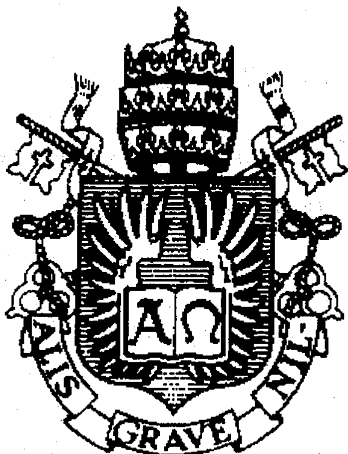


PUC

ISSN 0103-9741

Monografias em Ciência da Computação
nº 29/96

PG-06 Regras e Recomendações para Fluxos de Controle em Algoritmos
- Versão 1.01 -

Arndt von Staa
(Editor)

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, Nº 29/96
Editor: Carlos J. P. Lucena Setembro, 1996

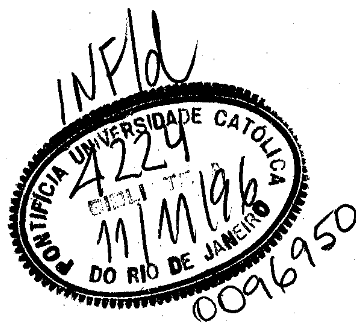
PG-06 Regras e Recomendações para Fluxos de Controle em Algoritmos *

- Versão 1.01 -

Arndt von Staa
(Editor)

* Trabalho patrocinado pelo Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

UC 67705-8



INF/d
4224
11/11/96
0096950

005.3
Pgs 531
PUC

Responsável por publicações:
Rosane Teles Lins Castilho
Assessoria de Biblioteca Documentação e Informação
PUC-Rio - Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 - Rio de Janeiro, RJ
Brasil
Tel +55-21-529 9386 Fax +55-21-511 5645
E-mail: biblio@inf.puc-rio.br www: http://www.inf.puc-rio.br

PG-06 Regras e recomendações para fluxos de controle em algoritmos

Versão 1.01

editor: A.v. Staa[1]
arndt@inf.puc-rio.br

Laboratório de Engenharia de Software
Departamento de Informática
Pontifícia Universidade Católica
22453-900 Rio de Janeiro,
Brasil

Setembro 1996

PUC-RioInf.MCC 29/96

Resumo

Neste documento definimos critérios de projeto de fluxos de controle em estruturas de algoritmos. Como conseqüência da adoção destes critérios espera-se que:

i. os custos de desenvolvimento e manutenção sejam reduzidos.
ii. a freqüência de erros de fluxo de controle seja minimizada.
iii. do ponto de vista do fluxo de controle dos algoritmos, a organização dos programas seja homogênea e independente do desenvolvedor.

Palavras chave: escopo de controle, escopo de efeito, fluxo de alteração de dados, fluxo de controle, funções geradoras, iteradores.

Abstract

In this document we define criteria for the control flow design of programs. By adopting these criteria we expect that:

i. a perceptible reduction of creation and maintenance cost.
ii. a perceptible reduction of control flow errors.
iii. a standard control flow organization of programs.

Keywords: data modification flow, flow of control, generating functions, iterators, scope of control, scope of effect.

[1] Trabalho apoiado por: CNPq, Bolsa de Pesquisador 300029/92-6, CENPES/Petrobrás, Itautec/ Philco

Histórico de evolução

PG-06 Regras e recomendações para fluxos de controle em algoritmos

Gestor: Laboratório de Engenharia de Software
Departamento de Informática, PUC-Rio

Arquivo: Estcntr1
Editado: 16 setembro, 1996
Impresso: 16 setembro, 1996

Documentos correlatos
PG-01 Regras e recomendações para a escolha de nomes em programas C e C++
PG-02 Regras e recomendações para o uso de constantes simbólicas em C e C++
PG-03 Regras e recomendações para a programação em C e C++

Versão V1.01

Editores:
Arndt von Staa (PUC-Rio)

Status: Em uso

Data homologação: 01/jul/1996
Data entrada em vigor: 01/jul/1996

Data de início da próxima revisão 01/jan/1997

Descrição de evolução
correções ortográficas e sintáticas

Descrição da retroação

Versão V1.00

Editores:
Arndt von Staa (PUC-Rio)

Status: Em uso

Data homologação: 01/mar/1996
Data entrada em vigor: 01/mar/1996

Data de início da próxima revisão 02/jan/1997

Descrição de evolução

Descrição da retroação

Créditos

Revisores

André Derraik	(TeCGraf PUC-Rio)	Versão 1.0
Claudio de Oliveira	(PUC-PR)	Versão 1.01
Geraldo Machado Costa	(LES PUC-Rio)	Versão 1.0
Lincoln Nobumiti Kanamori	(Itautec Philco)	Versão 1.0
Pedro Alexandre O. Giovani	(Itautec Philco)	Versão 1.0
Pedro Jorge E. Hübscher	(LES PUC-Rio)	Versão 1.0
Renan Martins Baptista	(CENPES)	Versão 1.0
Rosa Maria Ramalho Correia	(Itautec Philco)	Versão 1.0

Apoio

CENPES Petrobrás
CNPq
Itautec Philco

Marcas registradas e nomes de produtos

MS-DOS, Windows 3.xx e Windows NT são marcas registradas da Microsoft Corp
UNIX é marca registrada da ATT
OS/2 é marca da registrada da IBM
System 7 é marca registrada da Apple

Sumário

1. Objetivo 1
2. Conceitos 2
    2.1 Alinhamento da margem esquerda na representação linear 3
    2.2 Estruturas de repetição 4
    2.3 Estruturas de seleção 7
    2.4 Escopo de efeito e escopo de controle 8
3. Definição da Norma 10
    3.1 Regras para pseudo-instruções 10
    3.2 Regras para fluxo de alteração do conteúdo de estruturas de dados 10
    3.3 Regras para a serialização de comandos 11
    3.4 Regras para repetições 12
    3.5 Regras para seleções múltiplas 15
    3.6 Regras para escopo de controle e escopo de efeito 16
Bibliografia 17

1. Objetivo

Esta norma tem por objetivo definir critérios de projeto de fluxos de controle em estruturas de algoritmos.

Como conseqüência da adoção desta norma espera-se que:

- os custos de desenvolvimento e manutenção sejam reduzidos.
- a freqüência de erros de fluxo de controle seja minimizada.
- do ponto de vista do fluxo de controle dos algoritmos, a organização dos programas seja homogênea e independente do desenvolvedor.
- a manutenção de programas seja simplificada.

