Guilherme Pasqualette – 1610596

Resumo da matéria de Programação Modular

1 – ***INTRODUÇÃO***

- *Vantagens da programação modular:*
  - Ela vence barreiras de complexidade de um software;
  - Facilita o trabalho em grupo, após divisão das tarefas no grupo;
  - Possibilita o reuso;
  - Facilita a criação de um acervo, já que diminui a quantidade de novos programas;
  - Desenvolvimento incremental;
  - Aprimoramento individual de módulos;
  - Facilita a administração de baselines, que são versões "bases" do programa que funcionam de acordo com o pedido, as quais os programadores podem utiliza-las novamente caso tenha algum problema com uma versão posterior.

2 – ***PRINCÍPIOS DE MODULARIDADE***

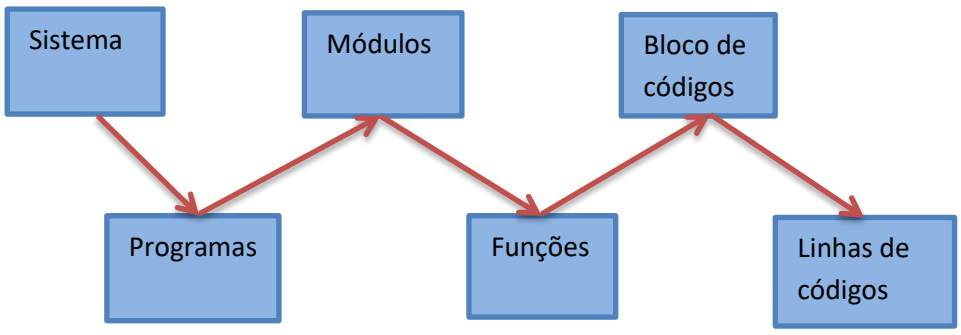
I) Módulo:

- A definição física de módulo consiste nele ser uma unidade de compilação independente (.c);
- E a definição lógica é baseada em um único conceito o qual o módulo trata.

II) Abstração de Sistema:

- Abstração é o procedimento de considerar somente o que é preciso ou não em diversas situações e, o que descartar com segurança nas mesmas situações.

i) Níveis de abstração:



Alguns conceitos importantes:

1) Artefato: Item com identidade própria, criado dentro de um processo de desenvolvimento podendo possuir baselines;

2) Construto (build): Artefato que pode ser executado, estando incompleto ou não.

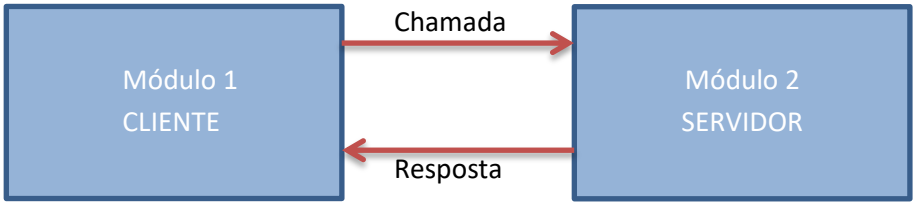
III) Interface:

- É o mecanismo de troca de dados, estados, eventos entre elementos de um mesmo nível de abstração;

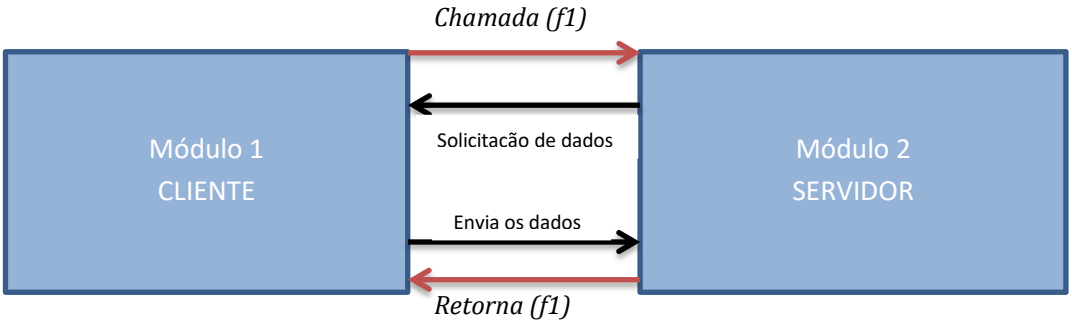
i) <u>Exemplo:</u>

- Arquivo entre sistemas
- Funções de acesso entre módulos
- Passagem de parâmetro entre funções
- Variável global entre blocos

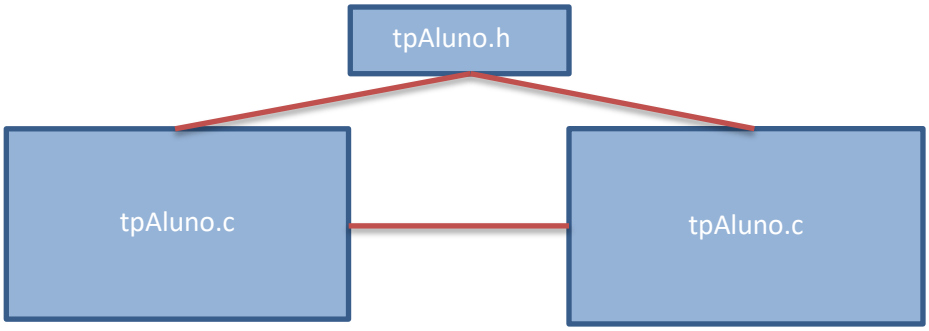
ii) <u>Relacionamento cliente-servidor:</u>



Caso Especial: Callback -> É quando o servidor, após o cliente enviar dados para ele, retorna uma resposta para o usuário, avisando sobre uma possível falta de certos dados essenciais para o funcionamento do programa.



iii) <u>Interface fornecida por terceiros:</u>



Importante! Evitar duplicidade de código!!!

iv) <u>Interface Em Detalhe:</u>

- Todo resultado trazido pelo compilador não depende propriamente dos blocos de códigos dentro dos módulos, mas também depende da sintaxe (regras) e da semântica (significado) inserido neles.

Exemplo:

- Sintaxe: Jamais poderá pedir para uma função do tipo Arvore pedir para retornar um valor Int. Isso resultará em um erro de compilação.

