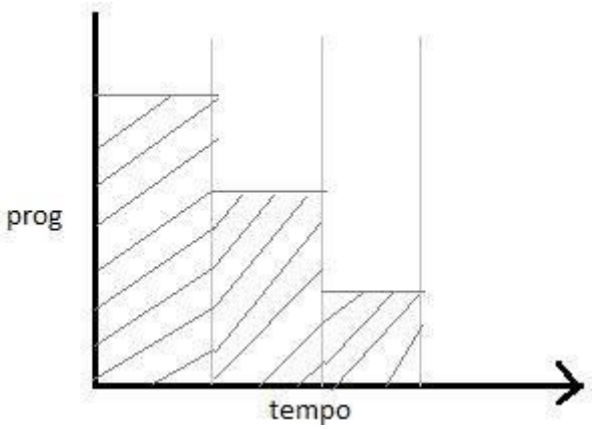
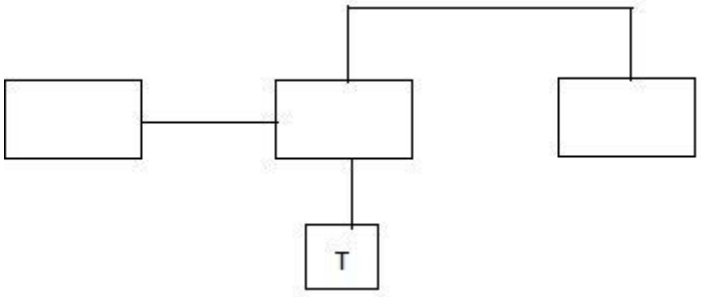
Caderno Gabriel Mascheroni - 1811312

Introdução

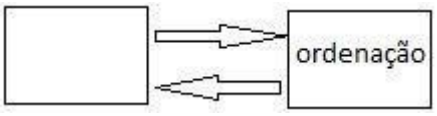
1. Vantagens da Programação Modular:
    a. Vencer barreiras de complexidade
    b. Facilita o trabalho em grupo (paralelismo)
    c. Reuso
    d. Facilita a criação de um acervo (coleção)



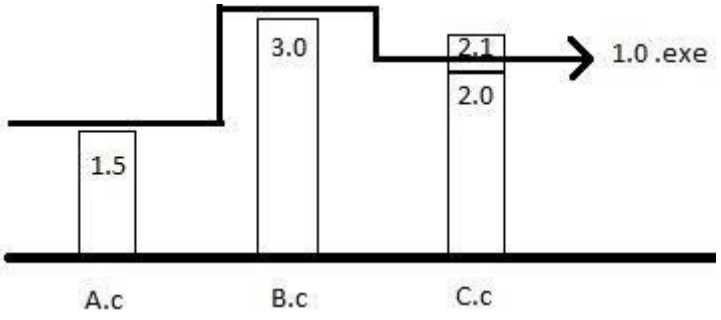
    e. Desenvolvimento incremental (entregas)



    f. Aprimoramento individual



g. Facilita a administração de baselines (linha de controle para saber as últimas versões dos módulos utilizadas para confeccionar uma versão do executável funcional)

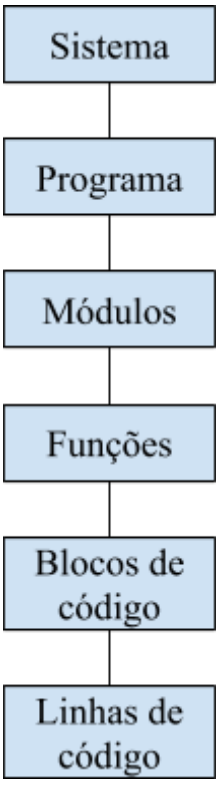


Princípios de Modularidade

1. Módulo

    a. Definição física: unidade de compilação independente
    b. Definição lógica: trata de um único conceito

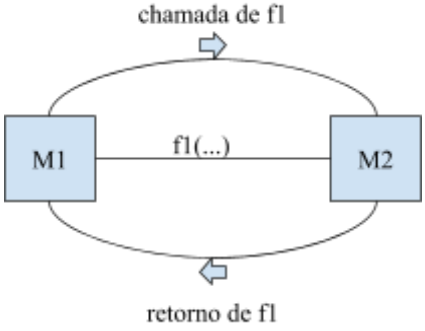
2. Abstração de Sistema
    a. Definição: abstrair é o processo de considerar apenas o que é necessário em uma situação e descartar com segurança o que não é necessário (escopo).
    b. Níveis de abstração:



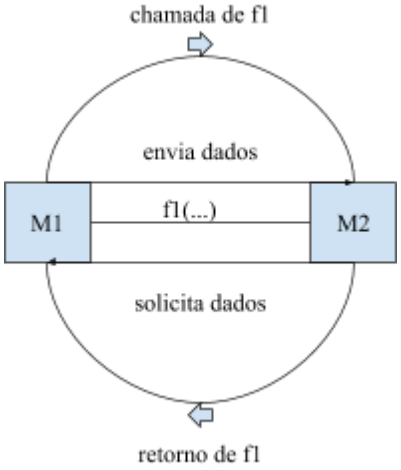
Obs$_1$.: Artefato: item com identidade própria criado dentro de um processo de desenvolvimento. Pode ser versionado.

Obs$_2$.: Construto (build): um resultado apresentável que pode ser executado, mesmo que incompleto.

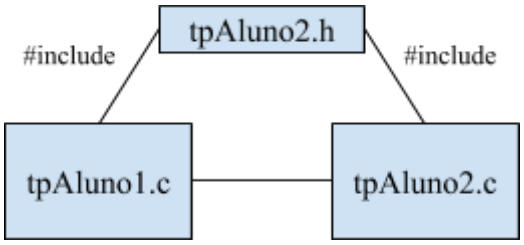
3. Interface
    a. Definição: mecanismo de troca de dados, estados ou eventos entre elementos da aplicação.
    b. Exemplos de interface:
        i. Arquivo (entre sistemas)
        ii. Funções de acesso (entre módulos)
        iii. Passagem de parâmetros (entre funções)
        iv. Variável global (entre blocos)
    c. Relacionamento cliente-servidor



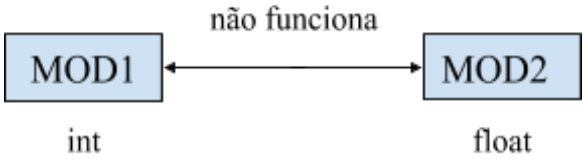
Caso especial: callback (quando o servidor solicita dados adicionais do cliente para completar os dados que já possuía de uma chamada de função anterior)



