Programação Modular

Introdução:
Vantagens de programação modular

- Vencer barreiras de complexidade do trabalho
- Facilita o trabalho em grupo, após a divisão de tarefas no grupo.
- Re-uso
- Facilita a criação de um acervo (Diminui a quantidade de novos programas implementados)
- Desenvolvimento incremental
- Aprimoramento individual
- Facilita do administrador de baselines

Princípios de modularidade:

1) Módulo
    - Definição física: Unidade de compilação independente
    - Definição logica: Trata de um único conceito

2) Abstração de sistema
    - Abstrair e o processo de considerar apenas o que é necessário em uma situação e descartar com segurança o que não é necessário.
    - Níveis de abstração: Sistema > Programa > Módulos > funções > Bloco de código > linhas de código

      Obs: Conceito artefato: é um item com identidade própria criado dentro de um processo de desenvolvimento que pode ser versionado.

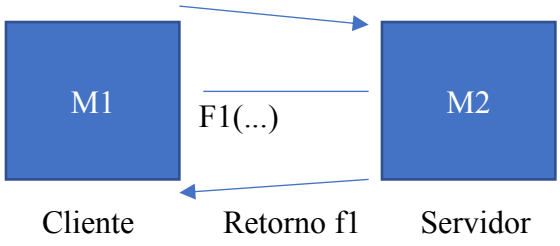
    - Construto (build): Algo para apresentar, artefato.

3) Interface

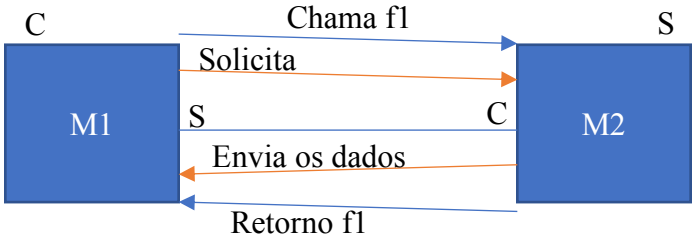
   Mecanismo de troca de {dados, estados, eventos} entre elementos de um mesmo nível de abstração.

   a) Exemplo de Interface:
      - Arquivo (entre sistemas)
      - Funções de acesso (entre módulos)
      - Passagem de parâmetros
      - Variáveis globais (entre blocos)
   b) Relacionamento cliente-servidor:

Request



Caso Especial: Callback

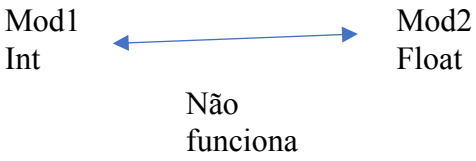


c) Interface fornecida por terceiros

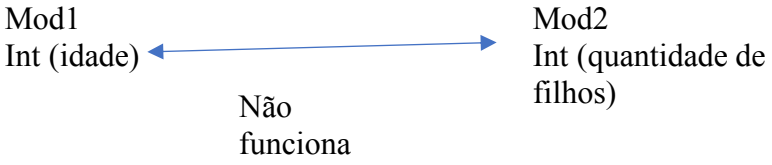
TpAluno.c←TpAluno.h→ TpAluno1.c

d) Interface em detalhe

- Sintaxe: Regra



- Semântica: Significado

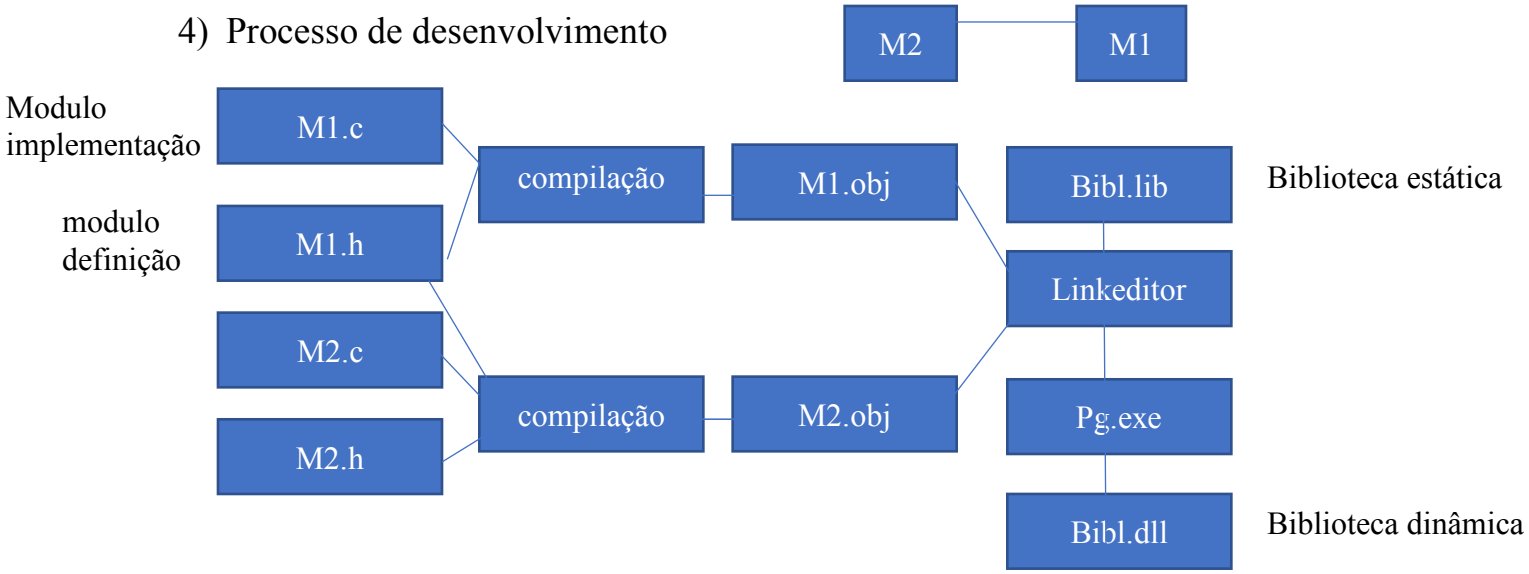


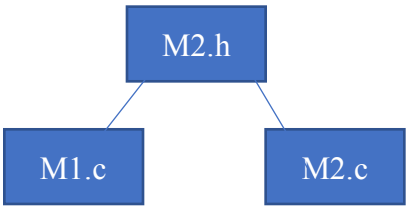
e) Análise de Interface

- Tp Dados Aluno * ObterDadosAluno(int mat) → Protótipo ou assinatura de função de acesso
- Interface esperada pelo cliente, ponteiro para dados válidos do aluno correto ou null.

- Interface esperada pelo servidor, inteiro válido representando a matricula de um aluno.
- Interface esperada por ambos
  TpDados Aluno

4) Processo de desenvolvimento





5) Bibliotecas estáticas e dinâmicas

- Estática
  Vantagens:
  - Lib acoplada em tempo de linkediçao a aplicação executável

  Desvantagens:
  - Existe uma copia dessa biblioteca para cada executável na memoria que a utiliza
- Dinâmica
  Desvantagens:
  - A dll precisa estar na maquina para a aplicação funcionar

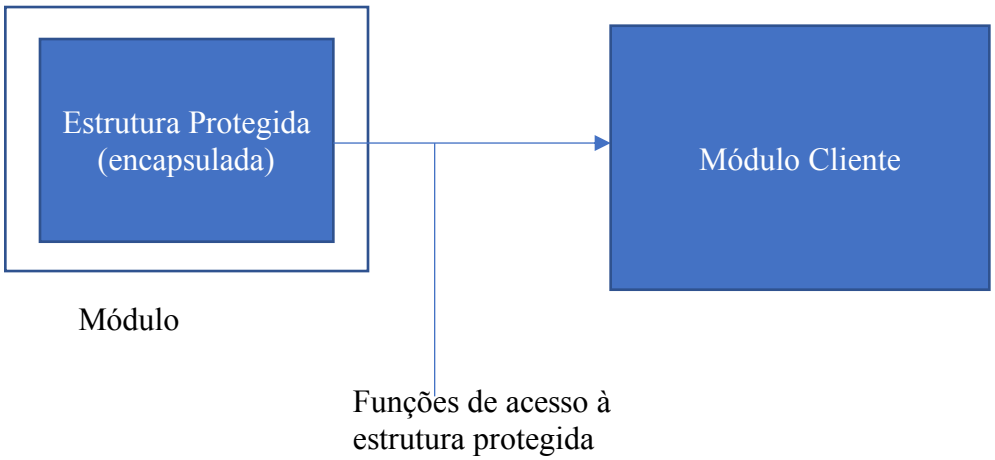
  Vantagens:

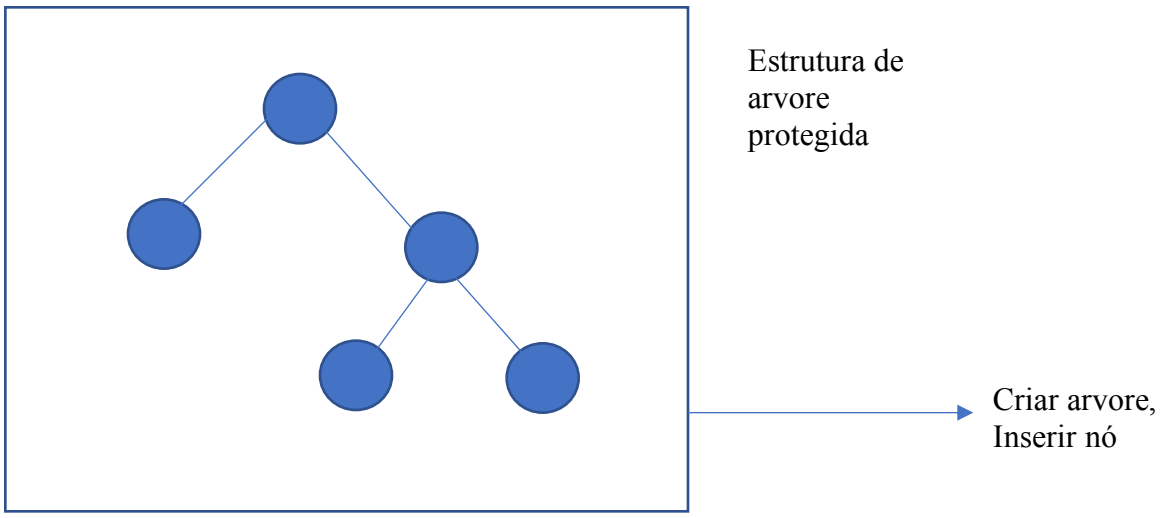
- Só tem uma instancia na memoria: só é carregada uma instancia na biblioteca dinâmica na memoria mesmo que varias aplicações à acessem.