- Semântica: Quando possui duas variáveis (v1 & v2) do tipo Float, em dois módulos diferentes (m1 & m2), uma pra área e outra tempo, respectivamente, não se pode utilizar uma delas no módulo onde não pertencem, pois o programa funcionará e, provavelmente, dará um resultado equivocado para o cliente.

v) <u>Análise de Interface:</u>

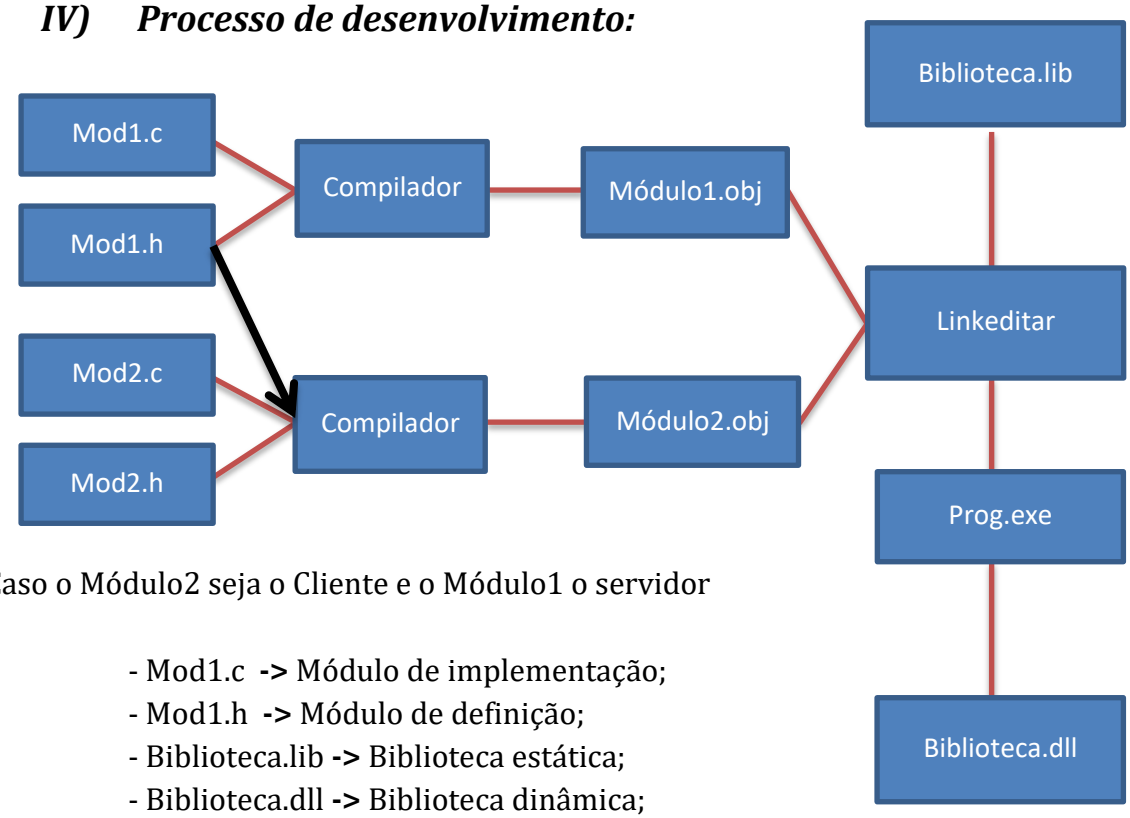
No caso de um protótipo de função de acesso: tpAluno *obterMatricula (int Mat):

- A interface esperada pelo cliente **->** Ponteiro para os dados válidos do aluno correto ou NULL;

- A interface esperada pelo servidor **->** Valor inteiro apto para representar a matrícula do aluno;

- A interface esperada pelo servidor e cliente **->** tpAluno.

IV) Processo de desenvolvimento:



Caso o Módulo2 seja o Cliente e o Módulo1 o servidor

- Mod1.c **->** Módulo de implementação;
- Mod1.h **->** Módulo de definição;
- Biblioteca.lib **->** Biblioteca estática;
- Biblioteca.dll **->** Biblioteca dinâmica;

V) Bibliotecas estáticas e dinâmicas:

i) <u>Estática:</u>

-Vantagem **->** Ela já é acoplada, em termo de linkedição, à aplicação executável;

-Desvantagem **->** Existe uma cópia dessa biblioteca para cada executável alocado na memória que a use;

ii) <u>Dinâmica:</u>

- Vantagem **->** Só é carregada à uma só instancia;

- Desvantagem **->** A DLL precisa estar na máquina, preferencialmente do cliente, para o executável funcionar;

VI) Módulo de definição(.h):

- Interface do módulo;

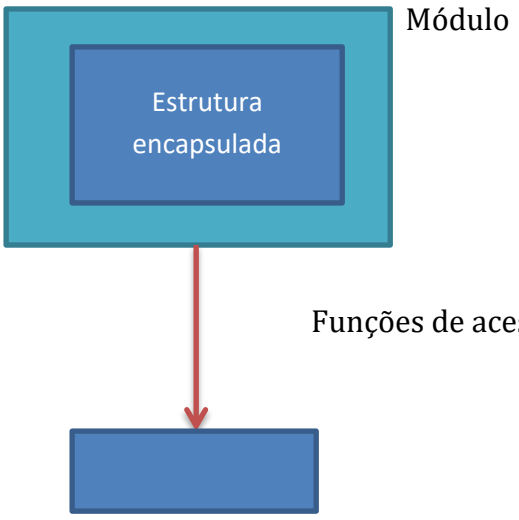
-Os protótipos das funções de acesso estão contidos nele, é uma interface fornecida por terceiros (tpAluno);

- Sua documentação é voltada para o implementador do módulo Cliente;

VII) Módulo de implementação:

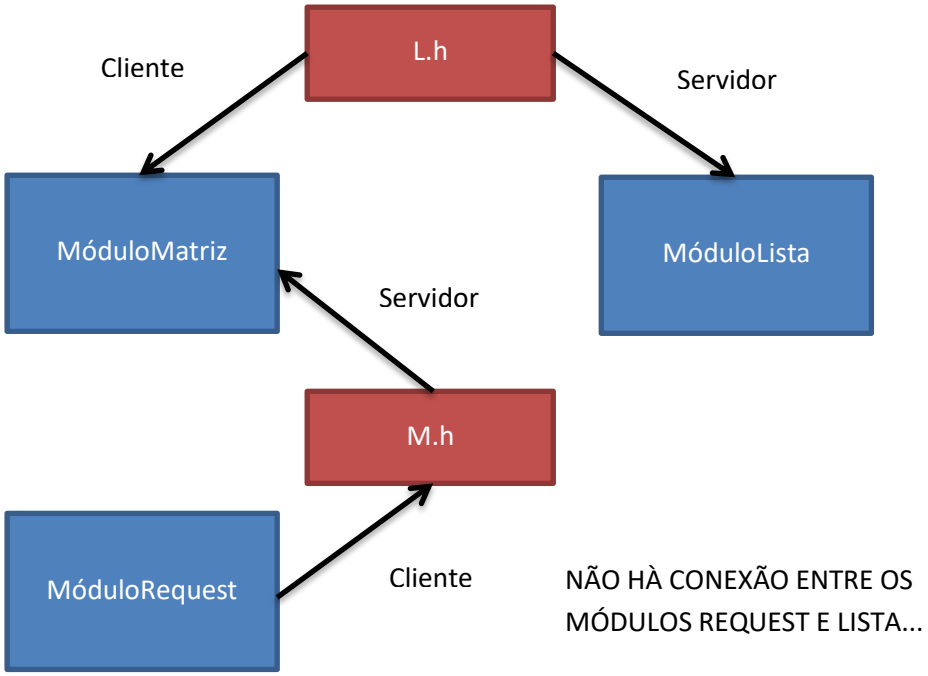
- Contém o código das funções de acesso;
- Códigos e protótipos de funções próprias do módulos (static);
- Contém variáveis internas ao módulo;
- Documentação voltada para o implementador do módulo servidor;

VIII) Tipo Abstratos de Dados:



Uma estrutura encapsulada em um módulo é apenas conhecida pelos módulos clientes a partir de funções de acesso disponibilizadas na interface.