2. Conceitos

Entendemos por *fluxo de controle em algoritmos* a seqüência de execução dos comandos contidos em um algoritmo. Esta seqüência depende da serialização de ações e do uso de comandos de repetição e de seleção. É sabido que um mesmo problema pode ser resolvido por um número grande de diferentes implementações, todas produzindo resultados exatamente iguais. No entanto, várias destas implementações poderão ser difíceis de entender e de manter. Não somente isto, diversas formas de organizar código são mais propensas a erros do que outras. O objetivo deste conjunto de regras e recomendações é reduzir a incidência de tais problemas.

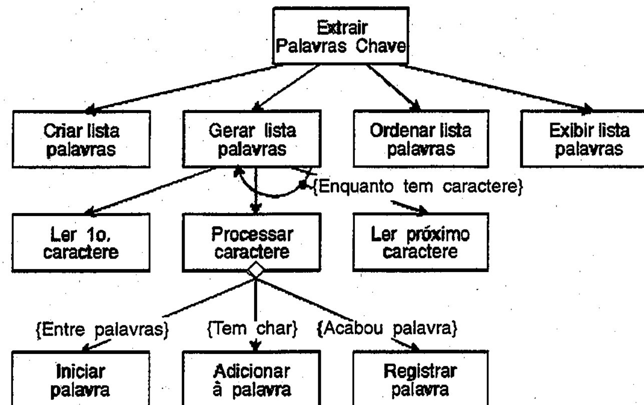


Figura 1. Ilustração de uma estrutura de algoritmo

Algoritmos possuem estrutura. Esta pode ser representada de diversas formas. A melhor forma sendo um *diagrama de estrutura de blocos*. A Figura 1 ilustra uma estrutura de algoritmo utilizando um diagrama de estrutura de blocos. Nesta forma cada bloco corresponde a uma *pseudo-instrução*[2]. O diagrama estrutura de blocos torna evidente a estrutura de decomposição do algoritmo. Cada sub-árvore corresponde a um sub-algoritmo. Ao ler os filhos descendentes de um mesmo nó pai, pode-se verificar facilmente se a implementação desse nó pai está completa ou não, mesmo que cada um dos nós filho seja, por sua vez, raiz de uma estrutura. Um algoritmo completo pode ser criado, agregando-se fragmentos de código, *anotações*, aos blocos da estrutura. Esta estrutura anotada é depois *linearizada*[3] segundo a seqüência de execução inerente a esta estrutura. Ao linearizar, os fragmentos de código e eventuais controles de execução associados aos blocos são externados à medida que os blocos vão sendo visitados em ordem de caminhamento prefixado pela esquerda. A Figura 2 apresenta o resultado da linearização da estrutura contida na Figura 1. O texto da Figura 2 tem exatamente o mesmo significado que a estrutura da Figura 1.

Estruturas de algoritmos podem ser representadas utilizando texto seqüencial endentado[4], ver Figura 2. Nesta forma cada comentário corresponde a uma pseudo-instrução. Na figura os textos redigidos em *itálico* devem ser substituídos por fragmentos de código implementando a especificação denotada pelo texto. Os fragmentos de código associados às pseudo-instruções aparecem imediatamente a seguir do comentário da pseudo-instrução. A forma seqüencial endentada é particularmente interessante quando se está utilizando editores de texto comuns para criar o projeto do algoritmo junto com o código fonte.

---

[2] No diagrama de *estrutura de blocos* cada bloco é a raiz de um sub-algoritmo. O diagrama de estrutura de blocos define a organização e a seqüência de execução do algoritmo. No diagrama de *estrutura modular*, encontrado freqüentemente em textos de análise estruturada, cada bloco corresponde a uma função. O diagrama de estrutura modular define as relações de chamadas de função existentes entre funções.

[3] *Linearizar*: é a operação de transformar em texto seqüencial uma estrutura e/ou diagrama contendo anotações em forma de fragmentos de texto.

[4] *Endentado*: é a organização do texto com recuos – dentes – da margem esquerda. A palavra *indentado* não existe em português.

```
// Extrair palavras chave
   // Criar lista de palavras chave
   // Gerar lista de palavras chave
      // Ler primeiro caractere
      while ( tem caractere )
      {
         // Processar caractere corrente
            if ( Caractere é delimitador
              && Palavra chave vazia      )
            {
               // Marcar o início de nova palavra chave
            } else if ( Caractere não é delimitador )
            {
               // Adicionar caractere à palavra
            } else if ( Caractere é delimitador
                     && Palavra chave não está vazia )
            {
               // Registrar a palavra chave
            } // if
         // Ler próximo caractere
      } // while
   // Ordenar a lista de palavras chave
   // Exibir a lista de palavras chave
```

Figura 2. Estrutura de algoritmo linearizado

Para que estruturas de blocos sejam instrumentos de projeto detalhado úteis, é necessário dispor-se de ferramentas para o projeto de algoritmos que permitam projetar estruturas, associar fragmentos de código às estruturas e, posteriormente, linearizá-las, produzindo arquivos contendo código fonte compilável.

2.1 Alinhamento da margem esquerda na representação linear

A representação linear reflete a estrutura do algoritmo por intermédio da endentação. Como esta representação contém código, o texto pode se tornar bastante longo, dificultando o entendimento da estrutura do algoritmo, principalmente no que toca as partes mais abstratas, ou seja mais altas do projeto estruturado.

Cada pseudo-instrução corresponde a um comentário. O nome da pseudo-instrução deve refletir a intenção desta operação. Desta forma, a pseudo-instrução torna-se um comentário que efetivamente adiciona informação à porção de código sob seu controle. Devido à endentação, este código é facilmente identificável. Ao ler o programa contendo pseudo-instruções como comentários, fica simples determinar o que o código faz e se está correta e completamente implementado.

Na norma *PG-03 Regras e recomendações para a programação em C e C++* são estabelecidas as regras de endentação a serem seguidas. A seguir detalharemos como essa norma deve ser aplicada ao criar uma representação linear.

A correta endentação reflete a estrutura do algoritmo. Para tal

i. todas as pseudo-instruções filhas de uma mesma pseudo-instrução mãe, estarão alinhados na mesma margem esquerda.
ii. a margem esquerda das pseudo instruções filhas fica 3 caracteres à direita da margem esquerda do pseudo-instrução mãe.
iii. cada pseudo-instrução deve estar alinhada à esquerda na sua margem de endentação,
iv. o fragmento de código correspondente a uma pseudo-instrução é endentado para a direita relativo à margem desta pseudo-instrução.