6) Módulo de definição(.h)
   - Interface do modulo
   - Contem os protótipos das funções de acesso interfaces fornecidas por terceiros (ex: tpDadosAluno de item 3e)
   - Documentação voltada para o programador do modulo-cliente.

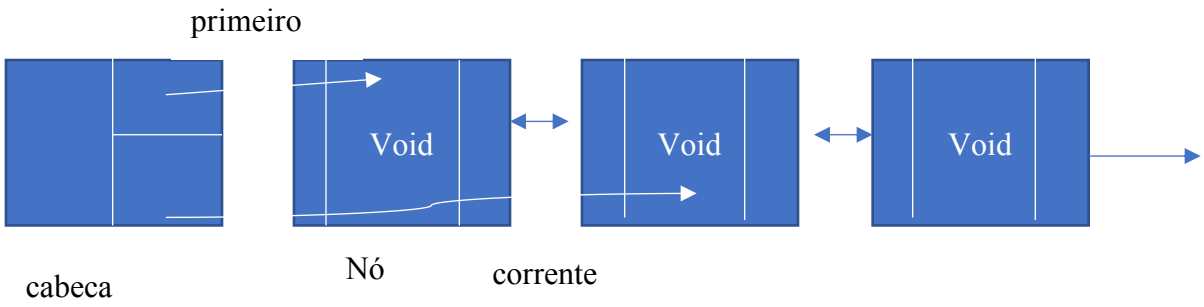
7) Modulo de implementação (.c)
   - Código das funções de acesso
   - Código e protótipos das funções internas
   - Variáveis internas ao modulo
   - Documentação voltada para o programador do modulo servidor

8) Tipo Abstrato de Dados





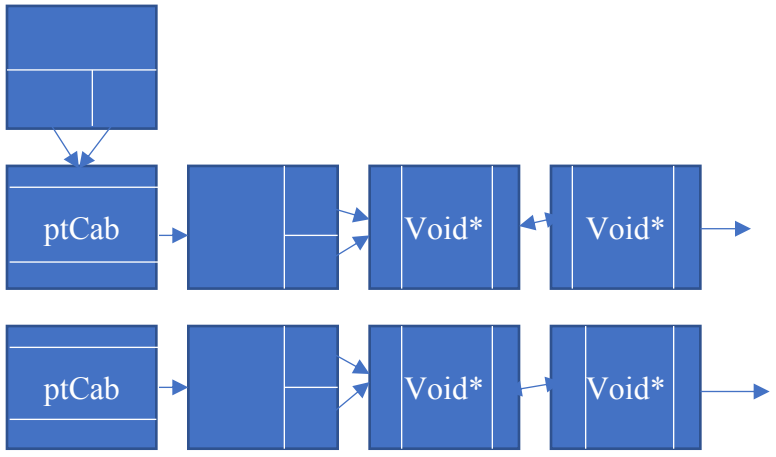
- TAD, é uma estrutura encapsulada em um modulo que somente é conhecido pelos módulos cliente através das funções de acesso disponibilizadas na interface.



No TAD, são usadas funções como:
CriarLista(ptCab *plista);
InserirNo(ptCab plista, void *item);

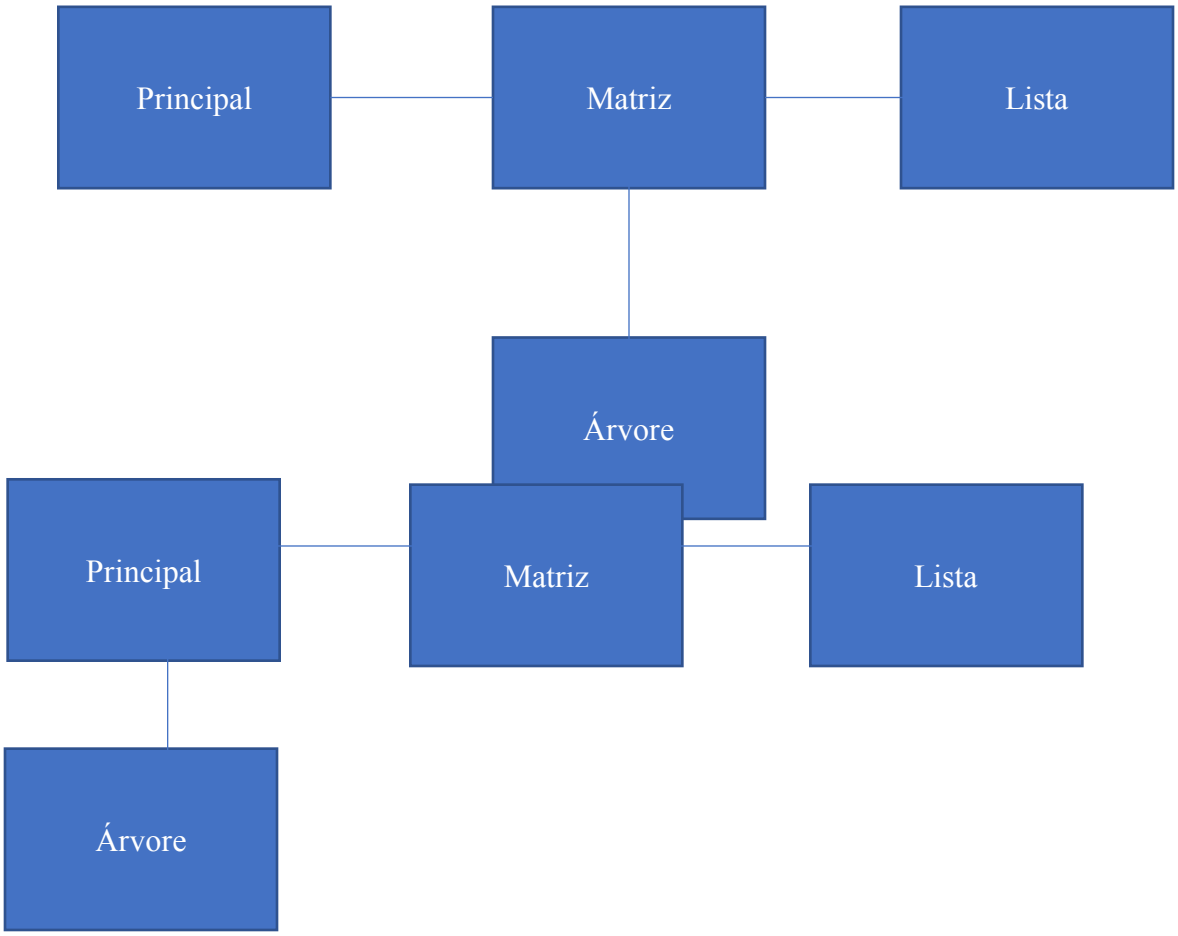
- Na primeira função e usado passagem de parâmetros por referencia, aonde o endereço de uma variável no modulo cliente terá seu valor alterado no modulo cliente.
- Na segunda função a passagem de parâmetros por valor, é acessado o valor da variável para uso interno, sem alterar seu valor.

Para montar uma matriz 2x2 por meio de TAD lista:
Crialista(p1);
Crialista(p2);
inserirNo(p2,NULL);
inserirNo(p2,NULL);
inserirNo(p1,p2);
criaLista(p3);
inserirNo(p3,NULL);
inserirNo(p3,NULL);
inserirNo(p1,p3);



- Ao reciclar o código da lista na implementação do modulo matriz, o modulo principal não precisa fazer chamadas de funções de acesso do modulo lista, essas chamadas são feitas no modulo matriz.

  Na estrutura abaixo o modulo matriz necessita de lista em sua estrutura e também pode armazenar árvores em seus elementos. Porém o módulo principal não pode criar árvores fora da matriz.



Neste modelo é possível criar arvores e matrizes separadamente, e também matrizes de arvores. O modulo principal pode fazer isso.

9) Encapsulamento
   Propriedade relacionada com a a proteção dos elementos que compõem um modulo.