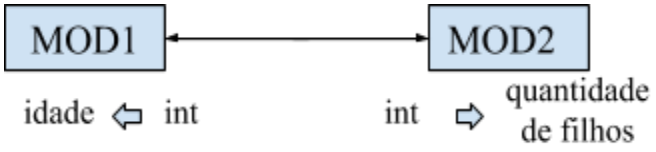
d. Interface fornecida por terceiros



e. Interface em detalhe



i. Sintaxe: regras



ii. Semântica: significado

f. Análise de interface

i. ***tpDadosAluno**** obterDadosAluno (int mat)

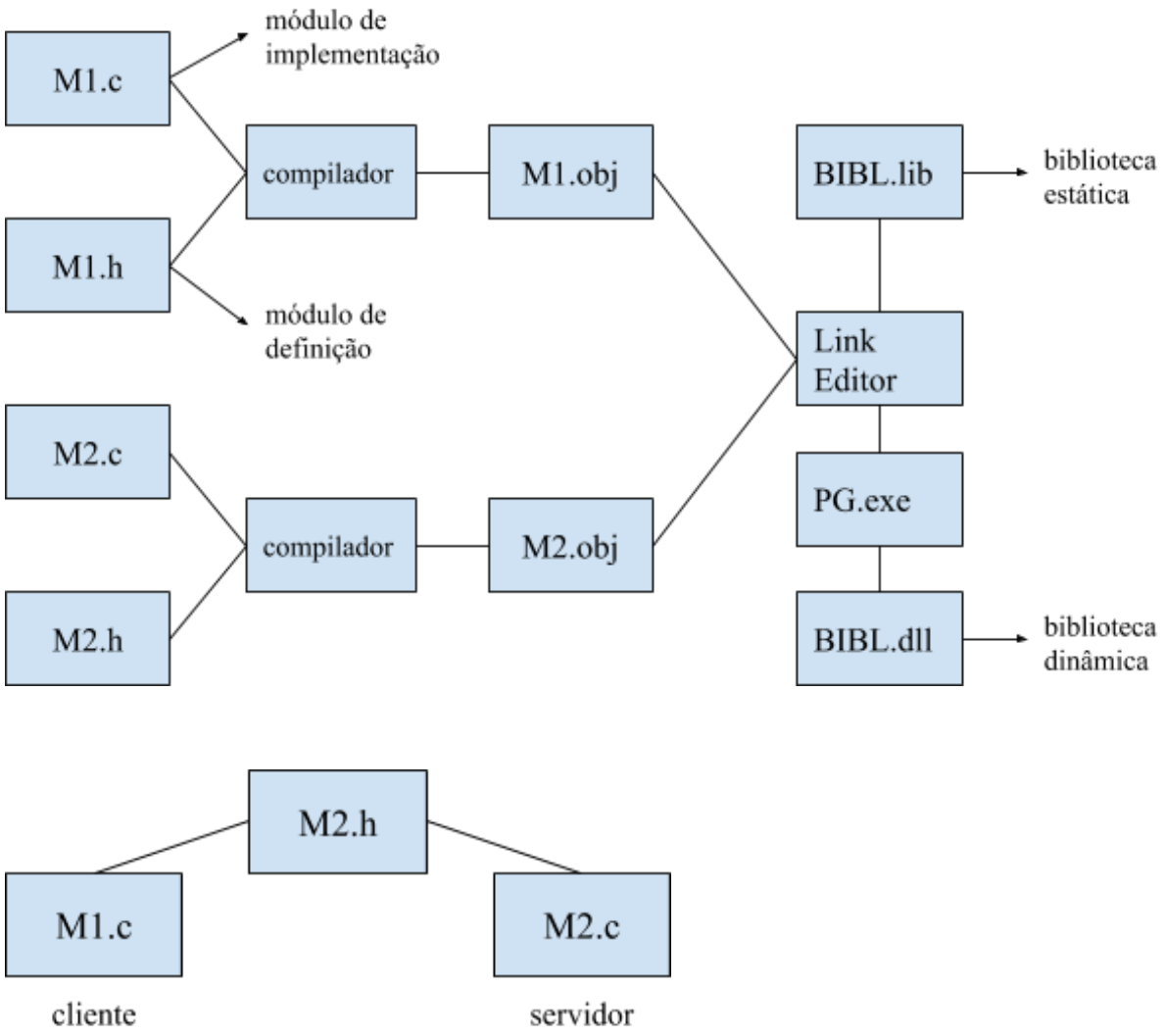
(protótipo ou assinatura de função de acesso)

ii. interface esperada pelo cliente: ponteiro para dados válidos do aluno correto ou null.

iii. interface esperada pelo servidor: inteiro válido representando a matrícula de um aluno.

iv. interface esperada por ambos: tpDadosAluno.

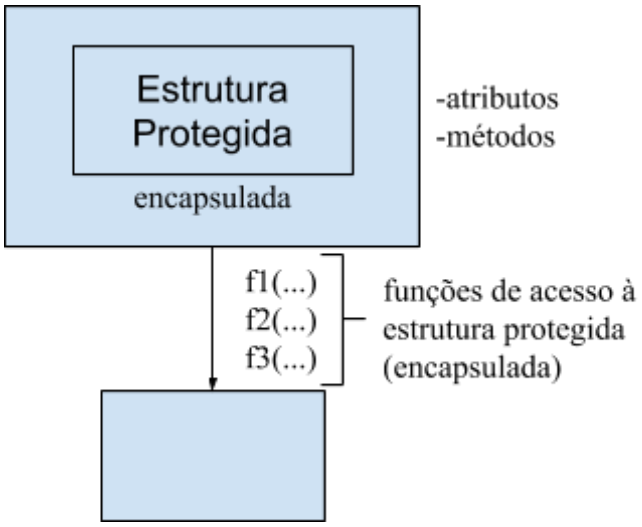
4. Processo de Desenvolvimento

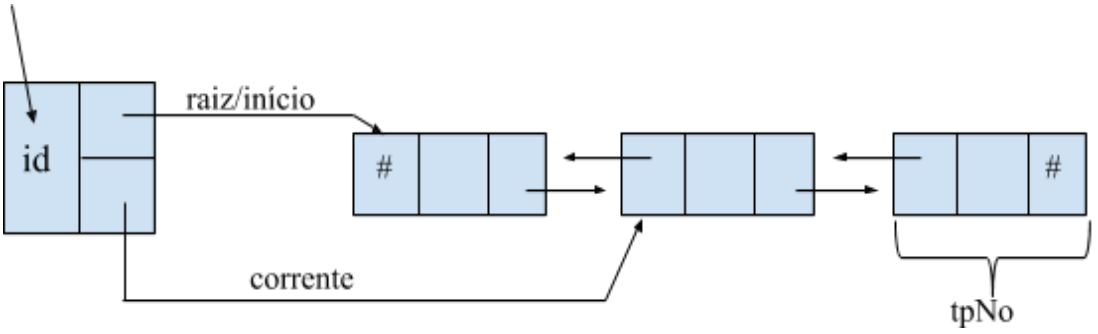


5. Módulo de Definição (.h)
    a. Interface do módulo
    b. Contém os protótipos das funções de acesso, interfaces fornecidas por terceiros (ex.: tpDadosAluno do item 3.f)
    c. Documentação voltada para o programador do módulo cliente