Exemplo:
- Para um módulo que manipula uma matriz utiliza listas para criar suas colunas e linhas, o esquema estaria de acordo com o pedido:



Nesse caso, a solução seria utilizar ponteiros para a "cabeça" da estrutura. Logo, as funções de acesso tem que receber tal ponteiro a fim de indicar qual estrutura de dados será modificada, com isso, interface deverá disponibilizar um typedef para corresponder com o ponteiro.

IX) Encapsulamento:

- Definição: Proteção de elementos que compõem módulos.

i) <u>Objetivos</u>

- Facilitaria a manutenção por manter os erros confinados;
- Impediria a utilização ou alteração indevida da estrutura do módulo;

ii) <u>Outros tipos de encapsulamento:</u>

- De documentação
  - **->** Interna do módulo de implementação;
  - **->** Externa do módulo de definição;
  - **->** De uso do manual do usuário (README);
- De código – Para blocos de códigos visíveis somente
  - **->** Dentro do módulo;
  - **->** Dentro do outro bloco de código;
  - **->** Código de uma função;

- De variáveis

-> Private: Encapsulado no objeto;

-> Static: Encapsulado no módulo (Ou na classe no caso de Orientação a Objetos);

-> Local: Em um bloco de código;

X) Acoplamento:

- Propriedade relacionada com a interface entre os módulos.

i) <u>Conector:</u> Item de interface, a partir de funções de acesso e variáveis globais.

ii) <u>Critérios de Qualidade:</u>

- Quantidade de conectores **->** Necessidade X Suficiência

----- Há excesso ou falta deles? -----

- Tamanho do conector **->** Quantidade de parâmetros de uma função
- Complexidade do conector **->** Tem que haver explicação na documentação e utilização de mnemônios.

XI) Coesão:

- Propriedade relacionada com o grau de interligação dos elementos que compõem um módulo.

i) <u>Níveis de coesão:</u>

- Incidental **->** Pior coesão, pois não há relação praticamente entre os elementos;
- Lógico **->** Elementos logicamente relacionados;
- Temporal **->** Itens que funcionam em um mesmo período de tempo;
- Procedural **->** Itens em sequencia;
- Funcional **->** Abstração de dados, torna-los em um único conceito (TAD).

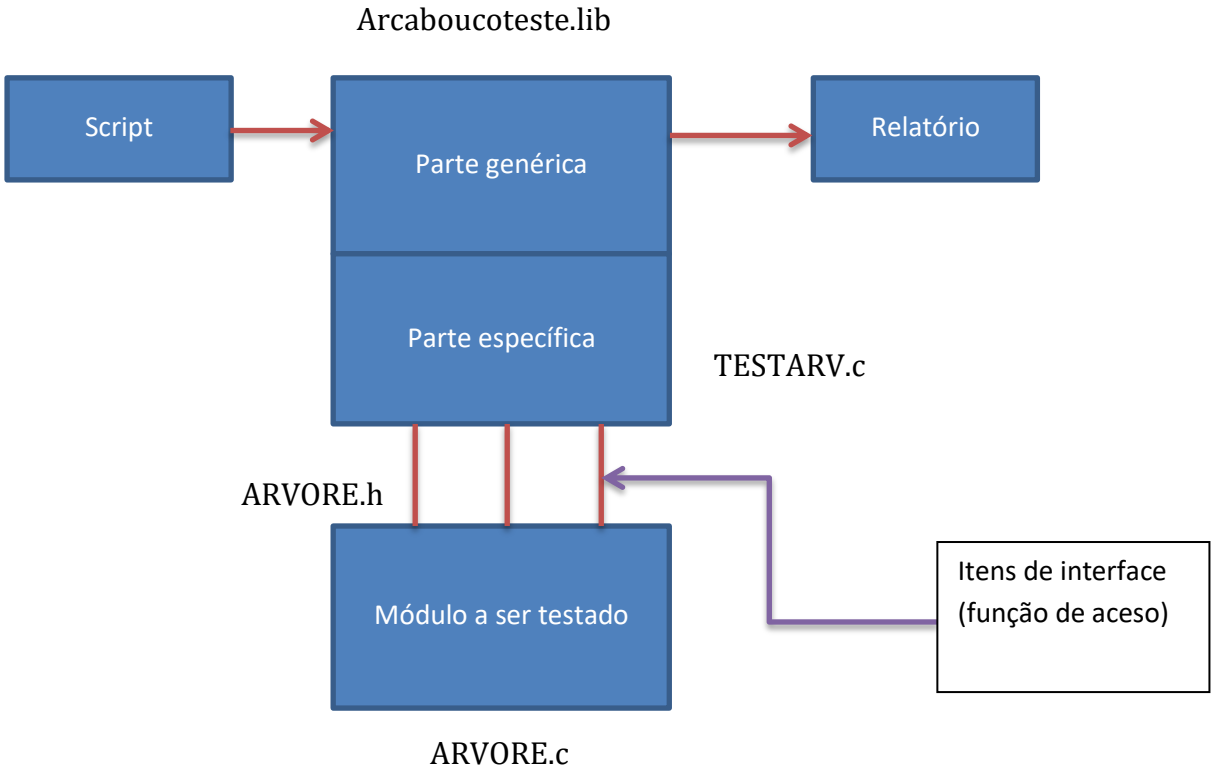
3 - ***TESTE AUTOMATIZADO***

I) Objetivo:

- Testar de forma automática um módulo recebendo um conjunto de casos de teste na forma de um script e gerando um relatório de saída com a análise entre o resultado esperado e o obtido.

OBS: A partir do primeiro retorno esperado diferente do obtido no relatório de saída, todos os resultados de execução de casos de teste não serão confiáveis.

II) Framework de testes:



III) Script de teste:

- "==" indica um determinado teste em uma situação;

- "=" indica um comando de teste, diretamente associado a uma função de acesso;

OBS: O teste completo consiste em testar todos os casos de teste para todas as condições de retorno de cada função de acesso do módulo (exceto para condições de estouro de memória).

IV) Relatório de saída:

== Caso 1

== Caso 2

== Caso 3

1 >> Função esperava 0 e retorna 1

0 << <- Recuperar (Parte genérica).

V) Parte específica:

- A parte específica, que necessita ser implementada a fim de que o framework (arcabouço) possa acoplar na aplicação, chama-se **HOTSPOT**.