Objetivo:

- Facilitar a manutenção
- Impedir a utilização indevida da estrutura de dados

Outros tipos de encapsulamento:

- Documentação interna – modulo impli.c
- Documentação externa – modulo def.c
- Documentação de uso – manual do usuário

De código:

- Blocos de código visíveis apenas
- Dentro do modulo
- Dentro de outro bloco de código (ex: conjunto de comandos dentro de um for)
- Código de uma função

De variáveis:

- Private(encapsulada no objeto), public, global, global static (modulo), protected (estrutura heranças), static (classe), local(bloco de código), etc

10) Acoplamento

- Propriedade relacionada com a interface entre os módulos
- Conector → item de interface

Ex: função de acesso

- Variável global

Critérios de qualidade:

- Quantidade de conectores

Necessidade X Suficiente
(tudo é útil?) (falta algo?)

- Tamanho do conector (ex: quantidade de parâmetros de uma função)

Complexidade do conector:

- Explicação em documentação
- Utiliza mnemônicos → nomes compatíveis

11) Coesão

Propriedade relacionada com o grau de interligação dos elementos que compõem um modulo:

Níveis de coesão:

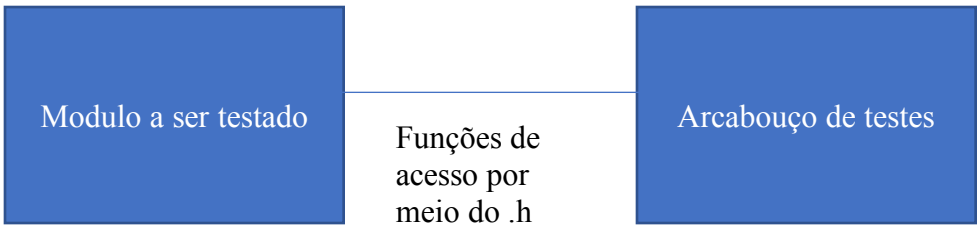
- Incidental – pior coesão
- Logica – elementos logicamente relacionados
- Temporal – itens que funcionam em um mesmo período de tempo
- Funcional – mesma função/funcionalidade

- Abstração de dados
  - Uma união de conceitos (ex: TAD)

Teste automatizado

1) Objetivo
   Testar de forma automática um conjunto de casos de teste na forma de um script e gerando um log de saída com a analise entre o resultado esperado e o obtido.
   Obs: A partir do primeiro retorno esperado diferente do obtido no log de saída, todos os resultados de execução de casos de teste não são confiáveis.

2) Framework de teste



- Genérica: lê o script e gera o log

- Específica: Traduz o script para as funções de acesso do modulo
- Parte genérica do arcabouço esta no Arcaboucoteste.lib, e a especifica esta no testarv.c. O conector entre os dois é a função efetuarComando, chamada na parte genérica. Aonde a entrada é o comando e a saída o retorno da função testada.

3) Script de teste
   - //-> Comentário
   - == Caso de teste -> testa determinada situação
   - =comando de teste -> associada a uma função de acesso
     Obs: teste completo -> casos de teste para todas as condições de retorno de cada função de acesso do modulo. (exceto falta de memoria)

4) Log de saída
   - ==caso1 – esperado coincide com obtido
   - ==caso2 – erros não esperados 1>> função esperava 0 e retorno 1
   - Erros esperados: 1>> 2>>
     Obs: Com a função recuperar é possível zerar o contador de erros, essa função server como um debug.

5) Parte especifica
   - A parte especifica que necessita ser implementada para que o framework(arcabouço) possa acoplar na aplicação chama-se hotspot.
     Ex: testearv.c

Processo de desenvolvimento em engenharia de software

Demanda (cliente) -> analista de negócios (contrato) ->Líder de projeto

- Projeto {tamanho (Ponto de função) ,esforço,recursos,prazo}
- Estimativa (pode ser feita por ponto de função, aonde é cobrado por cada item desenvolvido)
- Planejamento

- Acompanhamento

A. Requisitos:
   - Elicitaçao (coletar as informações do cliente)
   - Documentação
   - Verificação
   - Validação
B. Analise e projeto
   - Projeto lógico: modelagem de dados em UML
   - Projeto físico: tabelas do banco de dados
C. Implementação
   - Programas
   - Teste unitário (caso esteja tudo ok, passe para próxima fase)
D. Testes
   - Teste integrado (Build)
E. Homologação (beta)
   - Sugestões
   - Erros
F. Implantação
   - Gerencia de configurações (Baseline)
   - Qualidade de software (mesura etapas)

Especificações de Requisitos

1) Definição de requisito

- O que tem que ser feito (obs: **não é como** deve ser feito)

2) Escopo de Requisitos

- Requisitos mais genéricos (amplos)
- Requisitos mais específicos

3) Fases de Especificação

- Elicitação: Captar informações do cliente para realizar a documentação do sistema a ser desenvolvido.

Técnicas de elicitação:

- Entrevista
-Brainstorm (ideias)

-Questionário

Documentação

- requisitos descritos em itens diretos

-uso da língua natural cuidado com ambiguidade

-dividir requisitos em seus diversos tipos

(obs: tipos de requisitos)

requisitos funcionais: oque deve ser feito em relação a informatização das regras de negocio

requisitos não funcionais: propriedades que a aplicação deve ter e que não estão diretamente relacionadas com as regras de negocio.

Ex:- segurança

Ex: login e senha

-tempo de processamento

Ex: As consultas não podem demorar mais do que 5s.

-disponibilidade

Ex:24/7.

Requisitos inversos

- O que não é para fazer

Verificação

- A equipe técnica verifica se o que esta descrito na documentação é viável de ser desenvolvido.

Validação

-Cliente valida a documentação

4) Exemplos de Requisitos
   a) Bem formulados
      a. A tela de resposta da consulta de aluna apresenta nome e matricula.
      b. Todas as consultas devem retornar respostas no máximo em 2s

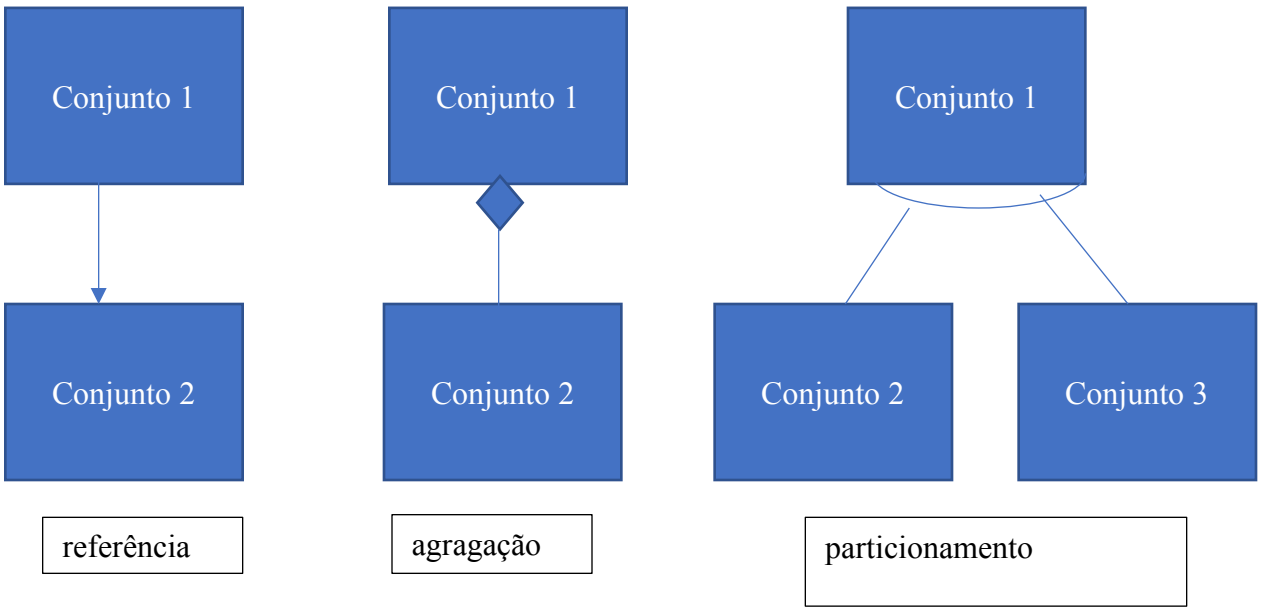
   b) Mal formulado
      a. O sistema é de fácil utilização
      b. A consulta deverá retornar uma resposta em um tempo reduzido.

c. A tela mostra seus dados mais importantes

Modelagem de Dados

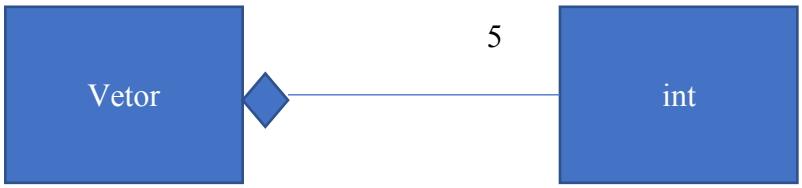
1 modelo -> n exemplos

Notação

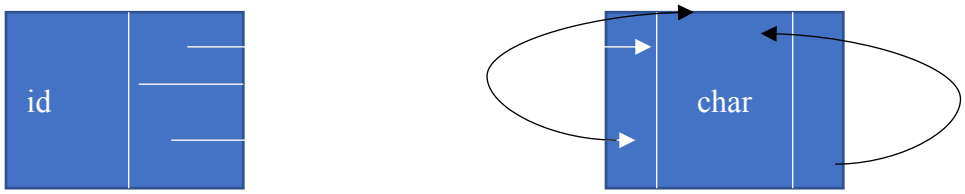


Exemplos:

a) Vetor de 5 posições que armazenam inteiros.



b) Árvore Binária com cabeça que armazena caracteres.



c) Lista duplamente encadeada com cabeça que armazena caracteres. (mesmo desenho do modelo acima)
Desempate das duas estruturas: assertivas estruturais – são regras usadas para o desempate de 2 modelos iguais. Essas regras complementam o modelo, definindo características que o desenho não consegue representar.