Desta forma, os comentários correspondentes a pseudo-instruções estarão bem destacados do código, permitindo que se consiga observar a estrutura do algoritmo. Mesmo não existindo código de controle, as pseudo-instruções filhas de um mesmo nó pai são endentadas com relação a este pai. Por exemplo, a operação *"Ler primeiro caractere"* é uma das operações que implementa a operação *"Gerar lista de palavras chave"*. A ope-

ração *"while (tem caractere)"* é irmã da operação *"Ler primeiro caractere"*, ficando, pois, no mesmo nível de endentação que esta. As duas operações filhas controladas pelo *"while"* são endentadas como de hábito.

2.2 Estruturas de repetição

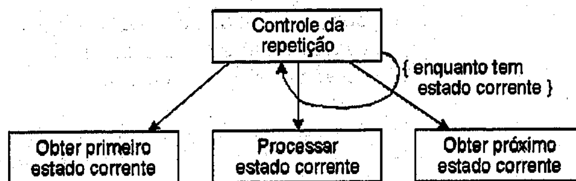


Figura 3. Estrutura de repetição

Ao projetar algoritmos são utilizadas diversas estruturas de controle tais como *if-then-else*, *switch-case*, *for* e *while*. Uma grande parcela dos problemas de projeto está diretamente relacionada ao uso incorreto ou incompleto destas estruturas de controle. Por exemplo, em repetições é freqüente inicializar o estado de forma incorreta ou incompleta, esquecer de avançar para o próximo estado, ou terminar incorretamente. Através de critérios de projeto bem definidos pode-se reduzir em muito a freqüência destes problemas ao criar ou manter algoritmos.

A Figura 3 mostra uma estrutura de algoritmo envolvendo uma repetição. Esta estrutura contém:

- um *controle* da repetição, na figura: o bloco *Controle da repetição*
- uma *preparação das condições iniciais* da repetição, na figura: o bloco *Obter primeiro estado corrente.* Quando o fragmento de código de preparação for simples, ele pode ser associado diretamente ao bloco controle da repetição.
- um *corpo* da repetição, na figura: os blocos *Processar estado corrente* e *Definir próximo estado corrente.* Podem existir tantos blocos de corpo quantos se queira. Cada um deles pode ser raiz de uma estrutura tão profunda quanto for necessário.

A execução de uma repetição efetua o corpo repetidas vezes para uma seqüência de 0 ou mais estados. Para cada iteração está definido um *estado.* São exemplos de indicadores de estados:

- o valor do índice de uma variável de contagem, usado tipicamente em comandos *for* ao percorrer um vetor.
- o valor de um ponteiro apontando para um elemento de uma lista, ao percorrer uma lista encadeada.
- o restante – *"tail"* – de uma lista ao processar uma lista de forma recursiva.
- a posição do cursor de leitura em um arquivo seqüencial.
- a posição do cursor de entrada de dados exibido no vídeo.
- o conjunto de elementos de um vetor ordenado delimitados pelos valores *Inf* e *Sup* em um algoritmo de pesquisa binária. Este algoritmo se encontra na Figura 4.

Deve estar claro que o estado não é necessariamente equivalente a um valor observável, por exemplo a posição do cursor de leitura em um arquivo seqüencial não é um valor explicitamente definido. Tampouco o estado será necessariamente um valor único. Por exemplo, no algoritmo de pesquisa binária o estado é um subconjunto do conjunto no qual estamos procurando o valor, ver Figura 4. Na leitura seqüencial de arquivos, o estado é dado pelo cursor de leitura e pelo conteúdo do *buffer* de leitura. Este contém sempre o valor do registro anterior à posição do cursor de leitura. Por outro lado podem estar associados um ou mais valores a um estado. Por exemplo:

- ao processar um vetor *A*, o estado é o índice *i* e o valor neste estado é *A[i]*.
- ao processar uma lista, o estado é o ponteiro *pCorr* e o valor neste estado é **pCorr*.
- ao processar um arquivo seqüencial, o valor é o conteúdo do *Buffer* lido, onde *Buffer* recebe um valor sempre que se progride para um novo estado. Ou seja, sempre que se efetua uma leitura, o valor de *Buffer* é atualizado com o registro lido e o cursor de leitura é posicionado antes do próximo registro a ler.

```
// Procurar um determinado valor em um conjunto ordenado
//    utilizando o algoritmo pesquisa binária
// Inf e Sup - delimitam o domínio do conjunto no qual é
//             realizada a pesquisa

   Inf = ObterLimInf( ) ;
   Sup = ObterLimSup( ) ;
   while ( Inf <= Sup )
   {

      // Verificar se encontrou
         Meio      = ( Inf + Sup ) / 2 ;
         ValorCorr = ObterValor( Meio ) ;
         if ( ValorProc == ValorCorr )
         {
            break ;
         } // fim if

      // Definir próximo intervalo
         if ( ValorProc < ValorCorr )
         {
            Sup = Meio - 1 ;
         } else {
            Inf = Meio + 1 ;
         } // fim if

   } // fim while
```

Figura 4. Algoritmo de pesquisa binária

Vamos ilustrar o conceito de estado de uma forma mais abrangente. Considere o algoritmo de pesquisa binária apresentado na Figura 4. Neste algoritmo é procurado um determinado valor em um conjunto ordenado de valores. Sejam `Inf` e `Sup` respectivamente os índices do primeiro e do último elemento do subconjunto de elementos no qual se está procurando. Note que neste exemplo o estado não é um valor simples, mas sim é o subconjunto de todos os elementos contidos entre os elementos acessíveis por `Inf` e `Sup` inclusive. O fato deste subconjunto estar vazio é caracterizado pela condição `Inf` > `Sup`.

Cabe enfatizar que neste exemplo a estrutura de dados utilizada para implementar o conjunto não está determinada. Conseqüentemente este algoritmo funcionará corretamente independentemente do fato do conjunto ser implementado por um vetor, por uma lista, ou por outra estrutura de armazenamento qualquer. A organização da implementação do conjunto está *encapsulada*[5] na função *ObterValor*. A implementação do conjunto deve assegurar que as seguintes *assertivas* valham:

- Sempre que *ObterLimInf( )* <= *i* <= *ObterLimSup( )*, *ObterValor(i)* retornará o i-ésimo valor existente no conjunto. *ObterLimInf( )* retorna o menor *i* correspondente a um elemento válido do conjunto e *ObterLimSup( )* retorna o maior *i* correspondente a um elemento válido do conjunto.
- o conjunto é parcialmente ordenado de modo que o valor de *ObterValor(i)* <= *ObterValor(j)* para todos os $i < j$.
- o conjunto é denso, ou seja, se $i<j$, então o conjunto contém todos os elementos acessíveis por i, $i+1$, $i+2$, ..., $j-1$, j.