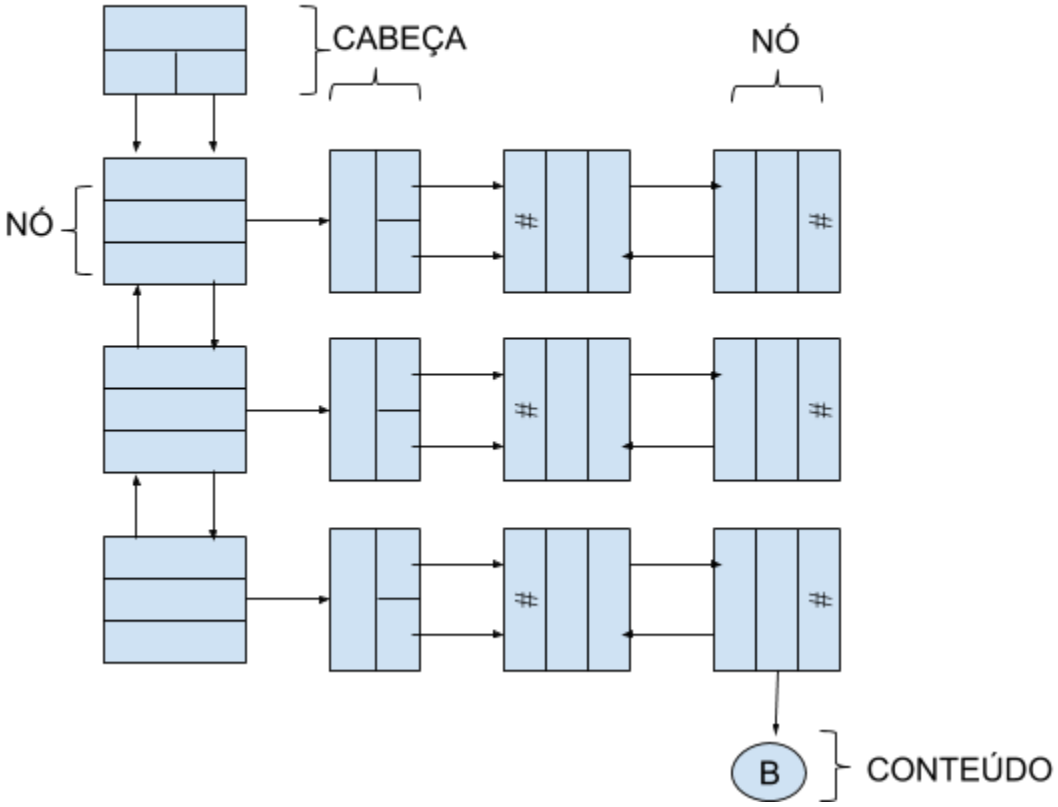
6. Módulo de Implementação (.c)
    a. Código das funções de acesso
    b. Códigos e protótipos das funções internas
    c. Variáveis internas ao módulo
    d. Documentação voltada para o programador do módulo servidor

7. Tipo Abstrato de Dados
    a. Estrutura encapsulada em um módulo que somente é conhecida pelos módulos-cliente através das funções de acesso disponibilizadas na interface.

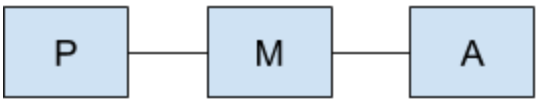




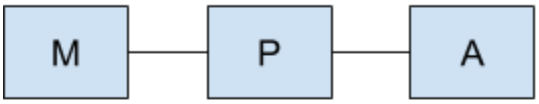
Matriz de Listas (3x2)



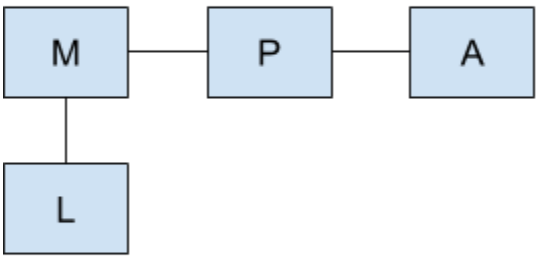

```
Exemplo de criação de matriz 2x2

criarLista(p1) /* Cabeça */
criarLista(p2)
inserirNo(p2, NULL) /* Nó */
inserirNo(p2)
inserirNo(p1, p2) /* Ligando p2 a p1 */
criarLista(p3)
inserirNo(p3)
inserirNo(p3)
inserirNo(p1, p3)
```



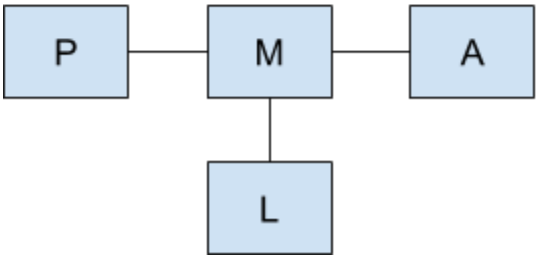
Matriz de conteúdo árvore e de estrutura genérica.



Matriz de conteúdo genérico e estrutura genérica.



Matriz de conteúdo genérico e estrutura lista ou matriz de conteúdo lista e estrutura genérica.



Matriz de conteúdo árvore e estrutura lista.

8. Propriedades da modularização
    a. Encapsulamento
    b. acoplamento
    c. Coesao
9. Encapsulamento
    a. Propriedade relacionada com a proteção dos elementos que compõem um módulo.
    b. Objetivos:
        i. Facilitar a manutenção
        ii. Impedir utilização indevida da estrutura do módulo
    c. Outros tipos de encapsulamento:
        i. De documentação:
            - interna → módulo de implementação (.c)
            - externa → módulo de definição (.h)
            - de uso → manual do usuário
        ii. De código:
            - blocos de código visíveis apenas:
                - dentro de módulo

        - dentro de outro bloco de código (ex.: conjunto de comando dentro de um for)
        - código de uma função
    - de variáveis
        - private (objeto), public, global, global static (módulo), protected (estrutura de herança), static (classe), local, etc.