Lista:

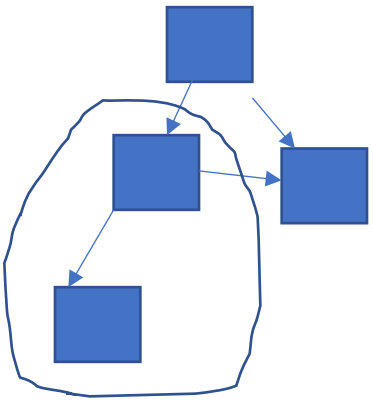
- Se pCorr->pAnt != nulo então pCorr->pAnt->pProx == pCorr

Se pCor->pProx != nulo então pCorr->pProx->pAnt == pCorr

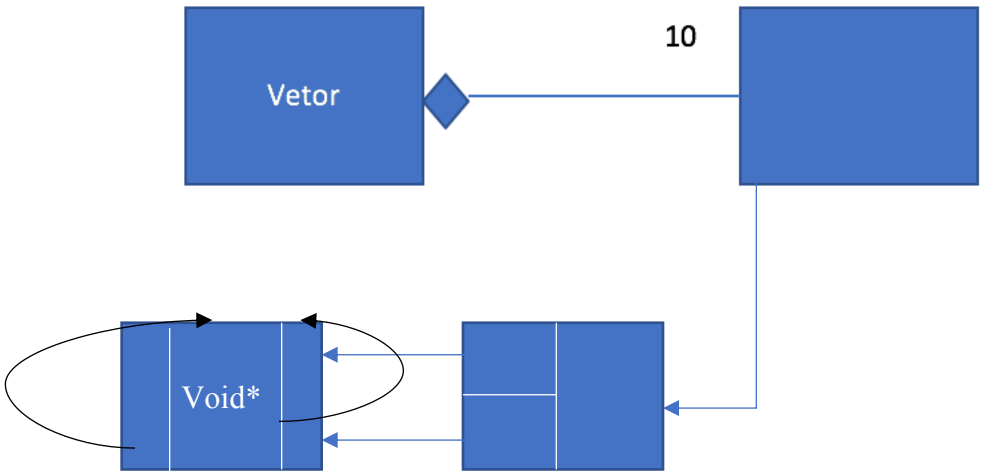
Árvore:

-um ponteiro de uma subarvore a esquerda nunca referencia um no de uma subarvore a direita

-pAnt e pProx de um no nunca aponta para o pai.



d) Vetor de listas duplamente encadeadas com cabeça e genérica.



e) Matriz tridimensional generica construida com listas duplamente encadeadas com cabeca.

Assertiva estrutural

-A matriz comporta apenas 3 dimensoes sendo o no da lista mais interna preenchido com um void*

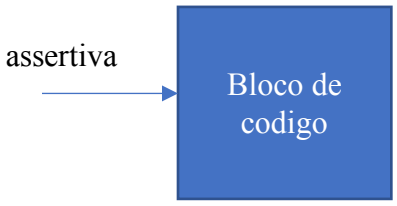
f) Grafo criado com listas

Assertivas

1) Definição

-qualidade por construção qualidade aplicada a cada etapa do desenvolvimento de um aplicação.

Assertivas são regras consideradas válidas em determinado ponto do código.



2) Onde aplicar assertivas

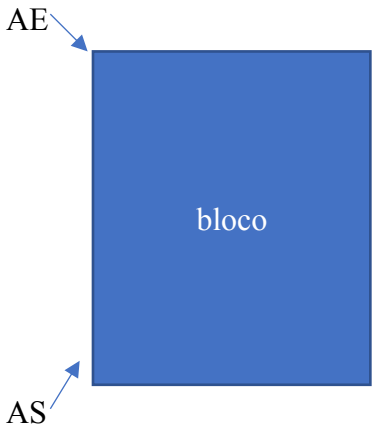
-Argumentacao de corretude

- instrumentação (quando voce implementa as assertivas de forma que próprio código saiba que esta correta)

- trechos complexos em que é grande o risco de erros.

3) Assertivas de entrada e saída

-obs: a assertiva de tratar de regras envolvendo dados e acoes tomadas.



4) Exemplos

-Excluir nó corrente intermediário de uma lista duplamente encadeada.

AE: - Ponteiro corrente referencia o no a ser excluído e este é intermediário.

- A lista existe.

AS:- nó referenciado foi excluído.

- corrente aponta para o anterior.

Exercicios

5) Por que a utilização de assertivas constribui para a qualidade por construção?
6) Apresente um requisito inverso mal-formulado.
7) O que é brainstorm e em qual etapa esta técnica é utilizada?

8) Escolha uma função do seu trabalho e inclua assertivas de entrada e saída e ao menos duas assertivas intermediárias.
9) Elabore o modelo, exemplo e asserivas estruturais de uma matiz estática de arvores n-árias (não construídas com arvores binarias)
10) Apresente 2 requisitos funcionais e não funcionais de seu trabalho.

IMPLEMENTAÇÃO DA PROGRAMAÇÃO MODULAR

Espaço de dados:

- São áreas de armazenamento
- Alocadas em um meio
- Possuem um tamanho (ocupam espaço)
- Possuem um ou mais nomes de referência

V[i] – i-ésimo elemento do vetor
ptAux* - espaço de dados referenciado/apontado por ptAux.
(*ObterElemento(int id)).id/ObterElemento(int id)->id é o subcampo de dados retornado pela função.

Tipos de dados:

- Determinam a organização, codificação, tamanho em bytes e valor permitidos.

Tipo de tipo de dados:

- Tipos computacionais: int,char,char*...
- Tipos básicos: enum, typedef,union
- Tipos abstratos de dados (TAD)

Tipos básicos:

- **Typedef:** Nomeia um tipo de dados
- **Enum:** enumera nomes idênticos associados a um inteiro
- **Struct:** Dados conglomerados
- **Union:** Como se fosse um typecast,porém interpreta o campo da estrutura com tipos diferentes.

Declaração e definição de elementos:

- **Definir:** aloca espaço e associa a um nome
- **Declarar:** associa o espaço a um tipo de dados

Obs: Tipos computacionais, são simultaneamente declaração e definição daquela estrutura de dados.

Implementação em C/C++:

- Declarações e definições de nomes globais exportados pelo módulo servidor

Ex:

Int a;
Int F (int b);

- Declarações externas contidas no modulo cliente e que somente declaram o nome sem ser associado a um espaço de dados:

  Ex:
  Extern int a;
  Extern int F (int b);

- declarações e definições de nomes globais encapsulados no módulo:
  static int A;
  static F (int B);

Este encapsulamento protege as estruturas de dados de um módulo dos módulos clientes.

Resolução de nomes externos: Um nome externo declarado em um modulo precisa estar declarado e definido em outro modulo.

- Ajusta endereços para os espaços de dados

  Pré-processamento: Os comandos iniciados com # são interpretados pelo pré-processador, e gerados códigos fontes. Ex:

#define nome valor (atribuir o valor ao nome)

#include <arquivo>/"arquivo"(inclui todo o texto do arquivo)

#undef nome (A partir dessa linha, não haverá mais a substituição da string nome)

#ifdef nome

código

#endif (definição condicional de um string)

#if !defined (nome) ou #ifndef nome (negação da condição de uma string)

Exemplo 1:

#if !defined (EXEMP_MOD)

#define EXEMP_MOD

Código

#endif

Evita que um módulo seja incluído duas vezes,caso por algum erro seja definido mais de uma vez pelo cliente.

Exemplo 2:

M1.H

```
#if defined (EXEMP_OWN)
#define EXEMP_EXT
#else
#define EXEMP_EXT extern
#endif
EXEMP_EXT int vetor[7]
#if defined EXEMP_OWN
      ={1,2,3,4,5,6,7}
#else
;
#endif
```

M1.C

```
#define EXEMP_OWN
#include “M1.h”
#undef EXEMP_OWN
```

Resulta em:

Int vetor[7] = {1,2,3,4,5,6,7};