Por exemplo: O módulo de implementação testArvore.c.

4 – PROCESSO DE DESENVOLVIMENTO DE ENGENHARIA DE SOFTWARE

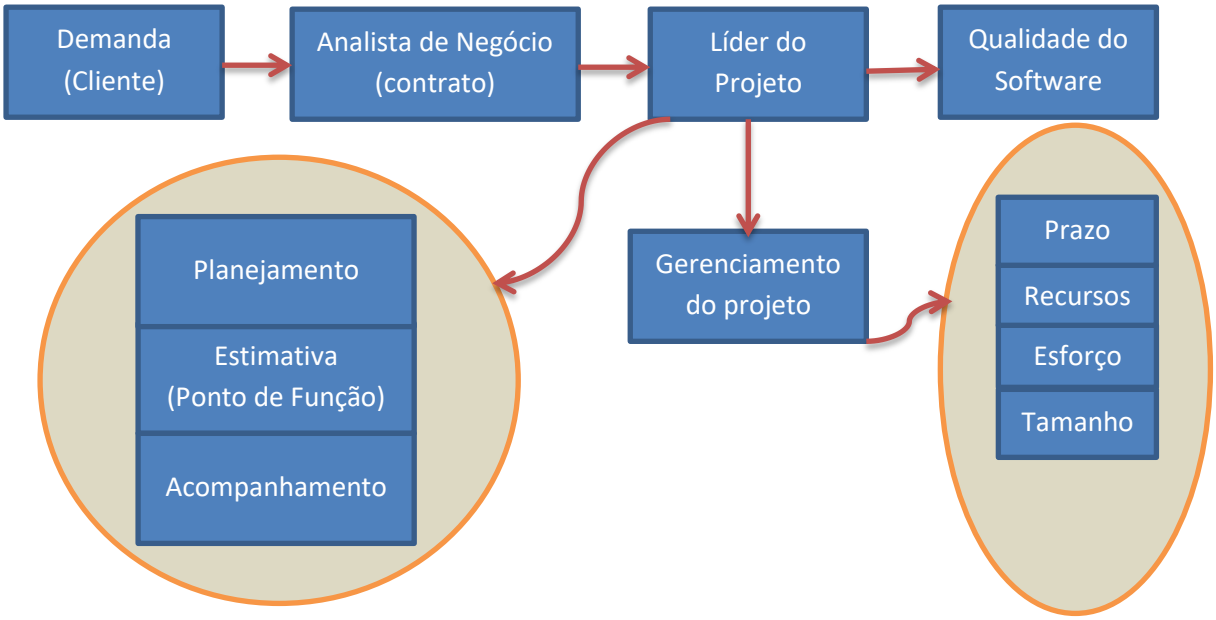
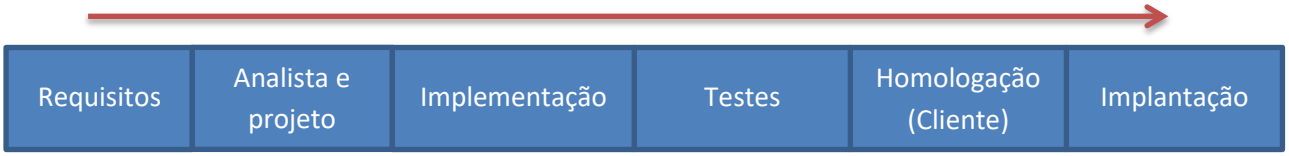


Gráfico sobre as etapas de um projeto:



I) Requisitos:

- Elicitação (Pegar informações com o cliente necessárias para o projeto).
- Documentação (Sem ambiguidades).
- Verificação (O que é possível ou não implementar).
- Validação

II) Análise e projeto:

- Projeto Lógico.
- Projeto Físico.

III) Implementação:

- Programas.
- Teste unitário (Do próprio implementador).

IV) Testes:

- Teste integrado, com todos os casos de erros "forçados", devendo resultar em um programa sem erros.

V) Homologação:

- Sugestão: Adicionamento de mais especificações **->** Retrabalho
- Erro: Especificação do cliente que a equipe não cumpriu.

5– <u>ESPECIFICAÇÃO DOS REQUISITOS</u>

I) Definição de Requisito:

- O QUE tem que ser feito;
- NÃO É COMO deve ser feito;

II) Escopo de Requisito:

- Requisitos mais genéricos (Sistema Autosau, por exemplo);
- Requisitos mais específicos (Funções);

III) Fases de Especificação:

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

    - Técnicas de Elicitação
        - ➔ Entrevista;
        - ➔ Brainstorm;
        - ➔ Questionário;
    - Documentação: Requisitos descritor em itens diretos
        - Uso da língua natural: Cuidado com ambiguidade;
        - Dividir requisitos em seus diversos tipos

          OBS.: Tipos de requisitos
            - Funcionais: O que deve ser feito em relação a informatização das regras de negócio
            - Não-funcionais: Propriedades que a aplicação deve ter e que não estão diretamente relacionadas com as regras de negócio

              Exemplo: - Segurança (Login e senha);
              - Tempo de processamento;
              - Disponibilidade (Ex.: 24h x 7 dias);
            - Requisitos Inversos: É o que não deve ser feito.

- Verificação: A equipe técnica verifica se o que está descrito na documentação é viável de ser desenvolvido.
- Validação: Cliente valida a documentação.

IV) Exemplo de Requisitos:

I) Bem formulados:

- "A tela de resposta da consulta de aluno apresenta nome e matrícula"
- "Todas as consultas devem retornar respostas no máximo em 2 segundos"

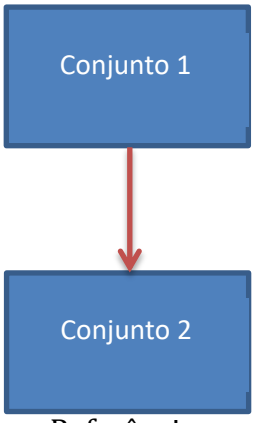
II) Mal formulados:

- "O sistema é de fácil utilização"
- "A consulta deverá retornar uma resposta em um tempo reduzido"

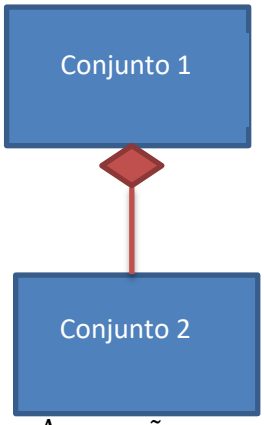
6– ***MODELAGEM DE DADOS***

I) Modelo corresponde n exemplos:

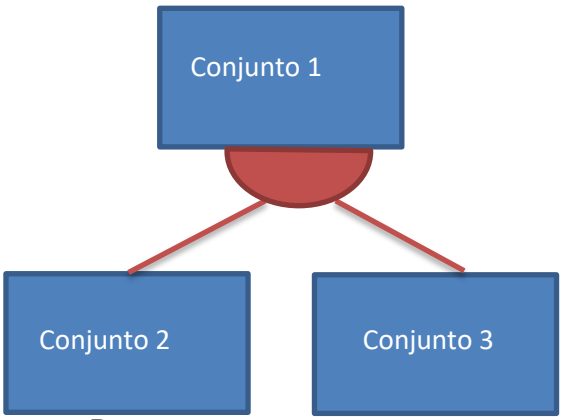
Notação:



Referência



Agregação



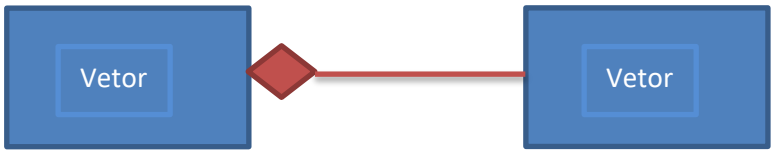
Particionamento

Estrutura Dinâmica
(Lista)

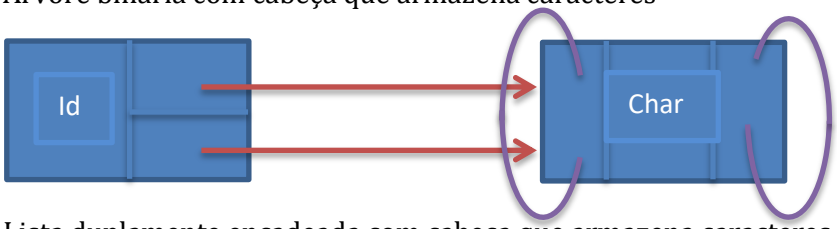
Estrutura Estática
(Vetor)

Exemplos:

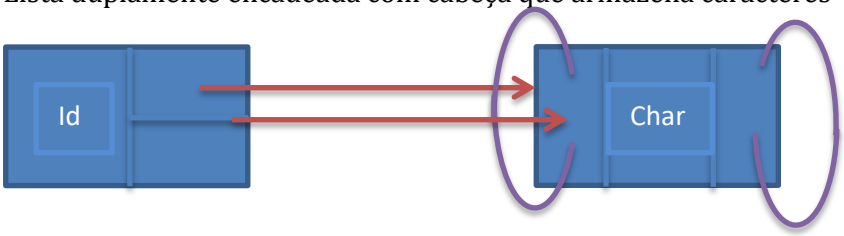
i) Vetor de 5 posições que armazenam inteiros



ii) Árvore binária com cabeça que armazena caracteres



iii) Lista duplamente encadeada com cabeça que armazena caracteres

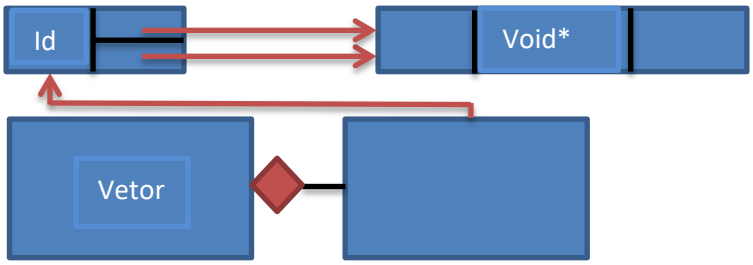


Assertivas Estruturais: Regras usadas para "desempatar" dois modelos iguais. Estas regras complementam o modelo, definindo características que o desenho não consegue representar.

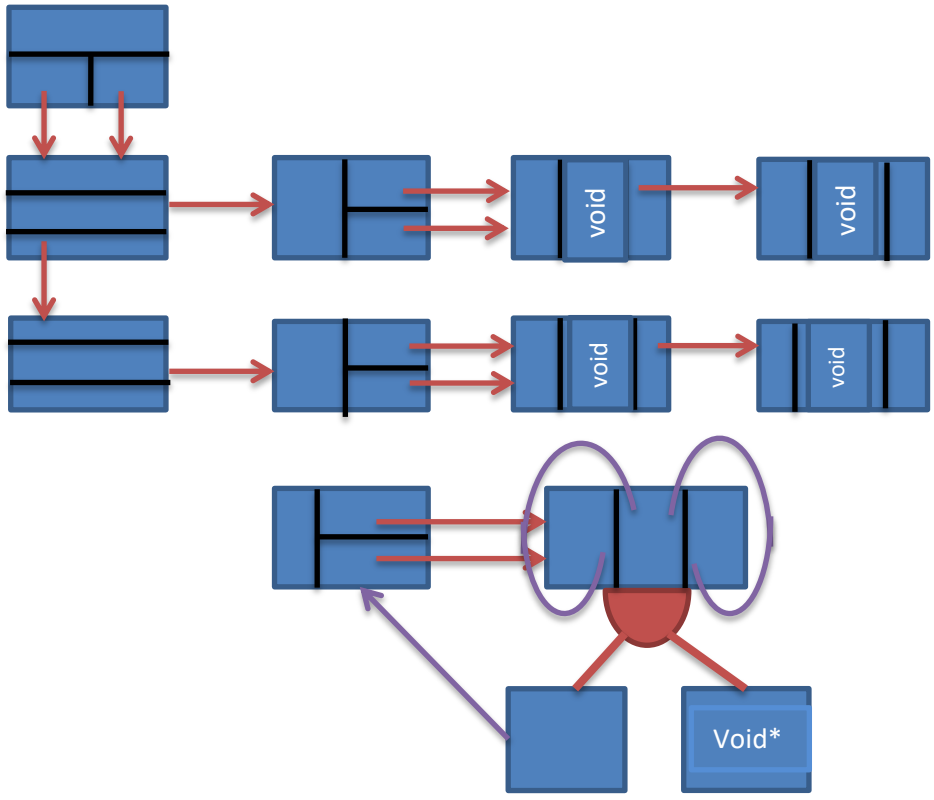
Lista: Se pCorr -> pAnt != NULL então,
pCorr -> pAnt -> pProx = pCorr
Se pCorr -> pProx != NULL então,
pCorr -> pProx -> pAnt = pCorr