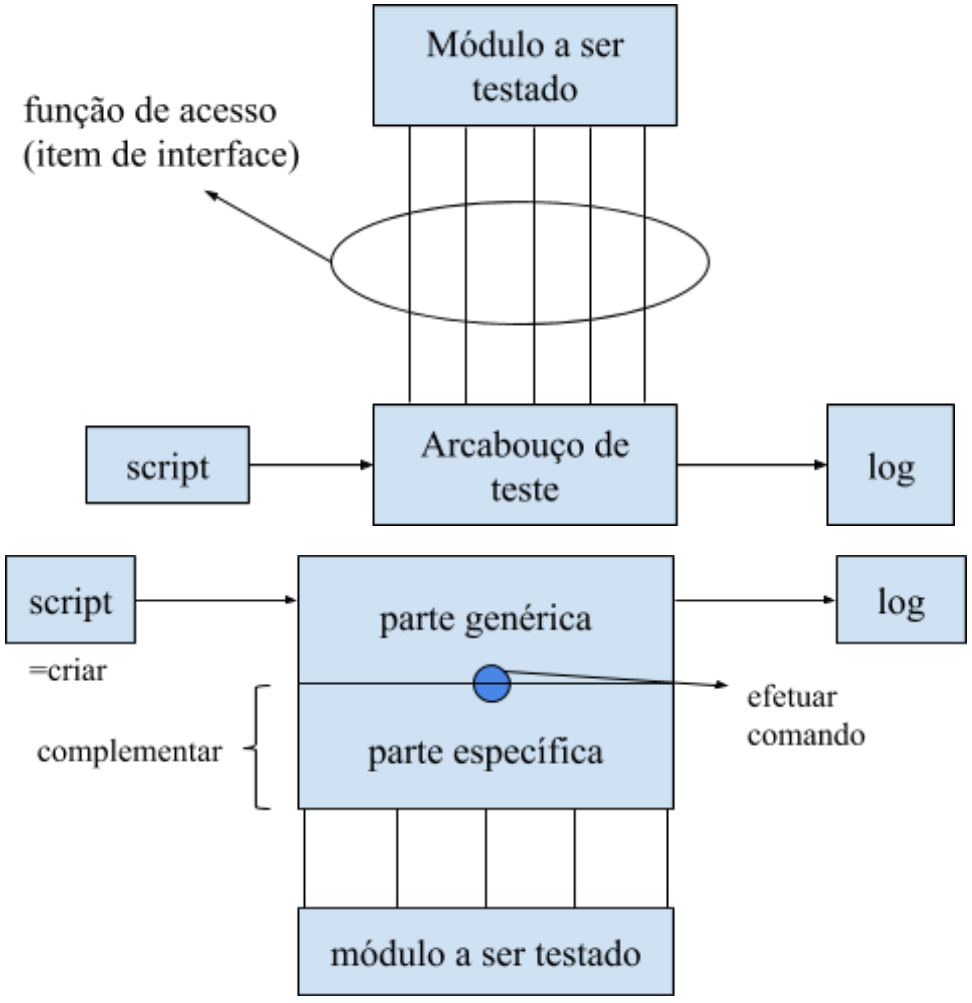
10. Acoplamento
    a. Propriedade relacionada com a interface entre os módulos.
    b. Conector → item de interface
    c. Ex.: função de acesso, variável global,...
    d. Critérios de qualidade:
        i. quantidade de conectores
            - necessidade (tudo é útil?) x suficiência (falta algo?)
        ii. tamanho do conector (ex.: quantidade de parâmetros de uma função)
        iii. complexidade do conector
            - explicação em documentação
            - utilização de mnemônicos (nomes de variáveis autoexplicativos)

11. Coesão
    a. Propriedade relacionada com o grau de interligação dos elementos que compõem um módulo
    b. Níveis de coesão:
        i. incidental (quando são usados conceitos sem nenhuma relação em um mesmo módulo) → pior caso
        ii. coesão lógica → elementos logicamente relacionados
        iii. temporal → itens que funcionam em um mesmo período de tempo
        iv. procedural → itens em sequência
        v. funcional
        vi. abstração de dados → um único conceito (ex.: T.A.D.)

Teste Automatizado

1. Objetivo
    a. Testar de forma automática um módulo recebendo um conjunto de casos de teste na forma de um script e gerando um log de saída com a análise entre o resultado esperado e o obtido.
    b. Observação: 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



    a. parte genérica → ArcaboucoTeste.lib
    b. parte específica → TESTARV.c
    c. funções de acesso do módulo a ser testado → ARVORE.h
    d. módulo a ser testado → ARVORE.c

3. Script de teste
    a. // → comentário
    b. == → caso de teste → testa determinada situação
    c. = → comando de teste → associado a uma função de acesso
    d. Observação: teste completo → casos de teste para todas as condições de retorno de cada função de acesso do módulo (com exceção de condição de retorno de estouro de memória)

4. Log de saída
    a. 33 == Verificar assertivas de entrada de irdir → linha e caso testado
    b. >>> 1 Linha: 26 Retorno errado ao inserir à esquerda. Deveria ser: 1 É: 4 → contador de erros, linha e descrição do erro
    c. <<< 0 Linha: 31 Falha esperada foi recuperada. → comando recuperar (decrementa 1 do contador de erros)

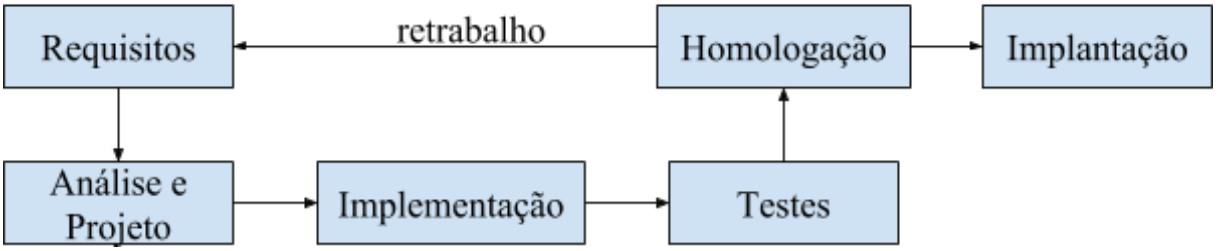
5. Parte específica
    a. A parte específica que necessita ser implementada para que o framework (arcabouço) possa acoplar na aplicação chama-se hotspot. (ex.: TESTARV.c)

Processo de Desenvolvimento em Engenharia de Software

1. Processo

- Responsabilidades do líder de projeto:
    - projeto
        - tamanho (ponto de função: dificuldade de fazer um tarefa; cobrado por processo elementar)
        - esforço
        - recursos
        - prazo
    - estimativa (estimar um tempo perto do real sempre que possível)
    - planejamento
    - acompanhamento (verificar a situação do projeto e preparar para o caso de ter de avisar ao cliente sobre atraso ou adiantamento)

2. Etapas de um projeto



    a. Requisitos
        i. elicitação (o que o cliente deseja que seja feito)
        ii. documentação
        iii. verificação (checar se o que foi elicitado é possível fazer)
        iv. validação
    b. Análise e projeto
        i. projeto lógico (modelagem de dados)

ii. projeto físico (como implementar)

c. Implementação
    i. programas
    ii. teste unitário
d. Testes
    i. teste integrado (testa a aplicação como um todo)
e. Homologação (teste do cliente para verificar se o projeto atende suas necessidades específicas)
    i. sugestão (pode gerar um retrabalho remunerado)
    ii. erro (pode gerar um retrabalho não remunerado)
    iii. Obs.: Em caso de haver erros no programa e novas sugestões por parte do cliente, pode ser feito um novo acordo de modo que seja pago e entregue em uma nova data de entrega.
f. Implantação

3. Outros detalhes
    a. Gerência de configuração: responsável pela baseline e instalação de máquinas.
    b. Gerência de qualidade de software: mensura se está sendo mantida a qualidade do projeto, além de supervisionar as etapas do projeto.