Qualquer repetição envolve os seguintes itens:

i. tipo *tpEstado* o tipo do estado. Pode ser um tipo composto, neste caso procure definir o tipo como uma estrutura – `struct`. Cada iteração opera com um determinado valor corrente deste estado. Pode ser um valor composto, por exemplo os valores `Inf` e `Sup` do exemplo acima. Pode ser ainda um valor contextual, por exemplo o cursor de leitura de um arquivo seqüencial.

---

[5] *Encapsulado*: diz-se que uma estrutura de dados está encapsulada se a sua organização não necessita ser conhecida para que possa ser corretamente utilizada. Estruturas de dados encapsuladas são manipuladas estritamente por intermédio de funções.

ii. um valor constante *ESTADO_NIL*, indicando um estado não definido. A constante é do tipo *tpEstado*. Em alguns casos, ao invés de uma constante, existirá uma relação envolvendo os valores do estado, que, se falsa, indica um estado não definido, por exemplo `Inf <= Sup` no exemplo acima. Cabe observar que o nome *Estado* usado no texto, deve ser trocado por um nome que descreva o significado do estado, conseqüentemente da repetição.
iii. um ou mais tipos de valor no estado *tpValor*. Pode existir mais de um tipo de valor. Por exemplo, ao invés de retornar uma estrutura de dados correspondente a um valor composto – por exemplo um registro de arquivo –, pode ser mais interessante criar diversas funções de acesso, uma para cada campo do valor composto. Cada uma dessas funções de acesso retorna um valor de um determinado tipo.
iv. uma ou mais funções *ObterValor(...)*. Retorna um dos valores associado ao estado corrente. Podem existir mais de uma dessas funções, ver item 3 acima.
v. função *DefinirPrimeiro(...)* define o estado corrente inicial do processamento da repetição.
vi. função *DefinirPróximo(...)* define o estado corrente a seguir do estado corrente atual.
vii. função *Existe(...)* retorna o valor *TRUE* enquanto não se tiver esgotado o conjunto de estados, retornando *FALSE* caso contrário.

Chamaremos de *função geradora*[6] o conjunto de tipos, funções e constantes descrito acima. Cabe salientar que nem todas as repetições definem todos estes elementos. Por exemplo, em um gerador de números aleatórios, usualmente não se define a função *Existe(...)*. Por outro lado, como já foi mencionado, dependendo da repetição, alguns destes elementos podem ter diversas implementações.

A Figura 5 ilustra o uso de uma função geradora em um esquema de algoritmo. Este esquema algoritmo será sempre o mesmo, independente da estrutura de dados utilizada para implementar a tabela. Diferentes estruturas de dados terão diferentes formas de implementar os componentes da função geradora. Na figura os elementos da função geradora estão representados em *itálico*.

Para assegurar que o processamento de uma repetição possa estar correto, deve-se assegurar que:

- antes de ativar o corpo da repetição, o estado corrente esteja sempre completamente definido;
- ao retornar do corpo para o controle da repetição, o estado corrente corresponda ao próximo estado a ser processado;
- cada estado seja um estado diferente dos demais;
- o número de estados gerados seja finito.

Caso estas recomendações não sejam seguidas, deve-se redigir explicitamente o por quê da exceção. Por exem-

```
// Esquema de algoritmo para pesquisa em tabela
   tpRefElemTabela Achou ;
   tpRefElemTabela Corrente ;

   Achou    = ELEM_TABELA_NIL ;
   Corrente = DefinirPrimeiro( Tabela ) ;
   while ( Existe( Tabela , Corrente ))
   {
      if ( ObterValor( Tabela , Corrente ) == ValorProcurado )
      {
         Achou = Corrente ;
         break ;
      } else {
         Corrente = DefinirProximo( Tabela , Corrente ) ;
      } // if
   } // while
// Se Achou != ELEM_TABELA_NIL, foi encontrado o valor procurado
```

Figura 5. Algoritmo genérico utilizando uma função geradora

plo, em uma aplicação Windows desenvolvida em C, o ciclo de mensagens – "*message loop*" – deve poder operar indefinidamente até que uma mensagem do gênero "fim de execução" seja recebida. Potencialmente, portanto, este ciclo poderá nunca terminar, uma vez que não está no domínio do programa gerar as mensagens a serem processadas. Nestes casos é importante demonstrar que existe alguma ação prevista do usuário que permita interromper uma seqüência de ações potencialmente infinita. Esta ação deve estar explicitada no código. Cabe salientar que uma repetição normal – isto é, sem exceções – precisa sempre assegurar que as 4 condições acima sejam válidas, de outra forma não há como possa executar corretamente.

Na Figura 6 apresentamos uma estrutura parcial de um programa que gera extratos a partir de um arquivo de movimentos seqüencial, ordenado segundo chave de cliente. Nesta figura existem 3 repetições. Destas somente uma corresponde a uma função geradora completa. As demais possuem elementos implicitamente definidos. As repetições são:

- *cliente*, corresponde a uma partição do arquivo de movimentos, em que cada grupo de movimentos de um mesmo cliente é tratado como uma unidade. Obter o primeiro cliente corresponde, portanto, a obter o primeiro movimento. Obter próximo cliente é uma função implícita, uma vez que ocorre como conseqüência de ter-se lido a chave de um novo cliente ao obter o próximo movimento.
- *página*, corresponde a uma partição do conjunto de registros de um mesmo cliente em *n* páginas, onde as primeiras *n-1* páginas estão completas e a última pode estar parcialmente completa. Obter próxima página ocorre quando se esgota a página sendo impressa, ou quando muda o cliente corrente.

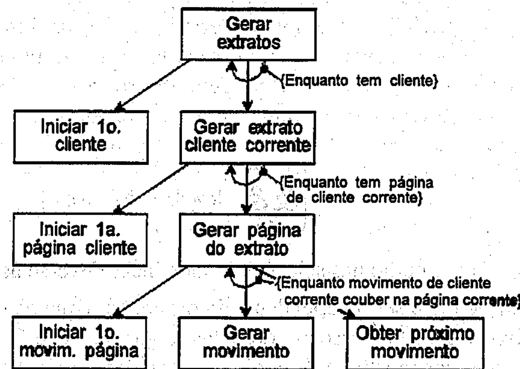


Figura 6. Ilustração de elementos de função geradora implícitos

- *movimento*, este é um arquivo seqüencial para o qual todas as funções da função geradora estão definidas.