Árvore: - Um poneteiro de uma subárvore a esquerda nunca referencia um nó
- pAnt e pProx de um nó nunca aponta para o pai

iv) Vetor de listas duplamente encadeadas com cabeça e genérica

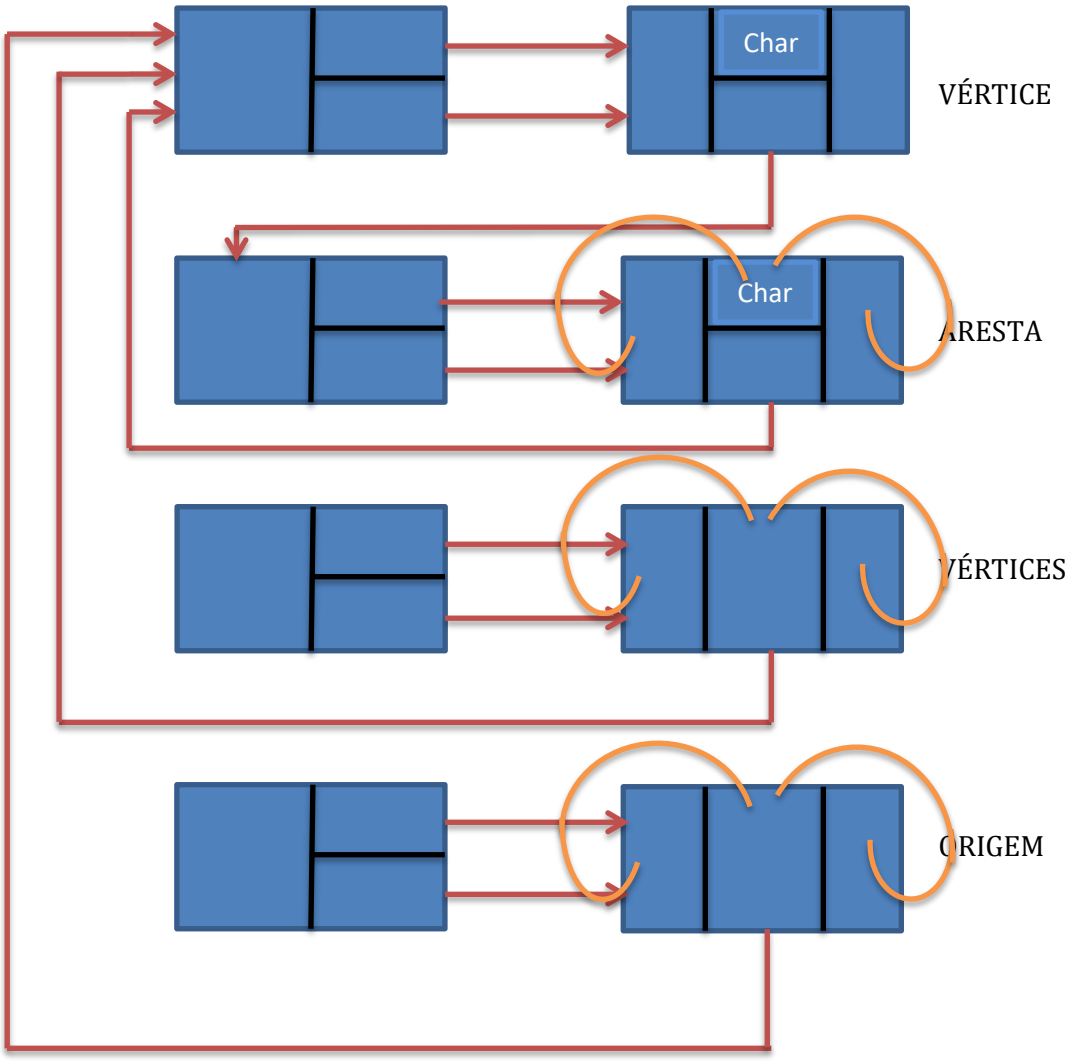


v) Matriz tridimensional genérica construída com listas duplamente encadeadas com cabeça



Assertivas Estruturais: A matriz comporta apenas 3 dimensões, sendo o nó da lista mais interna preenchido com um void *.

vi) Grafo criado com lista:

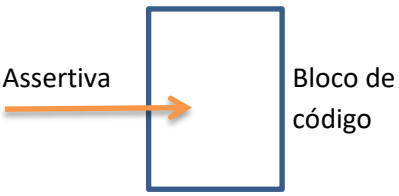


7 – ASSERTIVAS

I) Definição:

- Qualidade de construção: Qualidade aplicada a cada etapa do desenvolvimento de uma aplicação

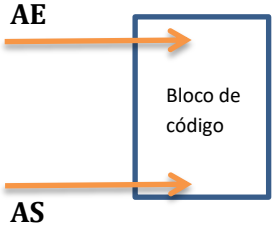
- Assertivas são regras consideradas válidas em determinado ponto do código



II) Aplicação de assertivas:

- Argumentação de corretude
- Instrumentação = Transforma as assertivas em bloco de código
- Trechos complexos em que é grande o risco de erros

III)Assertivas de entrada e saída:



OBS.: Há assertivas de tratas de regras envolvendo dados e ações tomadas

Cada função terá suas assertivas de entrada e saída

IV) Exemplos:

- Excluir nó corrente intermediário de uma lista duplamente encadeada

AE.: - Ponteiro corrente referencia o nó a ser excluído e este é intermediário
- a lista existe

AS.: - Nó referenciado inicialmente por corrente foi excluído
- Nó corrente aponta para o anterior

8 – Implementação da Programação Modular

I) Espaço de Dados

- São áreas de armazenamento, as quais são alocadas em um meio, possuindo um tamanho delimitado e um ou mais nomes de referência.

EX.: A[i] → O i-ésimo elemento do vetor A;
ptAux* → Espaço de dados apontado por ptAux;
ptAux → Espaço de dados que contém um endereço;
obterElemenTab(int id)->Id → É o subcampo Id presente na estrutura apontada pelo retorno da função

II) Tipos de Dados:

- Determinam a organização, codificação, tamanho em bytes e conjunto de valores permitidos.

OBS.: Um espaço de dados precisa estar associado a um tipo para que possa ser interpretado pelo programa desenvolvido em linguagem tipada.

III) Tipos de Tipo de Dados:
- Tipos computacionais:
    -- Int, Char, Char*
- Tipos Básicos (personalizados):
    -- Enum, Typedef, Union, Struct
- Tipos Abstratos de Dados:

OBS.: Imposição de Tipos - Forçar diferentes interpretações para o mesmo espaço de dados.
Ex.: Typecast (int *)malloc(...);

OBS.: Conversão de tipos – Não é igual a imposição de tipos, como transformar 1 em "1"

IV) Tipos Básicos:
- Typedef → "Sinônimo"
- Enum → enum{ texto1, texto2, texto3} tpExemplo;
    Enumeração, tpExemplo só pode ser texto1, 2 ou 3 ou 0,1,2.
- Struct → Junção (organização) de vários espaços
- Union →Várias interpretações para o mesmo espaço

V) Declaração e Definição de Elementos
- Definição
    -- Alocar espaço;
    -- Amarrar espaço a um nome (binding);
- Declaração
    -- Associar espaço a um tipo;

OBS.: - Quando o tipo é computacional ocorrem simultaneamente a declaração e definição;
- Malloc só define;
- Typedef struct só declara;

VI) Implementação em C e C++:
- Declarações e definições de nomes globais exportadas pelo servidor
    EX.: int a, int F(int b);
- Declarações externas contidas no modulo cliente e que somente declaram o nome sem associá-lo a um espaço de dados
    EX.: extern int a, extern int F(int b);