Especificação de Requisitos

1. Requisito
    a. O que tem que ser feito? (na etapa de análise e projeto)
    b. Nunca como deve ser feito

2. Características do requisito
    a. curtos e genericos
    b. linguagem natural

3. Fases da Especificação
    a. Elicitação: captar informações do cliente para realizar a documentação do sistema a ser desenvolvido.
    b. Técnicas de elicitação:
        i. entrevista
        ii. brainstorm
        iii. questionário
    c. Documentação
        i. requisitos descritos em itens diretos
        ii. uso da língua natural (cuidado com ambiguidade)
        iii. dividir requisitos em seus diversos tipos
            - tipos de requisitos
                - requisitos funcionais: o que deve ser feito em relação a informatização das regras de negócio.
                - requisitos não-funcionais: propriedades que a aplicação deve ter e que não estão diretamente relacionadas com as regras de negócio. Exemplos:
                    - segurança (ex.: login e senha)
                    - tempo de processamento (ex.: as consultas não podem demorar mais do que 5 segundos)
                    - disponibilidade (ex.: 24x7 -> 24h por 7 dias de semana)

- requisitos inversos: é o que não é para fazer (forma de resolver ambiguidades)

d. Verificação (checar se tudo o que foi pedido é computável)
   i. a equipe técnica verifica se o que está descrito na documentação é viável de ser desenvolvido

e. Validação
   i. cliente valida a documentação

4. Exemplos de Requisitos

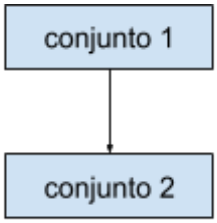
a. Bem formulados:
   i. a tela de resposta da consulta de aluno apresenta nome e matrícula
   ii. todas as consultas devem retornar respostas no máximo em 2 segundos

b. Mal formulados:
   i. o sistema é de fácil utilização
   ii. a consulta deverá retornar uma resposta em um tempo reduzido
   iii. a tela mostra seus dados mais importantes

Modelagem de Dados

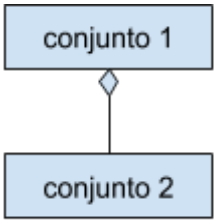
1. Um modelo equivale a n exemplos.

2. Notação:



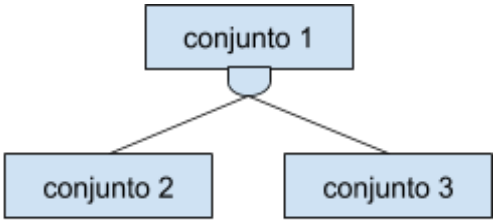
referência

(quando um elemento do conjunto 1 aponta para outro do conjunto 2)
Ex.: estrutura dinâmica



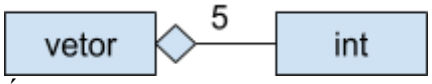
agregação

(várias instâncias do conj. 1 estão agregadas ao conj. 2)
Ex.: vetor

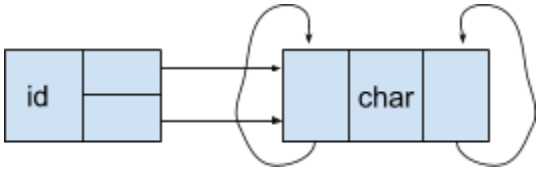


particionamento

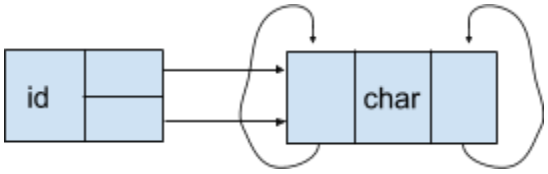
3. Exemplos:
   a. Vetor de 5 posições que armazena inteiros.



   b. Árvore binária com cabeça que armazena caracteres.



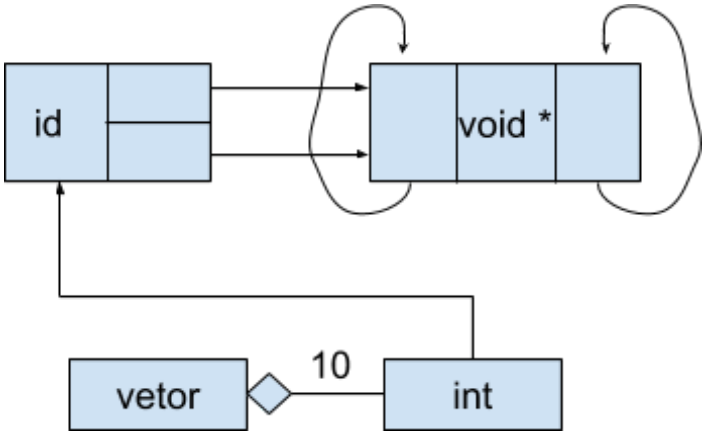
   c. Lista duplamente encadeada com cabeça que armazena caracteres.



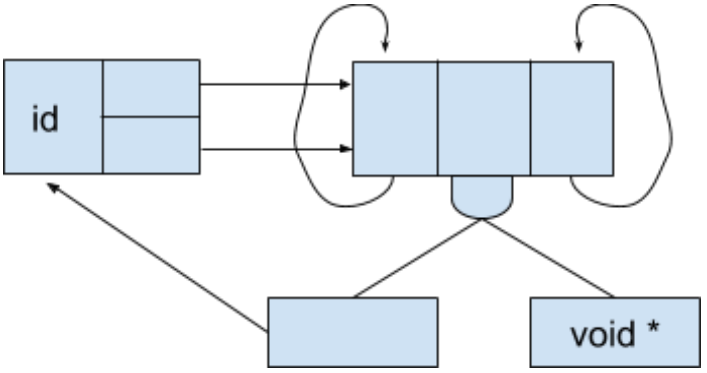
- Assertivas Estruturais: são regras utilizadas para desempatar dois modelos iguais. Estas regras complementam o modelo, definindo características que o desenho não consegue representar.
  - Lista:
    - Se pCorr→pAnt ≠ nulo, então pCorr→pProx == pCorr.
    - Se pCorr→pProx ≠ nulo, então pCorr→pProx->pAnt == pCorr.

- Árvore:
  - Um ponteiro de uma subárvore à esquerda nunca referencia um nó de uma subárvore à direita.
  - pAnt e pProx de um nó nunca apontam para o pai.

d. Vetor de listas duplamente encadeadas com cabeça e genéricas.