2.3 Estruturas de seleção

Estruturas do tipo *if-then-else* e *switch-case*, também geram problemas quando utilizadas de modo descuidado. É freqüente fazer-se seleções sucessivas utilizando construções do gênero *if-then-else-if-then-else-...-else*. Este tipo de construção é denominada uma *seleção heterogênea*, enquanto que uma seleção do tipo *switch-case*, ou do tipo *TRY-CATCH*, é denominada uma *seleção homogênea*.

Uma seleção heterogênea em C ou C++ tem a forma:

```
if        ( cond1 )
{
   // fragmento de código selecionado se cond1 for verdadeira
```

[6] Na literatura de programação orientada a objetos, a função geradora, é chamada de "*iterator*", e é, tipicamente, composta pelas funções: *DefinirPrimeiro(...)*, *DefinirPróximo(...)* e *Existe(...)*. Aqui estamos utilizando uma definição mais completa.

```
} else if ( cond2 )
{
   // fragmento de código selecionado se cond1 for falsa
   //      e cond2 for verdadeira
} else if ( cond3 )
{...
} else if ( condn )
{
   // fragmento de código selecionado se cond1 até condn-1 forem
   //      todas falsas e condn for verdadeira
} else {
   // fragmento de código selecionado se cond1 até condn forem
   //      todas falsas, condição default
} // if
```

Na seleção heterogênea acima será escolhido o primeiro fragmento de texto para o qual a condição seja verdadeira, mesmo que outra condição a seguir também seja verdadeira. Ou seja, o i-ésimo fragmento de código será selecionado caso for verdadeira a expressão:

$$\left(\mathop{\&\&}_{j=1}^{i-1} \; !cond_j \right) \; \&\& \; cond_i \; , \; i \text{ é mínimo}$$

Nesta forma cada elemento selecionável depende da posição em que ocorre. Se durante uma etapa de manutenção ocorrer uma mudança na ordem, o programa poderá deixar de funcionar. A condição "*i é mínimo*" é necessária, uma vez que mais de uma expressão poderia ser verdadeira para um determinado conjunto de dados. Para que não ocorra dependência da posição da seleção, deve-se procurar utilizar a *forma completa* de seleção heterogênea. Neste caso o *i*-ésimo fragmento de código será selecionado se e somente se for verdadeira a expressão:

$$\left(\mathop{\&\&}_{j=1}^{n,\, j \neq i} \; !cond_j \right) \; \&\& \; cond_i$$

Tanto nas seleções heterogêneas como nas seleções homogêneas deve-se assegurar que:

i. o resultado da seleção deve ser independente da ordem com que os seletores forem redigidos. No caso de uma construção `switch` isto é assegurado pelas linguagens C e C++. No entanto, numa seleção heterogênea, isto pode provocar repetição de código de condição, veja a fórmula da condição completa acima. Mesmo assim, para maior legibilidade e manutenibilidade, é recomendável redigir as condições na forma completa. Uma grande parte dos compiladores otimizadores modernos fatora expressões comuns, conseqüentemente, eventuais duplicações de um mesmo código serão automaticamente eliminadas pelo compilador.

ii. ao terminar a execução de cada fragmento de código selecionável, também deve ser terminada a execução da seleção como um todo. No caso de uma construção `switch` isto implica que cada fragmento de código selecionável deve terminar com um `break` ou `return`, ver norma *PG-03 Regras e recomendações para a programação em C e C++.*

iii. deve existir sempre um fragmento de código que capture condições não previstas. No caso de `switch` isto corresponde a existência de um seletor `default`, enquanto que no caso de seleção heterogênea isto corresponde à existência de um `else` final selecionando a condição *default*. Este `else` final usualmente não corresponderá a uma das condições normais válidas. Em ambas as formas, caso a condição *default* não seja uma condição válida, o fragmento de código *default* deve gerar uma interrupção do processamento correspondente à percepção de um erro de execução.

iv. deve-se assegurar que não foi esquecida nenhuma condição ou caso específico.

v. deve-se assegurar que cada seleção seja mutuamente exclusiva com todas as demais seleções. Isto é particularmente importante quando se estiver codificando seleções heterogêneas com controle posicional – a primeira forma das apresentadas acima. Somente assim pode-se evitar que ocorram casos em que mais de uma condição possa ser avaliada para `TRUE`.

2.4 Escopo de efeito e escopo de controle

Denomina-se *escopo de controle* a porção de código controlada por uma determinada condição [Gane79]. Na Figura 7 ilustramos este conceito. O escopo de controle da condição $Cond_1$ é a subestrutura X quando seleciona *Raiz da seleção 1*, e é a subestrutura Z quando seleciona *Outra Seleção 1*. O escopo de controle da condição $Cond_2$ é toda a subestrutura Y com raiz em *Raiz da seleção 2*.

Denomina-se *escopo de efeito* o conjunto de porções de código controladas pela mesma avaliação de uma determinada condição. Na Figura 7 o escopo de efeito da condição $Cond_1$ é formado pelas duas subestruturas X e Z. Já o escopo de efeito da condição $Cond_2$ é a subestrutura Y com raiz em *Raiz da seleção 2*.

Ao projetar algoritmos, deve-se procurar assegurar que o escopo de efeito seja igual ao escopo de controle. Isto é particularmente importante quando as expressões de controle forem avaliadas em funções ou em métodos diferentes. Por exemplo, na Figura 7 a subestrutura com raiz em *Controlar ação B* poderia estar implementada em uma função, enquanto que o restante estaria implementado em outra função. Neste caso a condição $Cond_1$ será avaliada em duas funções diferentes. Este tipo de situação freqüentemente leva a erros de compreensão e de manutenção do programa.

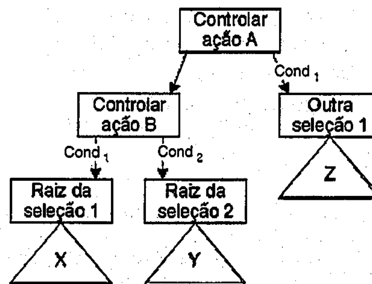


Figura 7. Ilustração de escopo de efeito e escopo de controle

3. Definição da Norma

3.1 Regras para pseudo-instruções

Regra 1:	Ao representar algoritmos utilizando texto seqüencial, a estrutura do algoritmo deve ser refletida através de comentários definindo pseudo-instruções.
Regra 2:	Cada pseudo-instrução deve identificar a intenção principal da estrutura ou fragmento de código sob seu controle.
Regra 3:	Cada função ou método deve implementar completamente uma única abstração.
Recom. 4.	Evite quebrar um programa em um conjunto muito grande de funções.
Recom. 5:	Fragmentos de código vinculados a uma pseudo-instrução devem ter, em média, de 4 a 8 linhas.