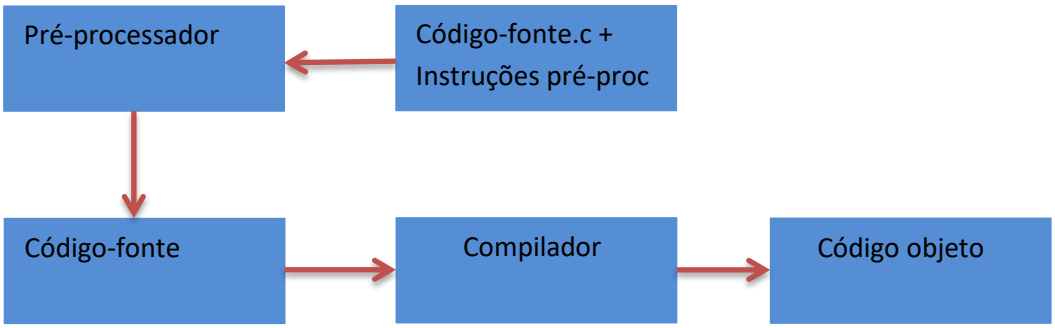
CLIENTE:	SERVIDOR:	CLIENTE:
Declaração	Declaração e Definição	Declaração
Extern int a;	int a;	Extern int a;

- Declaração e definição de nomes globais encapsuladas no módulo
    EX.: static int a, static int F(int b);

VII) Resolução de Nomes Externos
Por exemplo: Extern int a;

A resolução:
- Associa os declarados aos declarados e definidos
- Ajuste endereços para os espaços de dados definidos

VIII) Pré-processamento



#define nome valor
-> Substitui nome por valor
#undef nome
#if defined (nome) ou #ifdef nome
Texto verdade
#else
Texto falso
#endif
#if !defined (nome) ou #ifndef nome
#endif

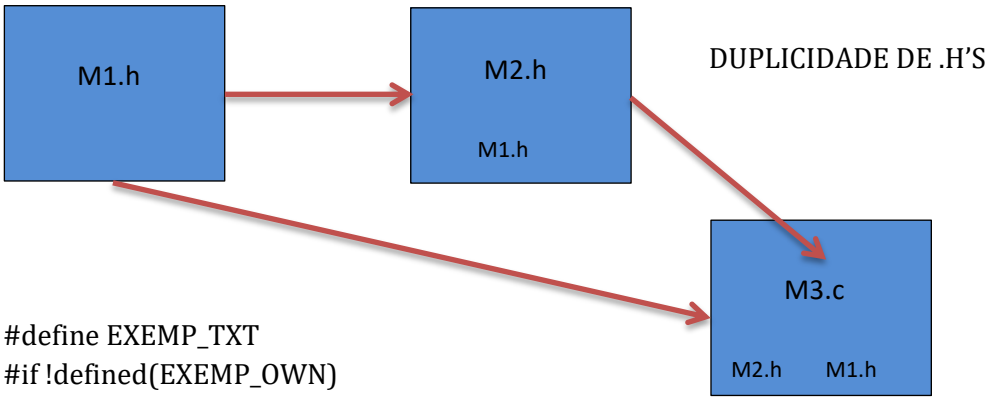
#include <arquivo> -> Inclui o conteúdo texto da biblioteca

Exemplo:

```
#if !defined (Exemp.mod)
#define Exemp.mod

        ...Texto do .h ....

#endif
```




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

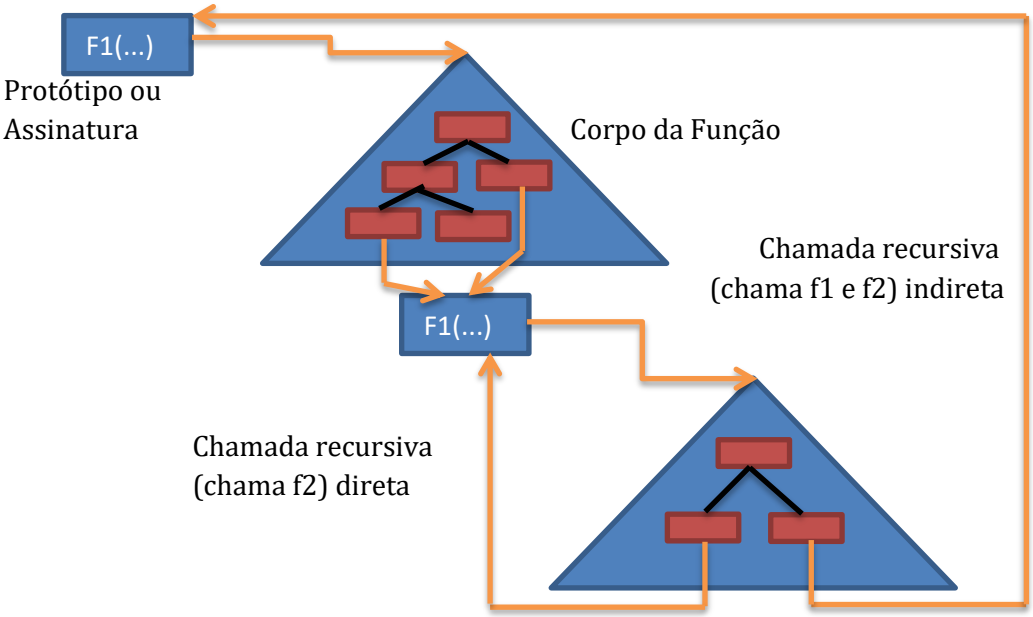
M1.c	M2.c	int vet[7] =	extern int vet[7];
#define EXEMP_OWN #include 'M1.h' #undef EXEMP_OWN	#include 'M1.h'	{1,2,3,4,5,6,7};	

9 – Estrutura de Funções

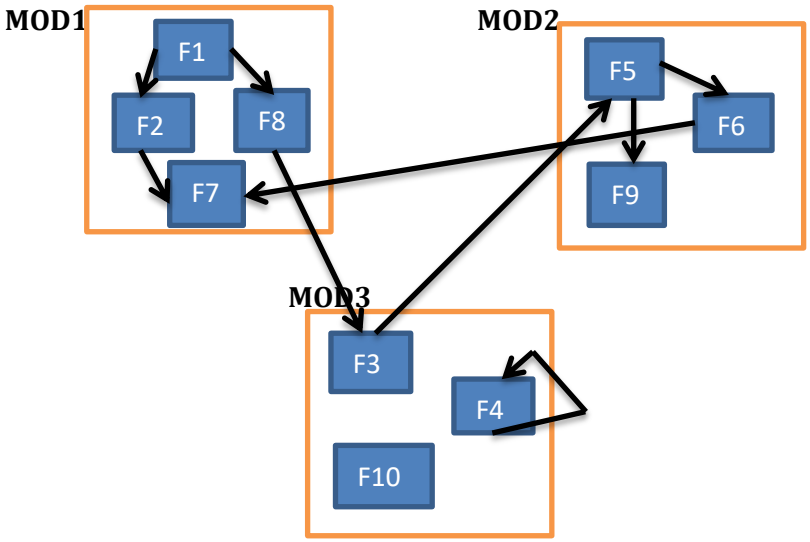
I) Paradigma:

- Forma de programar
    - -- Procedural: Programação estruturada;
    - -- Orientada a Objetos:
        - ->Programação orientada a objetos;
        - ->Programação modular (mesmo utilizando uma linguagem não orientada a objetos);

II) Estrutura de Funções:



III) Estrutura de Chamada:



- Função F4 -> F4: Chamada recursiva direta;
- F9 -> F8 -> F3 -> F5 -> F9: Chamada recursiva indireta;
- F10: Função morta (Não foi utilizada na estrutura de chamada);
- F8 -> F3 -> F5 -> F6 -> F7: Dependência circular entre módulos;
- F6 -> F7: Arco de chamada;

IV) Função:

- É uma porção auto-contida de código. Possui um nome, uma assinatura, um ou mais (ponteiro de função) corpos de código.