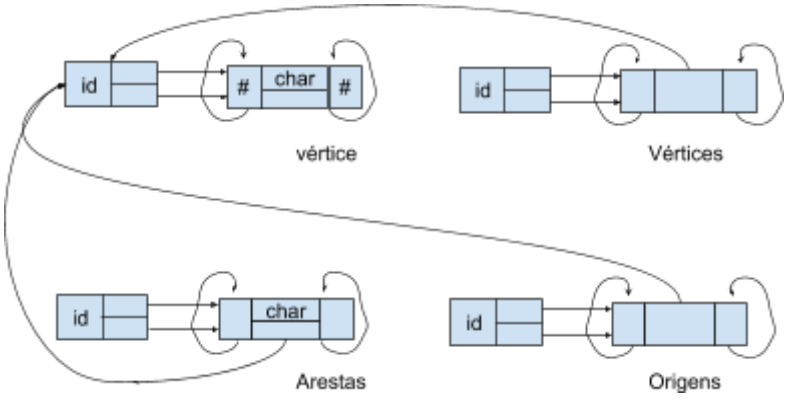


e. Matriz tridimensional genérica construída com listas duplamente encadeadas com cabeça.



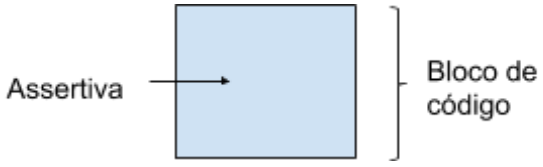
Assertiva Estrutural: a matriz comporta apenas 3 dimensões, sendo o nó da lista mais interna preenchido com um void *.

f. Grafo criado com listas.



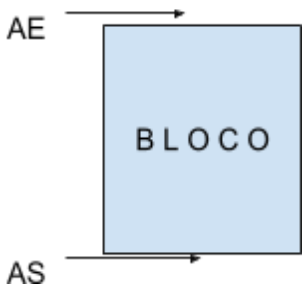
Assertivas

1. Definição
    a. qualidade por construção: qualidade aplicada a cada etapa do desenvolvimento de uma aplicação.
    b. Assertivas: regras consideradas válidas em determinado ponto do código (se chegar a uma assertiva sem ter erros significa que até aquela assertiva manteve-se a regra).



2. Onde aplicar assertivas
    a. argumentação de corretude (argumenta se o bloco de código funciona)
    b. instrumentação (criar um bloco de código para uma assertiva)
    c. trechos complexos em que é grande o risco de erros

3. Assertivas de Entrada e Saída
    a. AE: assertivas de entrada
    b. AS: assertivas de saída
    c. A assertiva deve tratar de regras envolvendo dados e ações tomadas



4. Exemplos
    a. Excluir nó corrente intermediário de uma lista duplamente encadeada.
        i. AE:

- ponteiro corrente referencia o nó a ser excluído e este é intermediário
- a lista existe
- … (quantas mais achar necessárias)

ii. AS:

- o nó corrente foi excluído
- corrente aponta para o anterior

Implementação de Programação Modular

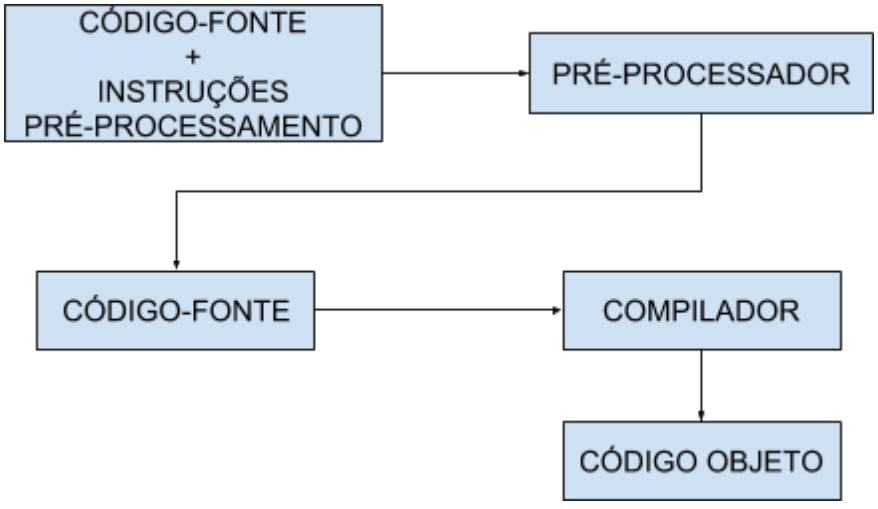
1. Espaço de Dados
   a. São áreas de armazenamento
      i. alocados em um meio
      ii. possui um tamanho
      iii. possui um ou mais nomes de referência
   b. Exemplos:
      i. A[J] → J-ésimo elemento do vetor A
      ii. ptAux * → espaço de dados apontado para ptAux
      iii. ptAux → espaço de dados que contém endereço
      iv. (ObterElemento(int id)).Id → subconjunto ID presente na estrutura ou ObterElemento(int id)->Id apontada pelo retorno da função

2. Tipo de Dados
   a. Determina a organização, codificação, tamanho em bytes e conjunto de valores específicos
   b. Obs.: Um espaço de dados precisa estar associado a um tipo para que possa ser interpretado pelo programa desenvolvido em linguagem tipada.
   c. Obs.: Conversão de tipos não é igual a imposição de tipos

3. Tipos Básicos
    a. typedef
    b. enum
    c. struct
    d. union
    e. Extra: typecast
        i. Exemplo: (int *) malloc(sizeof(int))

4. Declaração e Definição de Elementos
    a. Definir → alocar espaço e amarrar a um nome (bind)
    b. Declarar → associar um espaço a um tipo
    c. Obs.: Quando o tipo é computacional, ocorre simultaneamente a declaração e a definição.

5. Implementação em C e C++
    a. Declarações e definições de nomes globais exportados pelo módulo servidor
        i. Ex.: int a ; int F(int b) ;
    b. Declarações externas contidas no módulo cliente e que somente declaram o nome sem associá-lo a um espaço de dados.
        i. Ex.: extern int a ; extern int F(int b) ;
    c. Declarações e definições de nomes globais encapsulados no módulo
        i. Ex.: static int a ; static int F(int b) ;
6. Pré-processamento



    a. #define nome valor
        i. substitui nome por valor

b. #undef nome
    i. o nome deixa de estar ligado às substituições feitas anteriormente
c. #if defined(nome) ou
   #ifdef nome textoV