Ao programar em C ou C++ é costumeiro particionar o programa em um número grande de pequenas funções. No entanto, tem sido observado que o particionamento físico pouco contribui para a melhoria do entendimento de um programa. O particionamento físico é decorrência de se querer manter pequeno – por exemplo menor do que 25 – o número de linhas de código de uma função. As condições a seguir são sintomáticas de funções criadas por particionamento físico:

- a função ou método é encapsulado (não é público nem externo);
- existe somente uma chamada para esta função ou método em todo o programa.

Além de quebrar a visibilidade do algoritmo, o particionamento físico contribui para a geração de código morto como conseqüência de manutenções sucessivas. Dificilmente um programador vai controlar se uma determinada função ou método é efetivamente chamado de um ou mais lugares. Tendo em vista que uma função pode ser referenciada por um ponteiro do tipo função, torna-se difícil, se não impossível, o compilador fazer este controle. Ao invés de particionar fisicamente, é mais indicado estruturar o algoritmo utilizando pseudo-instruções. Neste caso cada função implementa completamente uma abstração ou conceito bem definido, além disso tenderá a ser chamada de diversos lugares.

Funções ou métodos devem sempre implementar uma única abstração bem definida. Desta forma se atinge coesão funcional [Gane 79].

A estrutura do algoritmo de cada função ou método deve ser refletida através de pseudo-instruções. O alinhamento dos comentários deve seguir as regras estabelecidas na norma: *PG-03 Regras e recomendações para a programação em C e C++.*

As finalidades de pseudo-instruções são:

- contribuir para tornar visível a estrutura do algoritmo;
- contribuir para tornar o código mais inteligível;
- facilitar a localização dos pontos do código que devem ser alterados durante um passo de manutenção.

Para aumentar a legibilidade deve-se ter uma pseudo-instrução para poucas linhas de código. No entanto, um número excessivo de pseudo-instruções aumenta em muito o custo de redação do algoritmo, nem sempre contribuindo para uma maior inteligibilidade. A prática tem mostrado que uma média razoável está entre 4 e 8 linhas de código, porém, nada impede que existam fragmentos de código bem maiores do que 8 linhas, ou fragmentos formados por somente 1 ou 2 linhas de código. É perfeitamente tolerável, ainda, que os fragmentos contenham estruturas de controle do tipo *if*, *while* ou semelhante. Deve-se evitar que a excessiva atomização da estrutura torne difícil a leitura ou a manipulação do código como um todo. Por outro lado, deve-se evitar que o tamanho excessivo de um fragmento de código dificulte a compreensão do propósito e funcionamento de todas as suas partes. Desde que, para entender o propósito e o funcionamento do algoritmo contido no fragmento de código, seja suficiente o próprio fragmento de código e a correspondente pseudo-instrução, o tamanho do fragmento de código estará, em princípio, bom.

3.2 Regras para fluxo de alteração do conteúdo de estruturas de dados

Recom. 6: Somente ative código de alteração do conteúdo de estruturas de dados:

1. depois que todos os espaços de dados necessários tiverem sido alocados;

2. depois que todos os dados necessários tiverem sido adquiridos e devidamente validados;
3. sem possível interferência do usuário ou da plataforma.

Recom. 7: Desenvolva programas de modo que possam ser migrados para plataformas preemptivas[7].

Uma parcela significativa de programas adquirem dados, validam estes dados, alocam espaços de dados para poder armazenar estes dados e incorporá-los a alguma estrutura de dados em memória ou em arquivo. Alterações do conteúdo de estruturas de dados freqüentemente são difíceis de serem desfeitas, podendo comprometer o correto funcionamento do programa, caso não sejam realizadas de forma completa. Por exemplo, se durante a inserção de um elemento em uma lista encadeada, por alguma razão a operação for cancelada antes de se completar a inserção, o resultado pode ser uma estrutura mal formada – por exemplo: ponteiros apontando para espaços de dados inexistentes – ou pode ser uma estrutura semanticamente ilegal – por exemplo: conteúdo do elemento da lista não é válido –.

As estruturas de dados em questão podem ser vetores, tabelas, listas, conjuntos de campos a serem exibidos em um diálogo, ou até mesmo arquivos. Ou seja, mesmo quando se estiver tratando de estruturas de dados pequenas, simples ou de âmbito local, é recomendável que se projete cuidadosamente o fluxo de operações realizadas para alterar o conteúdo da estrutura de dados.

Ao manipular estruturas de dados, por mais simples que sejam, sugere-se operar segundo o esquema a seguir:

i. adquirir os dados.
ii. validar os dados.
iii. validar as possíveis conseqüências da alteração da estrutura de dados. Por exemplo, ao tentar inserir um elemento, a memória disponível pode estar esgotada. Isto significa que devemos alocar todos os espaços antes de alterar a estrutura de dados.
iv. somente quando *todos os dados* tiverem sido adquiridos e estiverem válidos tanto do ponto de vista dos dados propriamente, quanto da estrutura de armazenamento, deve-se proceder a alteração da estrutura de dados.

Durante a alteração do conteúdo da estrutura de dados não devem ser realizadas operações de alocação de memória, de aquisição de dados, ou de validação de dados. Tampouco a operação de atualização poderá ser interativa. Ao utilizar sistemas operacionais pre-emptivos, é necessário assegurar que a porção de código que realiza a atualização da estrutura de dados seja não interrompível.

3.3 Regras para a serialização de comandos

Regra 8: Agregue e ordene seqüências de comandos ou pseudo-instruções de acordo com o seu significado.

