unidades suspeitas por alternativas que façam o mesmo / algo semelhante
- É mais fácil identificar até que ponto o resultado estava certo, e onde começa a dar errado

4 – Programas que terminam de executar sem erros, e o usuário nem sabe se o resultado é certo ou não.

O primeiro passo é analisar os resultados: ou está certo, ou vai para a categoria (**3**).

Debug – tipos de problemas

A principal ferramenta para resolver problemas que se manifestam na hora da execução é o debugger (que normalmente é integrado aos IDEs).

A tarefa mais comum para resolver este tipo de problema é inspecionar as variáveis do programa antes de o erro acontecer

- Em linguagens dinâmicas, verificar não só o conteúdo, mas também tipo e dimensões)

Uma forma comum, mas lenta e trabalhosa é usar de *prints* para que o programa mostre informações sobre algumas variáveis em lugares de interesse no programa.

Debuggers permitem fazer inspeções de forma interativa, mais fácil, rápida e flexível.

Debug – o que faz um debugger?

Executa o programa, observando todas as variáveis encontradas.

Quando ocorre uma interrupção (erro, interativamente, ou por chegar a um ponto de parada (*breakpoint*)), o debugger disponibiliza:

- Conteúdo, tipo e dimensões **de todas as variáveis** em escopo.
- Avaliação / visualização / análise de expressões com as variáveis em escopo (apenas nas linguagens com interpretador é possível fazer qualquer coisa (como gráficos)).
- **Argumentos usados** para chamar a rotina atual (onde está parado o código).
- *Call stack* - **a seqüência de chamadas de rotinas que levou ao ponto atual**. Ex: interrupção na linha 17 da rotina read_parameter_file, chamada da linha 8 da rotina whatever, etc..
- **Navegação pelo *call stack*** - mudar o escopo entre pontos no *call stack*. Ex: para ver as variáveis da rotina que chamou a atual, para descobrir em que condições ela foi chamada.
- **Execução incremental do código** - dar passos graduais, para ver as mudanças geradas ao executar uma [linha / rotina / trecho de código] de cada vez.
- Em linguagens onde há um interpretador, alterar valores de variáveis ou o controle de fluxo, antes de retomar a execução.

Debug – o que faz um debugger?

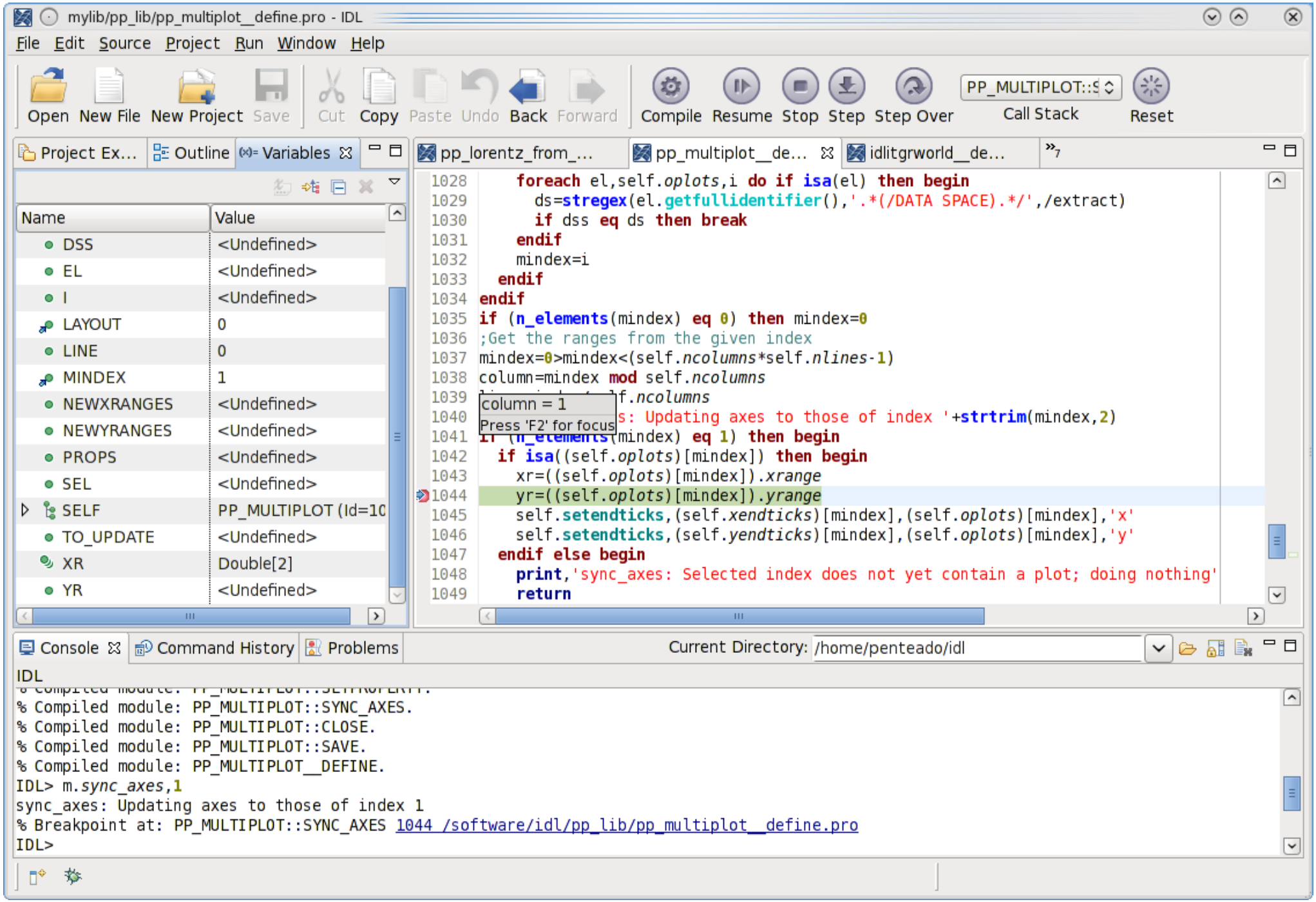
A principal vantagem de um debugger é permitir analisar “ao vivo” e interativamente a execução de um programa.