M2.C

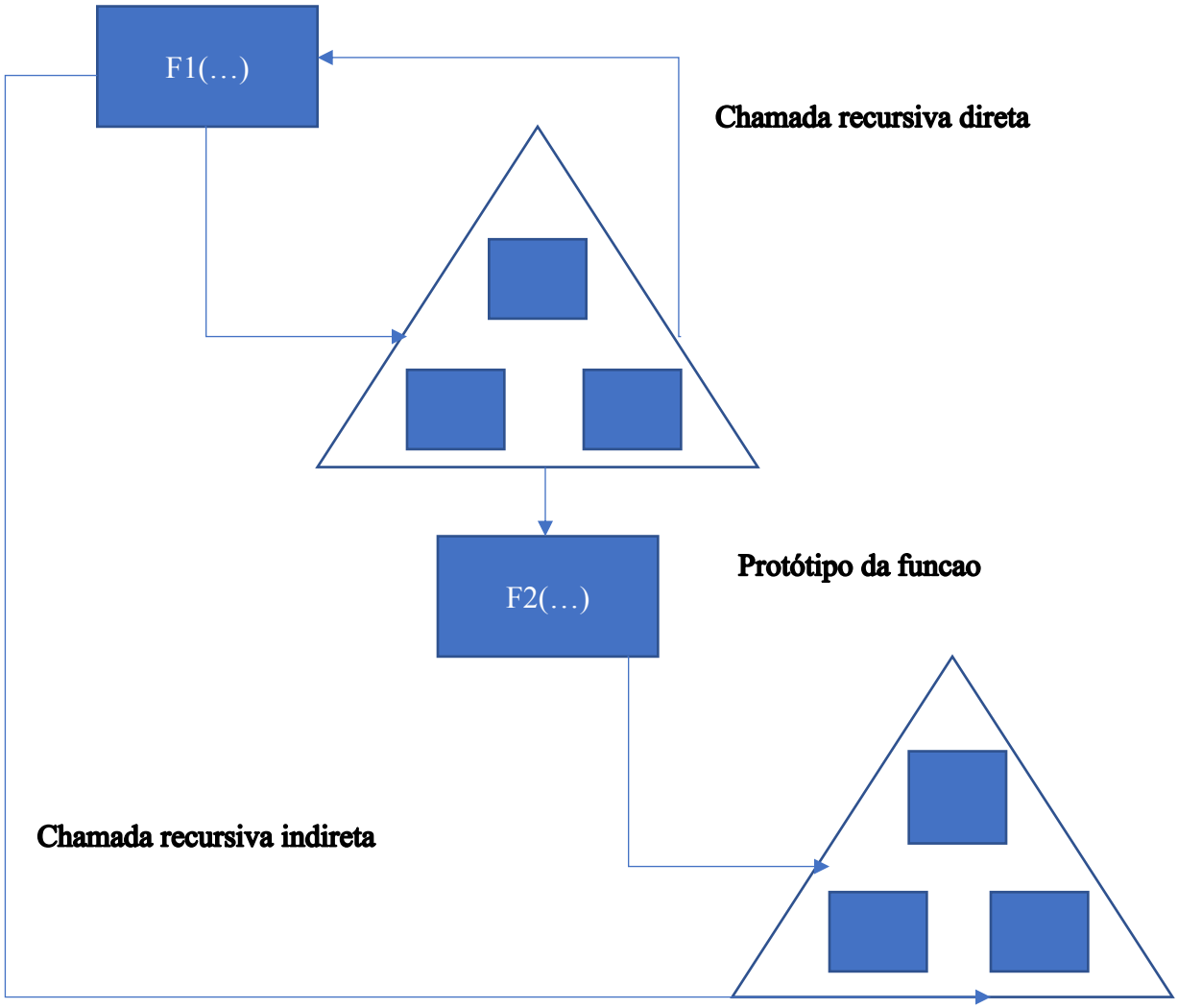
#include “M1.h”

Resulta em:

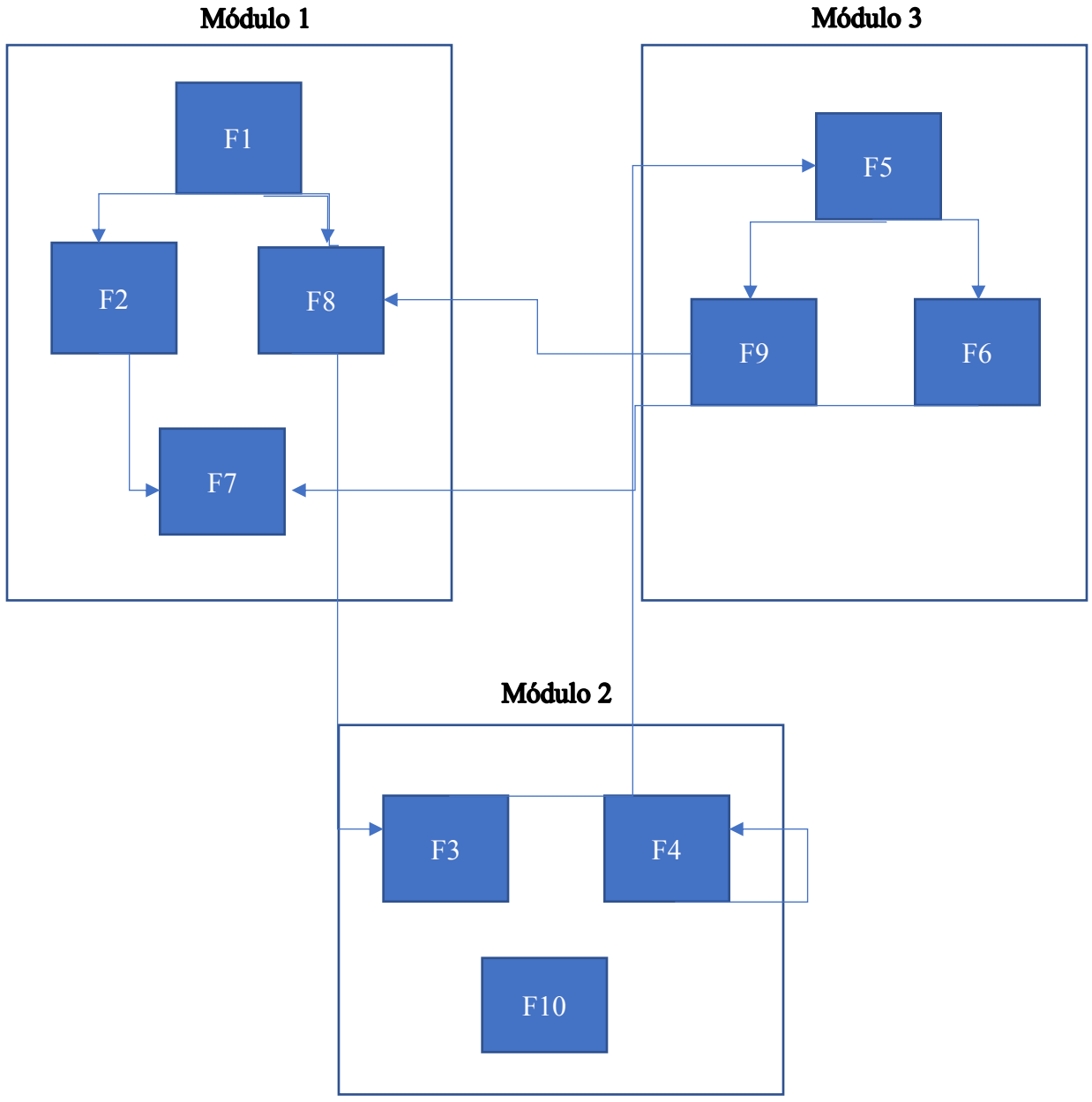
Extern int vetor[7];

EXTRUTURA DE FUNÇÕES

Paradigma: É a forma de programar. Um paradigma pode ser orientado a objetos (programação modular) ou procedural.


F1(…)
Chamada recursiva direta
F2(…)
Protótipo da funcao
Chamada recursiva indireta

Estrutura de chamadas:



F10

F4 -> F4

F9 -> F8 -> F3 -> F5 -> F9
F8 -> F3 -> F5 -> F6 -> F7

F10 é uma função morta, pois não chama nem é chamada por ninguém.
F4 faz uma chamada recursiva direta
F9 faz uma chamada recursiva indireta

Função: É uma porção autocontida de código. Possui nome, assinatura composta de retorno, nome, parâmetro e corpo da função (um ou mais).

Especificação de função: Possui um objetivo (pode ser igual ao nome), acoplamento (itens da interface), condições de acoplamento (assertivas de entrada e saída), interface com o usuário (em um input pedindo ao usuário), requisitos (todos os requisitos que são elicitados têm que ser armazenados em alguma documentação ou parte do código), hipóteses (consideradas válidas antes do desenvolvimento da função) e restrições (limitações do desenvolvimento da função).

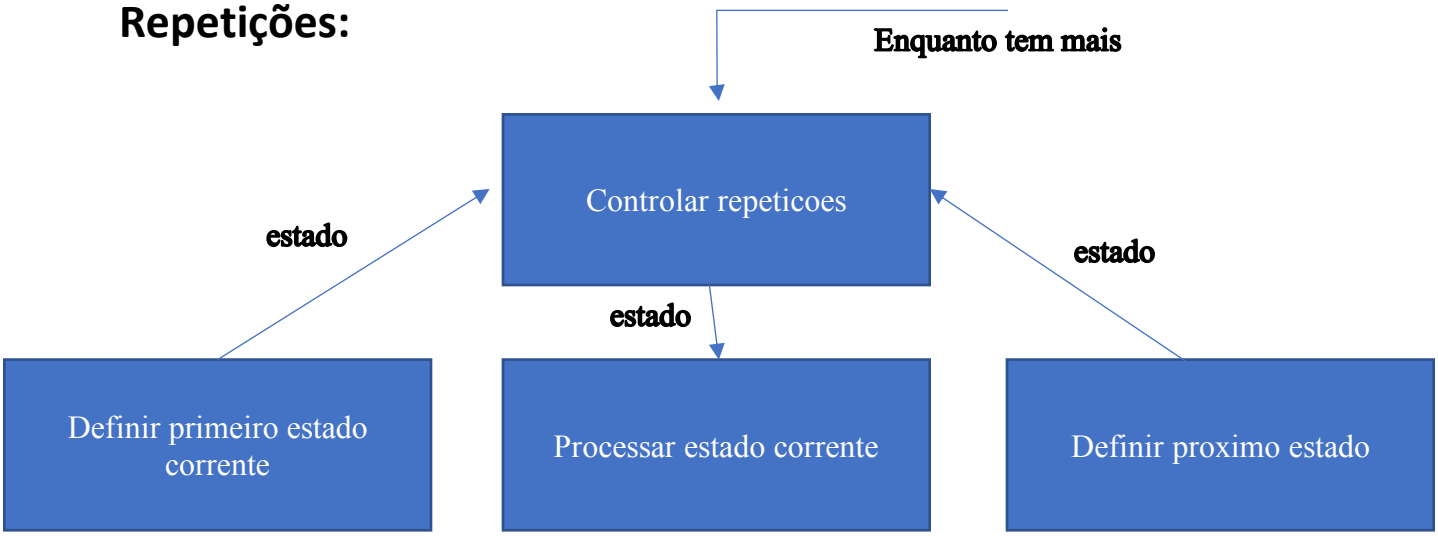
Interfaces:

- **Conceitual:** definição da interface da função sem preocupação com a implementação.
- **Física:** implementação da interface conceitual, com a sintaxe apropriada de uma linguagem de programação.