Usualmente seqüências de comandos podem ser redigidas em diversas ordens, todas elas produzindo o mesmo resultado. No entanto, deve-se escolher, uma ordenação tal que os comandos acabem organizados segundo um significado definido. Por exemplo:

```
// Seqüência a evitar
   Gravar BufferA em Saída
   Gravar BufferB em Saída
   Ler registro de ArquivoB para BufferB
   Ler registro de ArquivoA para BufferA

// Prefira a seguinte seqüência
   // Transferir A para Saída
      Gravar BufferA em Saída
      Ler registro de ArquivoA para BufferA
   // Transferir B para Saída
```

[7] Uma plataforma preemptiva permite diversos processos operarem concorrentemente, sendo que não existe restrição quanto aos pontos de troca de contexto. Exemplos de plataformas preemptivas são: UNIX, OS/2, System 7, Windows NT. Nestas plataformas é necessário sincronizar processos que compartilhem espaços de dados.

```
        Gravar BufferB em Saída
        Ler registro de ArquivoB para BufferB
```

Neste exemplo, os comandos foram agregados em torno de *BufferX*, uma vez que cada agregado pode ser entendido como uma pseudo-instrução completa. Note que na seqüência original não é possível identificar estas pseudo-instruções. A possibilidade de identificar novas pseudo-instruções é particularmente interessante pois facilita encontrar agregados que possam ser reutilizados. No exemplo acima, pode-se criar uma função que grava o buffer e lê um novo valor. Após esta nova transformação, o texto ficaria assim:

```
// Transferir A e B para Saída
      Transferir( ArquivoA, BufferA, ArquivoSaida) ;
      Transferir( ArquivoB, BufferB, ArquivoSaida) ;

// A função Transferir seria semelhante a:
   void Transferir( FILE * ArquivoEntra ,
                    char * BufferEntra  ,
                    FILE * ArquivoSai    )
{
   Gravar BufferEntra em ArquivoSai
   Ler registro de ArquivoEntra para BufferEntra
}
```

Regra 9. Seqüências de comandos parentetisantes devem ser redigidos obedecendo ao correspondente aninhamento.

Uma par de operações é dito ser parentetizante quando delimita início e fim de atividades. São exemplos: abrir e fechar arquivos, criar e destruir objetos, empilhar e desempilhar elementos, enfileirar elementos e retirar elementos da fila.

```
// Ordenação de código a evitar
   Abrir arquivo A
   Abrir arquivo B
      ...
...Fechar arquivo A
...Fechar arquivo B

// Ordenação de código segundo a regra
   Abrir arquivo A
   Abrir arquivo B
      ...
...Fechar arquivo B
...Fechar arquivo A
```

3.4 Regras para repetições

Recom. 10: Sempre que o estado de uma repetição for complexo, preceda a repetição com um comentário definindo todos os componentes da função geradora.

Regra 11: Os componentes de uma função geradora a serem definidos são:

1. o tipo *tpXXX* o tipo do estado da repetição.
2. a constante *XXX_NIL*, ou relações entre os valores do estado, indicando que o estado está indefinido.
3. um ou mais tipos *tpValor*, tipos do valor no estado.
4. uma ou mais funções *ObterValor(...)*, cada uma retorna um dos valores associado ao estado corrente.
5. a função *DefinirPrimeiroXXX(...)*, define o estado corrente inicial do processamento do ciclo.
6. a função *DefinirPróximoXXX(...)*, define o estado corrente a seguir do estado corrente atual.
7. a função *ExisteXXX(...)*, retorna o valor *TRUE* enquanto não se tiver esgotado o conjunto de estados, retornando *FALSE* caso contrário.

Exceção 12: As funções mencionadas na regra anterior podem ser implementadas como fragmentos de código ao invés de como funções fechadas. Neste caso os fragmentos de código devem estar precedidos de um comentário identificando a função componente da função geradora.

Recom. 13: Um estado é simples somente se cair em uma das seguintes categorias:

1. é um valor inteiro simples.
2. é um ponteiro para uma lista seqüencial simples.
3. é o cursor de leitura ou de gravação de um arquivo seqüencial de texto.
4. é o cursor de leitura ou de gravação de um arquivo contendo registros uniformes e sendo lido ou gravado seqüencialmente.

Recom. 14: Caso um ou mais dos elemento da função geradora não sejam definidos, explicite o fato no comentário de descrição da função geradora, ou em um comentário específico caso a descrição não exista.

Recom. 15: Ao programar em C, defina estruturas – `struct` – para registrar os dados que definem estados de repetição compostos.

Recom. 16: Ao programar em C++, crie classes contendo as funções geradoras – *classes iteradoras* – vinculadas às classes que implementam estruturas de dados.

Regra. 17: O comentário de descrição de uma classe iteradora deve definir todos os itens da função geradora.

É amplamente reconhecido que uma repetição é a estrutura mais difícil de compreender e de assegurar estar correta. A explicitação do significado da repetição através da definição da função geradora, reduz em muito a dificuldade de sua compreensão. Além disso, obriga o redator a refletir mais aprofundadamente sobre a repetição, virtualmente eliminando erros de programação.

Em C++ pode-se criar classes iteradoras. Ao criar uma classe que implemente estruturas de dados complexas, tais como estruturas encadeadas, recomenda-se que sejam criadas classes iteradoras para percorrer estas estruturas. Podem ser criadas várias classes iteradoras para uma mesma estrutura de dados. Por exemplo, pode-se criar uma classe iteradora "caminhamento prefixado" e outra "caminhamento pós-fixado" para uma mesma estrutura de dados "árvore binária". Em C podem ser criados módulos com propriedades semelhantes.

Regra 18: Ao entrar no corpo de uma repetição, assegure que o estado corrente esteja completamente definido.

Regra 19: Ao retornar do corpo de uma repetição, assegure que o estado corrente da iteração a seguir seja um estado válido e que esteja completamente definido.

Exceção 20: Ao sair do corpo de uma repetição por ter atingido o objetivo deste ciclo, o próximo estado corrente pode estar indefinido.

O *corpo da repetição* é o código que está no escopo de controle da repetição. Assim, *retornar do corpo* significa retornar ao controle da repetição, e *sair do corpo* significa efetivamente sair do controle da repetição. Ao iniciar a execução do corpo da repetição deve-se estar com toda a informação de controle completa e corretamente definida. De outra forma, torna-se necessário incluir código de validação do estado corrente, aumentando a dificuldade de compreensão da repetição, bem com o risco de erro devido a ter-se esquecido de incorporar a validação.

Na Figura 8 ilustramos como transformar uma estrutura que não segue as regras de definição do estado da repetição, em uma estrutura que satisfaça estas regras. Tipicamente a transformação é feita criando uma função que define o próximo estado. Esta função, quando utilizada antes de iniciar a repetição, define o primeiro estado. Note que a redação do código transformado explicita a finalidade da repetição. Já o código antes da transformação é bem mais obscuro.

```
// Evite este tipo de estrutura
   while ( TRUE )
   {
      SelecaoMenu = ObterSelecao(...) ;
      if ( SelecaoMenu == FIM_MENU )
      {
         break ;
      }
      Processar seleção
   }

// Procure utilizar estruturas semelhantes a esta
   SelecaoMenu = ObterSelecao(...) ;
   while ( SelecaoMenu != FIM_MENU )
   {
      Processar seleção
      SelecaoMenu = ObterSelecao(...) ;
   }

// Ou semelhante a esta (máquina de estados)

   Continua = OK ;
   while ( Continua )
   {
      SelecaoMenu = ObterSelecao(...) ;
      Processar seleção
         // Torna Continua falso quando a seleção
         // processada levar ao estado final
   }
```

Figura 8. Transformando para atender às regras

Recom. 21: Assegure que cada estado seja diferente de todos os estados processados em iterações anteriores.