É como ver o que acontece dentro de uma máquina enquanto ela funciona, para ver onde está o problema, no lugar de tentar identificar o problema às cegas, com ela fechada.

Debuggers podem ter interface gráfica ou de linha de comando.

Normalmente são usados através de um IDE, que os integra ao contexto do código-fonte e do interpretador.

Debug – exemplo (Eclipse, integrado ao IDL ≥7)



Debug – programming pearls

http://users.erols.com/blilly/programming/Programming_Pearls.html

Don't debug standing up. It cuts your patience in half, and you need all you can muster,
Dave Storer, Cedar Rapids,Iowa

Testing can show the presence of bugs, but not their absence.
Edsger W. Dijkstra, University of Texas

Each new user of a new system uncovers a new class of bugs.
Brian Kernighan, Bell Labs

The first step in fixing a broken program is getting it to fail repeatably.
Tom Duff, Bell Labs

De *How To Ask Questions The Smart Way, http://www.catb.org/~esr/faqs/smart-questions.html*

Describe the goal, not the step

If you are trying to find out how to do something (as opposed to reporting a bug), begin by describing the goal. Only then describe the particular step towards it that you are blocked on.

Often, people who need technical help have a high-level goal in mind and get stuck on what they think is one particular path towards the goal. They come for help with the step, but don't realize that the path is wrong. It can take substantial effort to get past this.

Além disso, perguntar sobre os passos problemáticos pode levar à resposta ser literal (ao que foi perguntado, e não ao que levaria a onde a pessoa quer chegar).

Unit testing

It takes three times the effort to find and fix bugs in system test than when done by the developer. It takes ten times the effort to find and fix bugs in the field than when done in system test. Therefore, insist on unit tests by the developer.

Larry Bernstein, Bell Communications Research

***Unit tests* são programas que executam a rotina sendo testada, e comparam o resultado ao resultado certo (conhecido, embutido no teste).**

O objetivo é juntar casos que testem a maioria do código (se possível, todo), para verificar facilmente se ela funciona depois de uma edição.

Cada conjunto de testes verifica apenas uma unidade pequena do código (em geral, uma rotina), que é fácil de ser examinada e ter seus resultados testados.

Em ambientes profissionais, os *unit tests* são executados periodicamente (em geral, uma vez por dia) automaticamente, para manter um registro de que partes do software funcionam (em geral, associado ao de controle de versão).