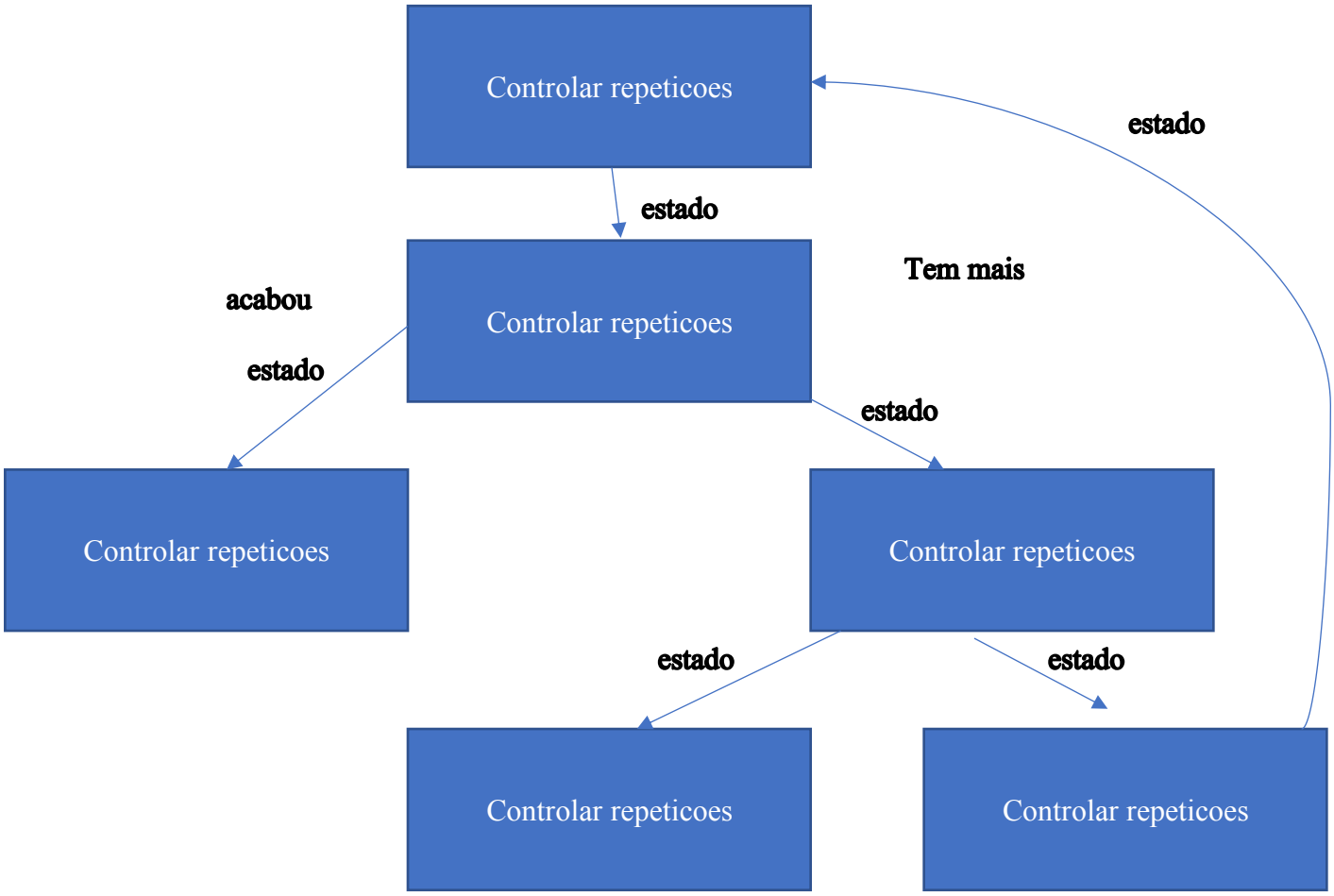
- **Implícita:** dados da interface diferentes de parâmetros e valores de retorno, como uma variável global que e padronizada, e é usada na função. Ex: o ponteiro global para arvore no modulo arvore.

Housekeeping: É o módulo responsável por liberar recursos alocados a programas,
componentes ou funções ao terminar a execução.

Repetições:



Recursão:



Estado:

- **Descritor de estado:** conjunto de dados que definem um estado. Exemplo: índice de um vetor.
- **Estado:** valoração do descritor
- **Obs:** não é necessariamente observável. Ex: cursor de posicionamento de arquivo.

Esquema de algoritmo:

```
inf = ObterLimInf();
 sup = ObterLimSup();
 while ( inf <= sup )  {
meio = (inf+sup)/ 2 ;
  comp = comparar(valorProc,obterValor(meio));
 if (comp == IGUAL) {
break ;
}
if (comp==MENOR){
sup=meio- 1 ;
}
else {
inf = meio + 1 ;
}
}
```

Permite encapsular a estrutura de dados utilizada. É correto, independente da estrutura, e incompleto, logo precisa ser instanciado.

Obs:

- Hotspot são funções de interface (parte especifica) ex:ObterValor
- Se o esquema estiver correto e o hotspot possuir assertivas válidas, o programa está correto.

Parâmetros do tipo ponteiro para funcao:

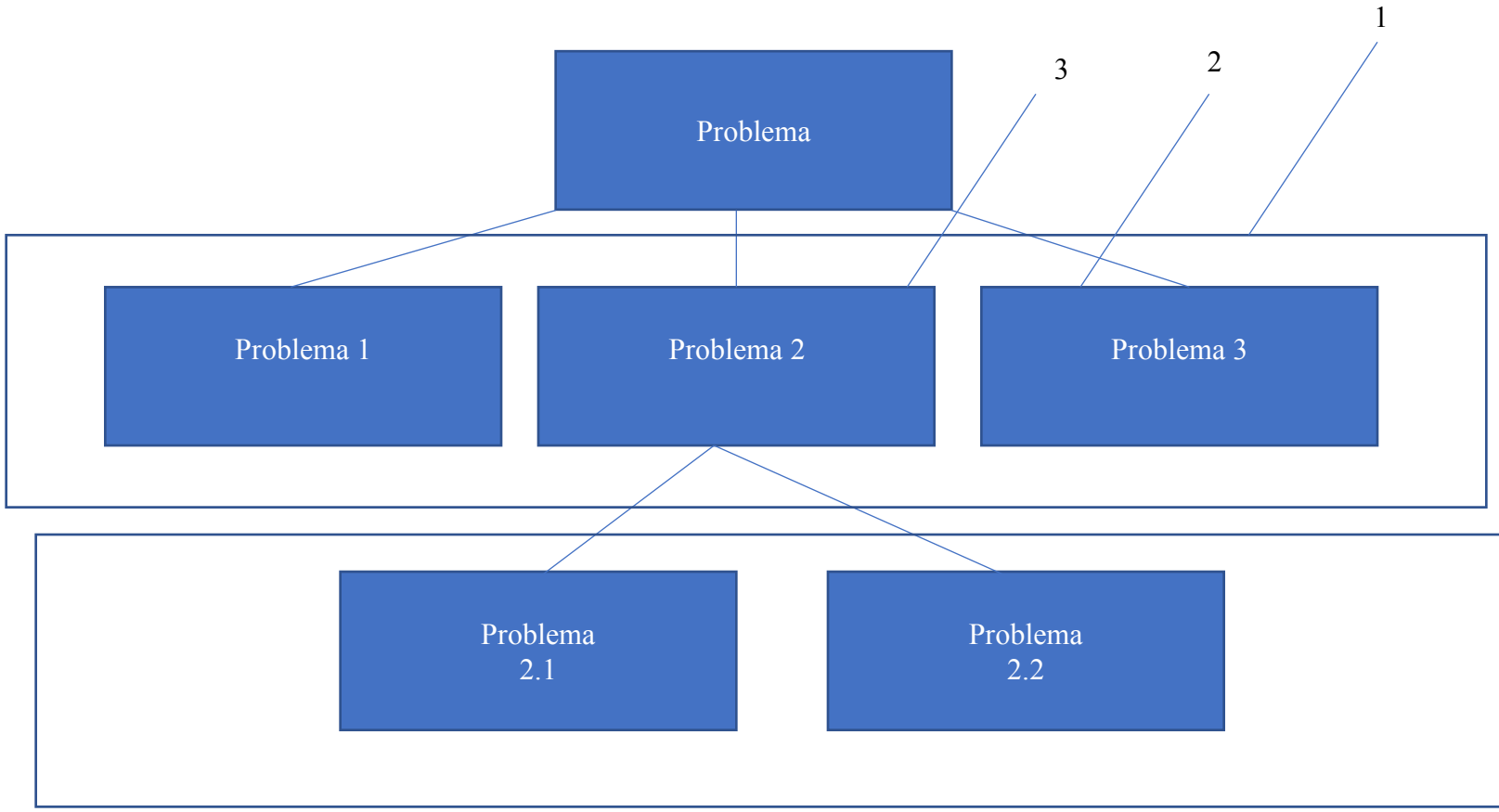
```
float areaQuad( float base, float altura ) {

return base*altura;

 }

float areaTri( float base, float altura ) {

return (base*altura)/ 2 ;

}

int processaArea( float valor1, float valor2, float (*Func)( float , float ) )  { printf ( "%f" , Func(valor1,valor2));

}

CondRet = processaArea( 5 , 2 ,areaQuad);

 CondRet = processaArea( 3 , 3 areaTrig);
```

Decomposição sucessiva:

Conceito:

- **Divisão e conquista:** dividir o problema em subproblemas menores, para resolve-los.