Recom. 22: A menos que seja óbvio, inclua um comentário na descrição do ciclo, explicando *porque* e *como* o ciclo termina de executar após um número garantidamente finito de iterações.

Recom. 23: Assegure que todas as repetições sejam capazes de executar corretamente 0 iterações e 1 iteração.

Qualquer repetição deve parar após um número finito de iterações. Para que isto aconteça, é necessário que cada novo estado seja diferente dos estados anteriores. Se não forem e o término da repetição não tiver sido alcançado, a mesma seqüência de estados vai ser repetida de novo, levando a um ciclo infinito. Note que, se o estado é composto, os valores de alguns dos elementos do estado podem ser iguais a valores anteriores, mas nunca *todos* os valores de um estado poderão ser exatamente iguais aos valores de um estado anterior.

Em alguns programas o término de execução depende de uma ação do usuário, por exemplo em sistemas operacionais o processamento é interrompido pelo usuário após o sistema ter atingido um estado de espera. Este tipo de término deve ser explicitado no comentário de descrição da correspondente função geradora. De modo semelhante, deve estar explicitado como interfaces de usuários são terminadas, caso elas permitam uma sucessão indefinida de ações. Por exemplo, ao programar um diálogo em Windows, deve estar claro quais todos os botões e eventos do diálogo que terminam a sua execução.

Para que um programa possa estar correto, é necessário – mas não suficiente – que as repetições executem corretamente 0 iterações, 1 iteração, e $n+1$ iterações, assumindo-se que tenha operado corretamente para n iterações[8]. Todas as repetições devem ser capazes de operar corretamente para 0 iterações, para 1 iteração, e para n

[8] Para demonstrar que uma repetição está correta, é necessário demonstrar que o ciclo opera corretamente e que ele pára após um número finito de iterações. Utiliza-se *indução matemática* para demonstrar que opera corretamente [LG86].

quaisquer iterações, mesmo que o número de iterações a efetuar seja uma constante, por exemplo: `for ( i= 0 ; i < 10 ; i++ )`. Ou seja, independentemente de como a repetição seja definida, deve-se sempre ser capaz de mostrar que o algoritmo funciona corretamente, considerando as hipóteses de um total de *0*, *1* e *n* iterações a serem executadas. Por exemplo, ao processar um vetor de *Dim* elementos, deve-se mostrar que o programa opera corretamente caso *Dim* seja *0*, e caso *Dim* seja *1*, mesmo que, na implementação, o valor de *Dim* seja uma constante. Ou seja, existe uma diferença entre um programa correto e uma utilização *razoável* deste programa. O programa estará correto somente se ele opera corretamente considerando-se as hipóteses *0*, *1* e *n*. No entanto, um uso razoável pode excluir os casos 0 e 1.

Recom. 24: Evite o uso de ciclos do gênero `do-while`.

Uma das propriedades de ciclos do tipo *do-while* é eles sempre operarem pelo menos uma vez. Na maioria das vezes, o corpo de um destes ciclos é iniciado sem que o estado no primeiro ciclo esteja completa e corretamente definido. Usados desta forma, ciclos *do-while* violam as regras de integridade de estado estabelecidas e, portanto, devem ser evitados. Na Figura 9 ilustramos como converter um algoritmo utilizando do-while em um algoritmo utilizando while.

```
// Forma em princípio válida para a redação de um do-while
   do
   {
      Compor / Recompor a mensagem
      Transmitiu = TransmitirMensagem(...) ;
         // somente agora o estado estará definido
   } while ( !Transmitiu ) ;

// Forma transformada, sem uso de do-while
   Transmitiu = FALSE ;
   while ( !Transmitiu )
   {
      Compor / Recompor a mensagem
      Transmitiu = TransmitirMensagem(...) ;
   }
```

Figura 9. do-while transformado para while

3.5 Regras para seleções múltiplas

Regra 25: Em seleções múltiplas, homogêneas ou heterogêneas, preveja sempre o caso *default*.

Regra 26: Nas expressões condicionais, não utilize funções que provoquem efeitos colaterais.

Recom. 27: Procure sempre utilizar o formato completo para as seleções em uma seleção heterogênea.

Recom. 28: Assegure que cada seleção em uma seleção heterogênea seja diferente de todas as demais.

Recom. 29: Assegure que todos os casos de seleção tenham sido considerados.

Todas as seleções múltiplas, heterogêneas ou não, devem ter um caso *default*. Este captura todas as condições não identificadas explicitamente. Na maioria das vezes, as condições *default* corresponderão a erros de execução.

O formato completo de condições de seleção heterogêneas gera repetições de avaliação. Em geral os compiladores otimizadores eliminam todas, ou quase todas estas redundâncias. No entanto, quando forem utilizadas funções, estas serão ativadas repetidas vezes, tantas quantas forem as expressões avaliadas em que as funções forem repetidas. Deve-se, então, fatorar estas funções, atribuindo o resultado de sua avaliação a um valor temporário.

Um função provoca um *efeito colateral*, caso ela altere valores de parâmetros, valores de variáveis globais, conteúdos de estruturas de dados ou de arquivos, ou estados de funcionamento do equipamento. Funções deste tipo normalmente produzem resultados diferentes a cada vez que forem ativadas. Por esta razão elas jamais deverão ser utilizadas em expressões condicionais.

3.6 Regras para escopo de controle e escopo de efeito

Recom. 30: Assegure que o escopo de efeito de uma expressão condicional seja igual ao seu escopo de controle.

Esta recomendação é particularmente importante quando as expressões de controle forem avaliadas em funções ou métodos diferentes.

Bibliografia

[Gane79] Gane, C.; Sarson, T.; *Structured Systems Analysis: Tools and Techniques*; Prentice Hall; Englewood Cliffs; 1979

[LG86] Liskov, B.; Guttag, J.; *Abstraction and Specification in Program Development*; McGraw Hill; 1986

Esta é uma boa referência para a teoria utilizada ao demonstrar a corretude de programas. Ela é particularmente interessante por tratar detalhadamente de tipos abstratos de dados, a base da programação orientada a o bjetos.