   #else textoF

   #endif

    i. se o nome está definido, ele recebe um textoV, caso contrário, recebe textoF.
d. #if !defined(nome) ou #ifndef nome
e. #include <arquivo> ou #include “arquivo”
    i. inclui todo o arquivo texto
f. Exemplo:
   #if !defined (EXEMP_MOD)
   #define EXEMP_MOD
   .
   . (texto do .h)
   .
   #endif

g. 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       |       int vetor[7]
#include “M1.h”          |               = {1, 2, 3, 4, 5, 6, 7};
#undef EXEMP_OWN        |
```

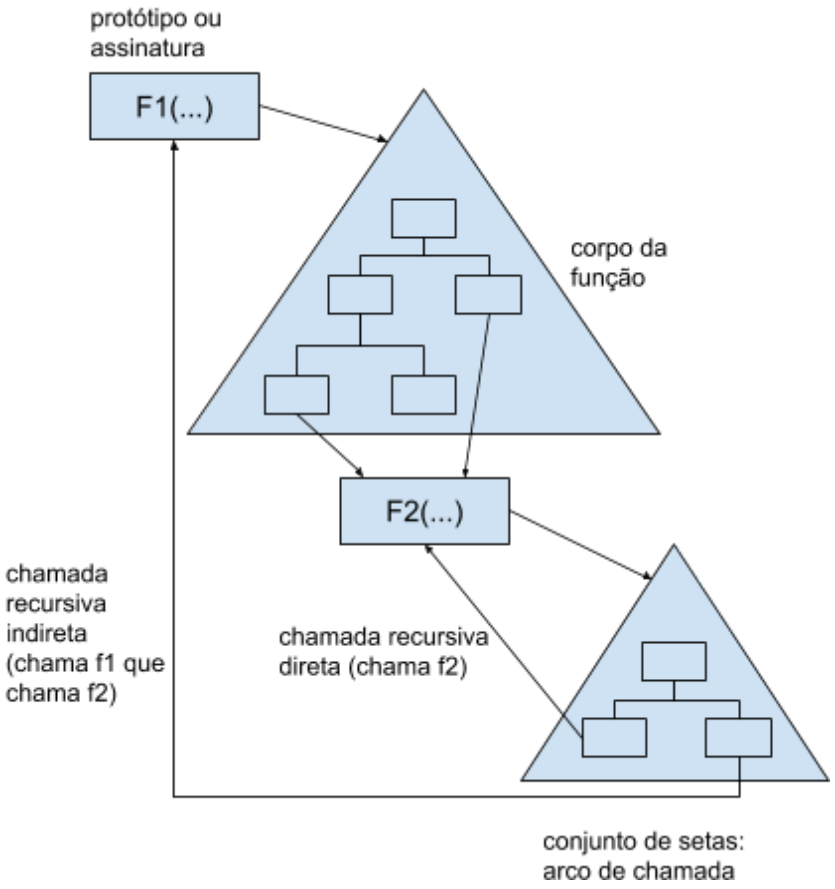
M2.c:

```
#include “M1.h”          |       extern int vetor[7];
```

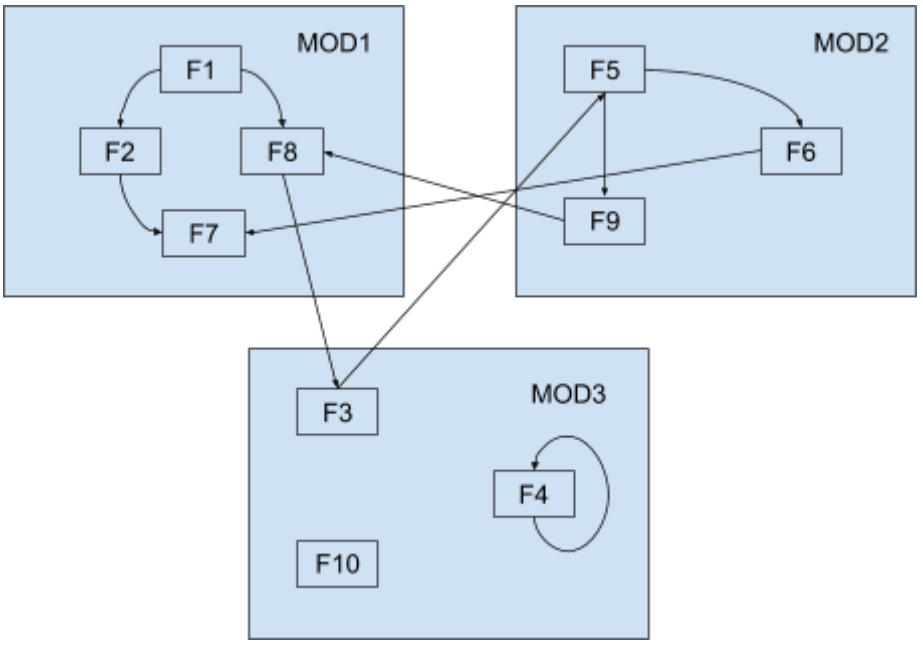
Estrutura de Funções

1. Paradigma
    a. forma de programar
        i. procedural: programação estruturada
        ii. orientada a objetos: programação orientada a objetos
            programação modular (mesmo utilizando uma linguagem não orientada a objetos)

2. Estrutura de Funções



3. Estrutura de Chamadas



a. F4 → F4

chamada recursiva direta

b. F9 → F8 → F3 → F5 → F9

chamada recursiva indireta

c. F10

função morta (em outra aplicação ela pode ter utilidade)

d. F8 → F3 → F5 → F6 → F7

dependência circular entre módulos

e. Supondo: F6 → F7

arco de chamada

4. Função

a. É uma porção auto-contida de código. Possui:

i. um nome

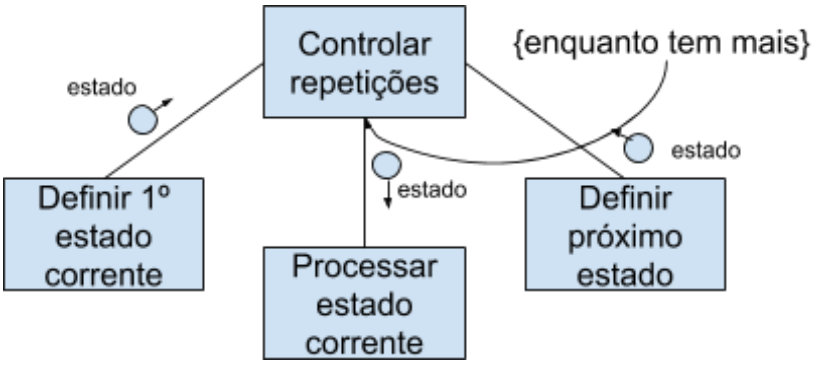
ii. uma assinatura

iii. um ou mais (ponteiro de função) corpos de código

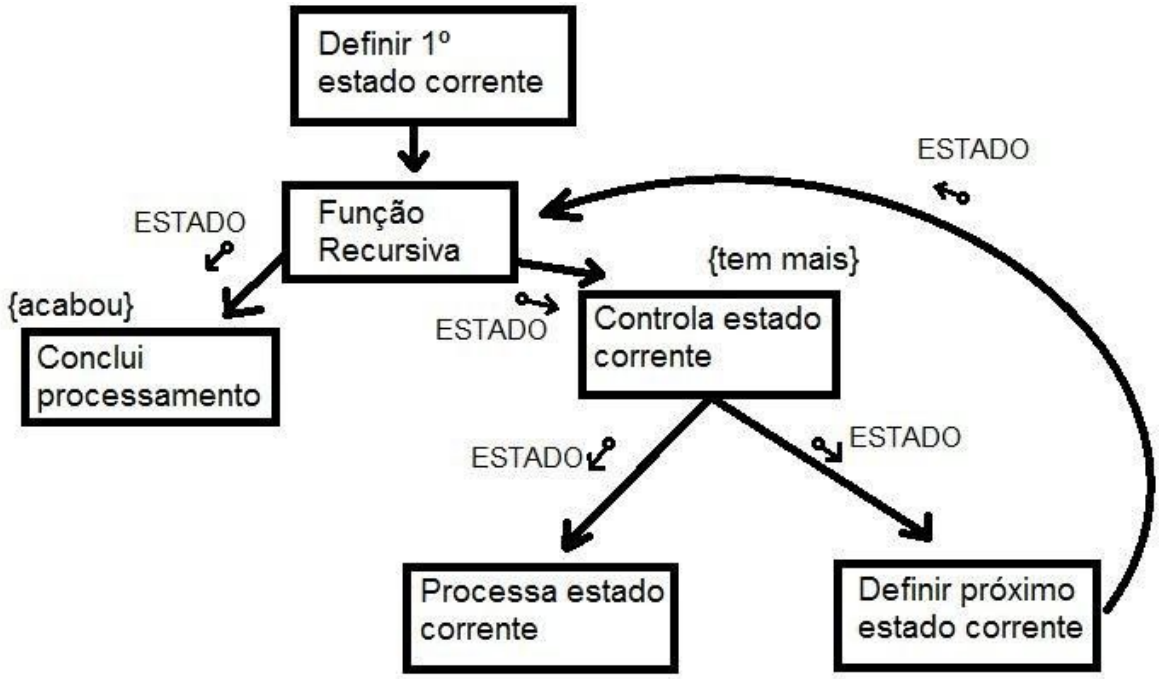
5. Especificação da Função
    a. Objetivo (pode ser igual ao nome)
    b. Acoplamento (parâmetros e condições de retorno)
    c. Condições de acoplamento (assertivas de entrada e assertivas de saída)
    d. Interface com o usuário (mensagens, saídas em tela para o usuário)
    e. Requisitos (o que deve ser feito)
    f. Hipótese: são regras pré-definidas que assumem como válida uma determinada ação ocorrendo fora do escopo, evitando assim o desenvolvimento de códigos desnecessários.
    g. Restrições: são regras que limitam as escolhas das alternativas de desenvolvimento para uma determinada solução.

6. Housekeeping
    a. Código responsável por liberar componentes e recursos alocados a programas ou funções ao terminar a execução.

7. Repetição



8. Recursão

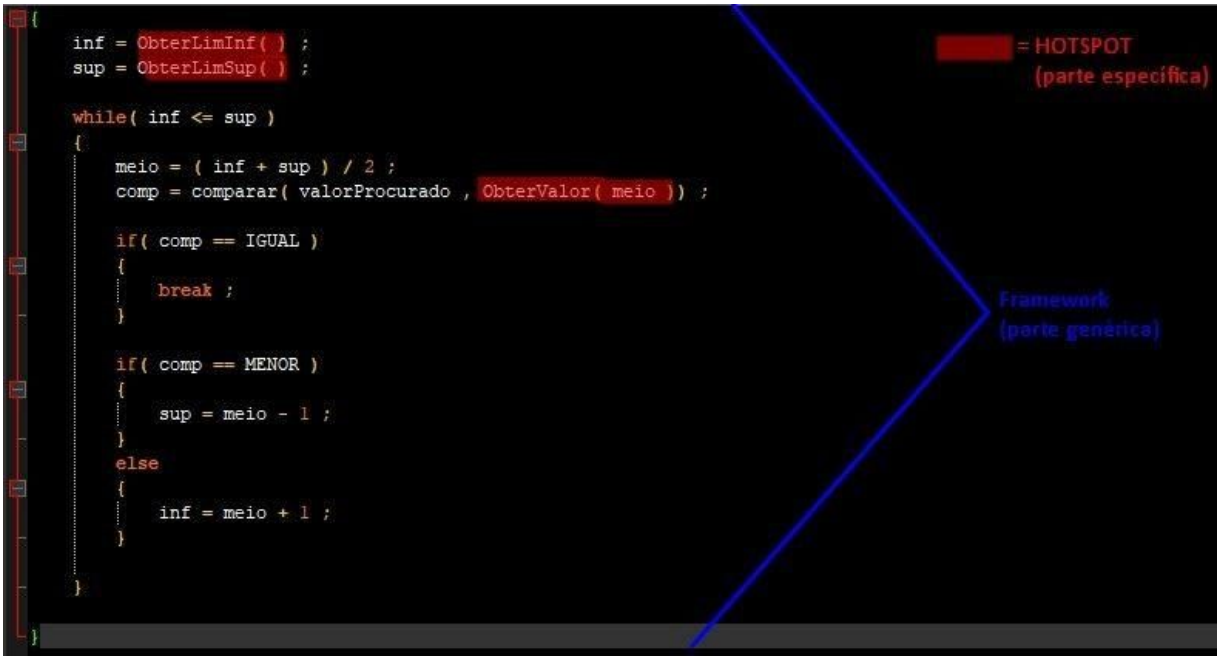


9. Estado
    a. descritor de estado (variável utilizada para descrever o estado de uma repetição ou recursão)
        i. conjunto de dados que definem um estado. Ex.: índice numa pesquisa em vetor estado.
    b. estado

i. valoração do descritor. Ex.: i = 0

c. Obs1.: Não é necessariamente observável.
Ex.: cursor de posicionamento de arquivo. Obs2.: Não precisa ser único.
Ex.: limiteInferior e limiteSuperior de uma pesquisa binária.

10. Esquema de Algoritmo (parte genérica)



a. Esquemas de algoritmo permitem encapsular as estruturas de dados utilizadas. É correto, independe de estruturas e é incompleto (precisa ser instanciado). Normalmente ocorrem em:
- programação orientada a objetos
- frameworks

b. Se o esquema estiver correto e o hotspot tiver assertivas válidas, então o programa está correto.

11. Parâmetros do tipo ponteiro para função

```c
#include <stdio.h>

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 base , float altura ))
{
    ...
    printf( "%f" , func( valor1 , valor2 )) ;
    ...
}

int main( void )
{
    ...
    CondRet = ProcessaArea( 5 , 2 , areaQuad ) ;
    ...
    CondRet = ProcessaArea( 3 , 3 , areaTri  ) ;
    ...
    return 0 ;
}
```