Estrutura de decomposição:



1)Conjunto solução: todos os componentes englobam o mesmo problema.
2) Componente abstrato: aquele que se divide em outros componentes.

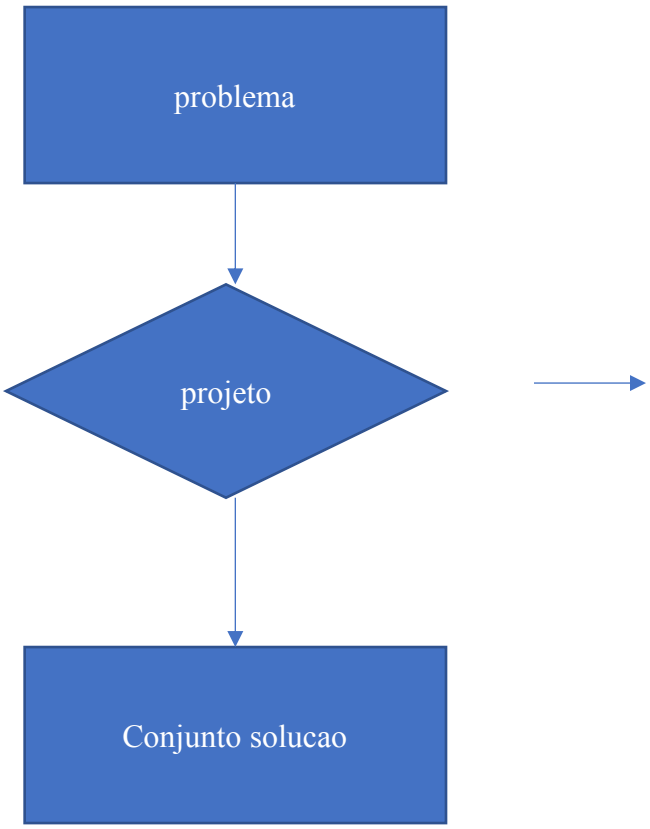
3) Componente concreto: aquele problema/componente que não precisa ser subdividido em problemas menores.

- No diagrama anterior, o problema é o componente raíz.
- Uma estrutura de decomposição gera uma solução apenas.

Critérios de qualidade:

- **Complexidade:** quantidade exagerada de etapas/ sub-componentes.
- **Necessidade:** sub-componente fora do escopo, que não altera a solução do problema. (Acontece geralmente no controle de versão, quando há uma mudança na estrutura, mas as funções de acesso não são atualizadas.)
- **Suficiência:** Saber se os componentes que fazem parte do conjunto solução são suficientes para resolver o problema.
- **Ortogonalidade:** dois componentes não realizam a mesma funcionalidade, disjuntos, dentro do mesmo conjuntos solução.
- **Obs:** se uma mesma funcionalidade repetir em uma solução, provavelmente trata-se de uma função, que é chamada em mais de um componente.

Passo de projeto:



Etapa do desenvolvimento que procura projetar o conjuntos solucao necessario para resolver o problema.

Direção de projeto:

- **Bottom-up:** primeiro são implementados os sub-componentes mais internos, para então implementar os mais acima da estrutura.
- **Top-down:** Primeiro são implementados os componentes principais para então desenvolver os componentes mais específicos.
- **Obs:** É possível conciliar as duas direções de projeto.

ARGUMENTACAO DE CORRETUDE

```
INICIO

  IND ← 1

  ENQUANTO IND <= LL FAÇA

     SE ELE[IND] = PESQUISADO
        BREAK
FIM-SE

IND←IND+ 1

FIM-ENQUANTO

  SE IND <= LL

MSG "ACHOU"

 SENÃO

MSG "NÃO ACHOU"

 FIM-SE

FIM
```

Definição: é um método utilizado para argumentar que um bloco de código está correto (mais para fins acadêmicos, mas também empregado no mercado para trechos de códigos não-triviais).

Tipos de argumentação:

- **Sequência** : blocos um após o outro
- **Seleção** : if, switch, else
- **Repetição** : for, while

Argumentação de sequência :

Assertivas intermediárias acontecem quando a assertiva de saída de um bloco coincide com a assertiva de entrada do bloco seguinte. No código-exemplo, as assertivas de entrada e saída e intermediárias do código seriam as seguintes:

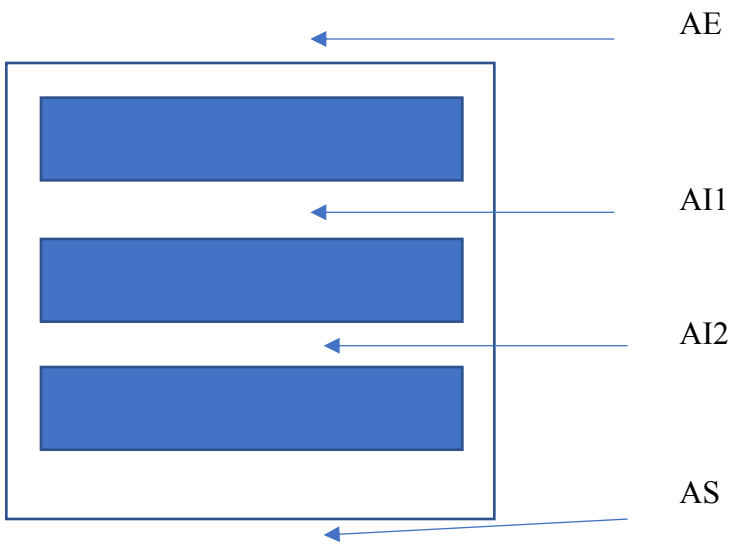
AE: existe um vetor válido e um elemento a ser pesquisado.

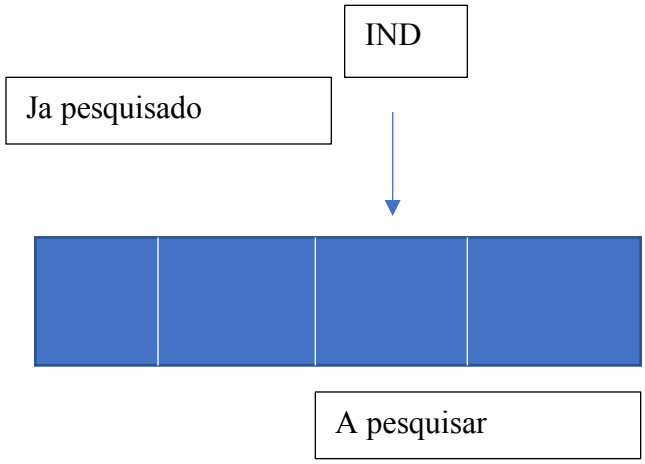
AS: imprima “ACHOU” se o elemento foi encontrado ou “NÃO ACHOU” se end > LL^2.

AI : IND aponta para primeira posição do vetor **1**

AI : Se o elemento foi encontrado, IND aponta para o **2**

mesmo, senão IND > LL.





Argumentação de seleção :

1) AE&&(C==T)+B→ AS

AE&&(C==F)→ AS

2) AE&&(C==T)+B $_1$ → AS

AE&&(C==F)+B $_2$ → AS

(1) Se as assertivas de entrada estiverem verdadeiras e a condição estiver verdadeira e o bloco **B** foi executado, a assertiva de saída é verdade.

(2) Pela AE, se o elemento for encontrado IND aponta o mesmo (e é menor do que LL) com (C == T), IND <= LL, **B $_1$** apresenta mensagem “ACHOU”, valendo AS. Pela AE, IND > LL se o elemento pesquisado não foi encontrado. Como (C == F), INF > LL, neste caso **B $_2$** é executado apresentando mensagem “NÃO ACHOU”, valendo AS.

Argumentação de repetição :

AE: AI $_1$ AS: AI $_2$

AINV: assertiva invariante, envolve os dados descritores de estado, que deve ser válida a cada ciclo de repetição - isto é, não varia com os ciclos.

AINV : Existem dois conjuntos: já pesquisados e a pesquisar. INF aponta para elemento do a pesquisar.