São limitados em possibilidades para códigos que produzem resultados que não são variáveis (números, strings): difícil o fazer para código que gere uma interface gráfica, ou que produza uma visualização.

Unit testing – exemplos (IDL)

Um teste para a função findgen (que gera um array 1D de n inteiros, de 0 a n-1):

```
function findgen_ut::test_basic
  compile_opt strictarr
    a = findgen(5)
  assert, array_equal(a, [0.0, 1.0, 2.0, 3.0, 4.0]), 'Correct elements'
  return, 1
end
```

Uma função que gera um erro se a condição testada é falsa

O resultado do uso (incluindo de outros 3 testes não mostrados):

```
IDL> mgunit, 'findgen_ut'
"All tests" test suite starting (1 test suite/case, 4 tests)
   "findgen_ut" test case starting (4 tests)
      test_basic: passed
      test_error: passed
      test_fail_example: failed "Wrong number of elements"
      test_baderror: failed "Type conversion error: Unable to convert given
STRING to Long64."
    Results: 2 / 4 tests passed
  Results: 2 / 4 tests passed
```

Bibliotecas para *unit tests* existem para qualquer linguagem.

As comuns são as xxxUnit, derivadas da / inspiradas na **JUnit**: JUnit (Java), PyUnit, (Python), CppUnit, RUnit (R), fUnit (Fortran), mgunit (IDL).

Controle de versão

Sistema que contabiliza todas as mudanças em um conjunto de arquivos, possivelmente por vários usuários em computadores diferentes.

Introdução (software carpentry): http://software-carpentry.org/4_0/vc/intro/

É essencial para projetos em que mais de uma pessoa trabalha ao mesmo tempo:

- Mantém no repositório o conjunto de arquivos do projeto, preservando informações de cada atualização feita por cada usuário.
- Cada usuário tem um lugar onde obter a versão mais atualizada dos arquivos.
- Dá formas de lidar com conflitos (mais de um usuário editando a mesma parte ao mesmo tempo).

Mas é também útil para um único usuário:

- Mantém sincronizadas cópias em diferentes computadores.
- Não é necessário lidar com conflitos (não acontecem conflitos).
- **Dá um registro de todas as versões passadas**, como um *Undo* infinito.
  - Se algo deixa de funcionar, é possível procurar a versão velha onde ainda funcionava, sem ter que manter muitas cópias de arquivos com nomes diferentes.

Útil não só em programação. Se adequa bem a escrita de documentos em formatos como LaTeX, HTML (baseados em arquivos de texto).

Controle de versão

Principais sistemas (subversion (SVN), CVS, GIT) têm servidores e clientes (linha de comando e gráficos) para todas as plataformas comuns.

Uma opção conveniente, presente em muitas plataformas (escrita em Java) é o **SmartSVN** (GUI para SVN).

Difícil colocar um repositório em um computador do IAG para acessar externamente (e o astroweb não tem svn instalado).

Tutoriais: http://software-carpentry.org/4_0/vc/ e http://software-carpentry.org/3_0/version.html

Sumário

2 – Slides em http://www.ppenteado.net/pea/pea01_organizacao.pdf

- Organização de código
- Documentação
- IDEs
- Debug
- Unit testing

Sugestão: ver o vídeo de introdução a controle de versão em

http://software-carpentry.org/4_0/vc/intro/

Nota: Na página que contém todas as apresentações do tema

http://software-carpentry.org/4_0/vc/

Os vídeos (com áudio explicando o conteúdo das apresentações) são os links com os títulos das apresentações. Não tão óbvios como os links para os pdfs e ppts, que não têm áudio.

Ou a versão anterior (html) em

http://software-carpentry.org/3_0/version.html

Próxima aula

3 – Slides em http://www.ppenteado.net/pea/pea02_variaveis.pdf

- Tipos de variáveis
- Representações de números e suas conseqüências
- Ponteiros
- Estruturas
- Objetos

Algumas perguntas* que serão respondidas:

1)This may be a stupid question, but I really want to know why. Please, see below and explain. Thanks.
*IDL> print, 132*30*
3960
*IDL> print, 132*30*10*
-25936

2)There's something I can not explain to myself, so maybe someone can enlighten me?
*IDL> print, fix(4.70*100)*
469

**Perguntas reais, postadas no newsgroup do IDL.*