1. AE → AINV : A assertiva invariante é válida antes da assertiva de entrada. Conjunto já pesquisado é vazio e IND aponta para primeiro elemento. Portanto todos os elementos estão no conjunto a pesquisar.

2. AE && (C == F) → AS : A condição será falsa se não entrar no loop, ou entrar mas não fechar o primeiro ciclo.

2 . 1 . N ã o e n t r a : p e l a A E = A I $_1$, I N D = 1 . C o m o (C = = F) , L L < 1 . O u s e j a , L L = 0 . O u s e j a , o vetor é vazio. Neste caso, é válido a assertiva de saída pois o elemento procurado não foi encontrado.

2.2. Não completa o primeiro ciclo : pela AE, IND aponta para o primeiro elemento do vetor. Se este for igual ao pesquisado, o break é executado e IND aponta para elemento encontrado, valendo AS.

3. AE && (C == T) + B → AINV : Pela assertiva, IND aponta para primeiro elemento do vetor. Como (C == T), este primeiro elemento é diferente do pesquisado. Este então passa do conjunto a pesquisar para os que já foram pesquisados e IND é reposicionado para outro elemento de a pesquisar, valendo AINV.

4. AINV && (C == T) + B → AINV : Para AINV continuar valendo, B deve garantir que um elemento passe de a pesquisar para já pesquisado e IND seja reposicionado.

5. AINV && (C == F) → AS : Ou a condição é falsa ou o ciclo não completa.

5.1. Condição falsa: pela AINV, IND ultrapassou o limite lógico e todos os elementos estão em já pesquisado. Pesquisado não foi encontrado com IND > LL, vale AS.

5.2. Ciclo não completou: pela AINV, IND aponta para elemento de a pesquisar que é igual a pesquisado. Neste caso vale a AS pois ELE[IND] = PESQUISADO.

6. TÉRMINO : como a cada ciclo, B garante que um elemento de a pesquisar passe para já pesquisado e o conjunto a pesquisar possui um número finito de elementos, a repetição termina em um número finito de passos.

Argumentação de sequência (dentro do ENQUANTO) :

AE : AINV, **AS** : AINV

AI : o elemento pesquisado não é igual a ELE[IND] ou o elemento foi encontrado em IND. **3**

★ **Argumentação de seleção (dentro do ENQUANTO)** : **AE** : AINV, **AS** : AI 3

AE && (C == T) + B → AS : Pela AE, IND aponta para elemento do conjunto a pesquisar. Como C == T, o elemento de IND = PESQUISADO e assim o elemento foi encontrado em IND, valendo a AS.

AE && (C == F) → AS : Pela AE, IND aponta para elemento a pesquisar. Como C == F, o elemento apontado por IND não é o pesquisado, valendo a AE.

INSTRUMENTAÇÃO:

- **Problemas ao realizar testes (diagnóstico):**
- Normalmente é grande
- Muito sujeito a erros
- Não mostra com exatidão a causa do problema
- Tempo decorrido entre o instante da falha e o observado
- Ponteiro “louco” (crazy pointers)
- Comportamento inesperado do hardware

O que é instrumentação?

- **Fragmentos inseridos nos módulos**
  - Códigos
  - Dados (variáveis de controle)
- Não contribuem para o objetivo final do programa
- Monitora o serviço enquanto ele é executado
- Consome recursos de execução (logo não pode estar na versão final).

Objetivos:

- **Detectar falhas:** encontrar falhas no programa com rapidez.
- **Impedir** que falhas se propaguem
- Medir propriedades dinâmicas do programa

Conceitos:

- **Programa robusto** intercepta a execução quando observa um problema. Mantém o dado confinado, explanando ou sugerindo onde está o erro.
- **Programa tolerante a falhas** é robusto, e corrige o erro para integridade do programa. Possui mecanismos de recuperação.
- **Deterioração controlada** é a habilidade do programa de continuar operando corretamente mesmo com a parada de funcionalidade. Exemplo: quando o Microsoft Word permite que o usuário salve o arquivo mesmo quando o processo trava.

Instrumentação em C

```
#ifdef _DEBUG
    // código
#endif
```

Esse código ira rodar em tempo de desenvolvimento na versão debug.

Assertivas executáveis

ASSERTIVAS -> CÓDIGO

- **Vantagens:**
- Informam o problema quase imediatamente após ter sido gerado
- Controle de integridade feito pela máquina
- Reduz o risco de falha humana
- Precisam ser: completas (cobrir todas as assertivas) e corretas

Depuradores (Debuggers): ferramenta utilizada para executar um código passo-a-passo, permitindo que se distribuem breakpoints, com o objetivo de confinar os erros a serem pesquisados. Deve ser utilizado como último recurso.

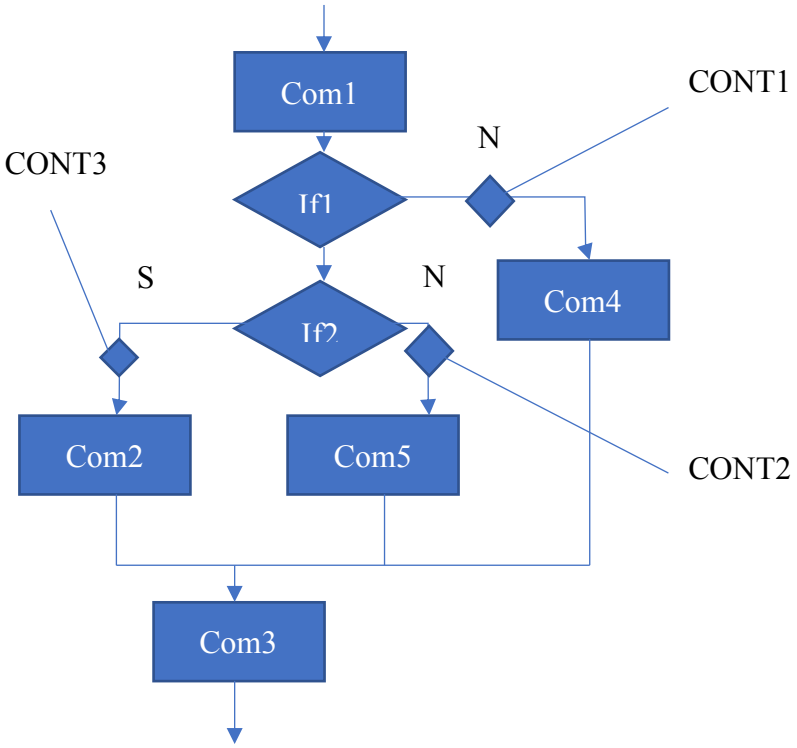
Trace: instrumento utilizado para apresentar uma mensagem no momento em que é executado (printf da linguagem C).

- Trace de **instrução** : printf que apresenta mensagem.

- Trace de **evolução** : apresenta mensagem apenas quando o conteúdo de uma variável for alterado, permitindo a visualização da evolução de um indicador de estado, como um índice de um vetor sendo analisado.

Controlador de Cobertura

- Teste **caixa fechada/preta** : observa-se apenas o que entra na função (parâmetros) e o que sai da função.
- Teste **caixa aberta/branca** : analisa-se todos os caminhos internos da função.



Definição: instrumento composto de um vetor de controladores que tem como objetivo acompanhar os testes caixa aberta de uma aplicação, mostrando todos os caminhos percorridos. Caso um dos caminhos não foi percorrido ,percebe-se que nem todos os caminhos possíveis foram verificados, e a instrumentação não é rigorosa o suficiente.

Vetor de controladores: armazena as labels e contador condizente com a quantidade de vezes que se passou por tal contador. Se pudéssemos modelar o fluxograma como uma árvore, os contadores estariam nos nós-folha, pois pressupõem os “IF’s” passados.

Cont1 = 7

Cont2 = 8

Cont3 = 0

Obs: Na caixa fechada, testa-se as condições de retorno e na caixa aberta, testam-se os caminhos.

Verificador estrutural:

- **Definição:** instrumento responsável por realizar uma verificação completa da estrutura em questão é a implementação de código relacionado com as assertivas estruturais e modelo.

Deturpador estrutural:

- **Definição:** instrumento responsável por inserir erros na estrutura com objetivo de testar o verificador.
- A função deturpa recebe o tipo de deturpação. A deturpação é sempre realizada no nó corrente.

=criaestrutura

=deturpar 1

=verifica

=deturpar 2

=verifica

...
=estatisticacontador

Recuperador estrutural:

- **Definição:** instrumento responsável por recuperar automaticamente uma estrutura a partir de um erro observado em uma aplicação tolerante a falhas.

Estrutura Auto verificável:

- **Definição:** é estrutura que contém todos os dados necessários para que o programa esteja totalmente verificado. Para isso são feitas algumas perguntas à uma lista simplesmente encadeada genérica.

É possível definir todos os tipos de dados apontados pela estrutura?

Não, basta armazenarmos então o tipo apontado por estrutura. Este “tipo” será passado pela função no módulo específico na versão instrumentada (debug) do programa. Vale lembrar que na chamada e definição da função de inserção de nó, deve-se passar este tipo. Além disso, é necessário incluir o campo “tipo” à cabeça da lista, para que nós não tenham o mesmo tipo de cabeça.

É possível acessar qualquer parte da estrutura a partir de qualquer origem?

Não, basta então armazenar um ponteiro para a cabeça da lista e um ponteiro para o nó anterior, tornando a lista duplamente encadeada.

```
Condret inserirno( pt, valor
    #ifdef _DEBUG
      , tipo
      #endif
      );
```