V) Especificação da Função:

- Objetivo (Podendo ser igual ao nome);
- Acoplamento (Parâmetros e condições de retorno);
- Condições de acoplamento (assertivas de entrada e saída;
- Interface como usuário (mensagens, saídas na tela para o usuário);
- Requisitos (O que deverá ser feito);
- Hipótese: São regras pré-definidas que assumem como válida uma determinada ação ocorrendo fora do escopo, evitando, assim, o desenvolvimento para uma terminada solução;
- Restrições: São regras que limitam as escolhas das alternativas de desenvolvimento para uma determinada solução;

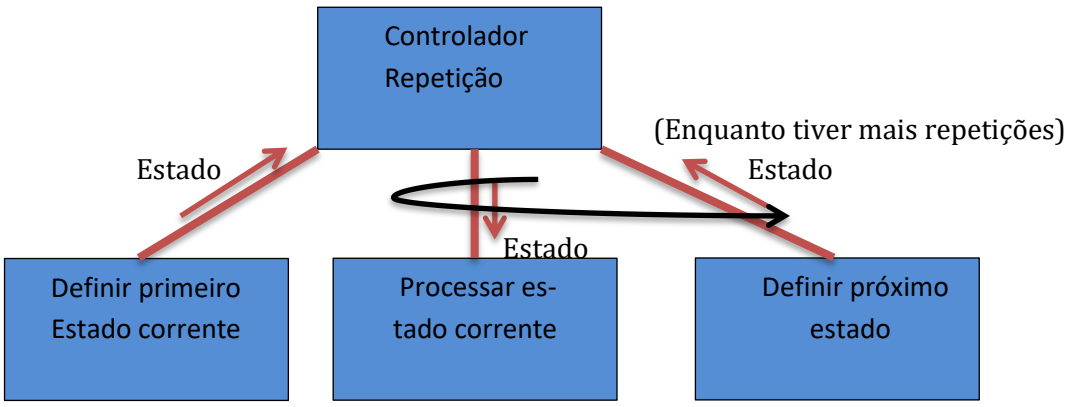
VI) Interfaces:

- Definições:
  i) Conceitual: Definição da interface da função sem a preocupação com a implementação
    -- inserirSimbolo(Tabela, Simbolo) -> Tabela, Id Símbolo, condRet
  ii) Físico: Implementação do conceitual
    -- tpCondRet insSimb(tpSimb *símbolo)
    -- tabela: global static no módulo
  iii) Implícito:
    -- Dados de interface diferentes de parâmetros e valores de retorno

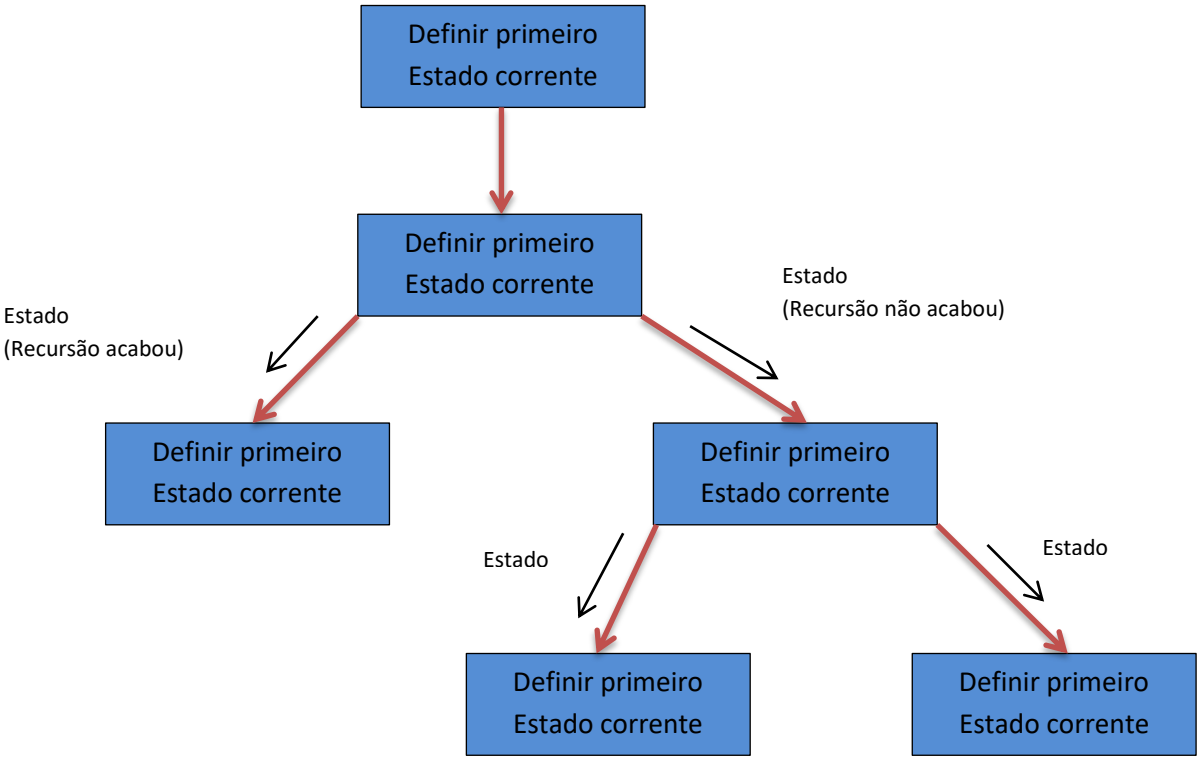
VII) Housekeeping:

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

VIII) Repetição:



IX) Recursão:



X) Estado:

- Descritor de estado: Conjunto de dados que definem um estado

Exemplo: índice na pesquisa vetor

O estado é a valoração do descritor.

OBS.: Não é necessariamente observável

Exemplo: Cursor de posicionamento de arquivo.

OBS2.: Não precisa ser único

Exemplo.: limSup e limInf de uma pesquisa binária

XI) 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 == Maior) sup = meio -1;
        else{
                inf = meio + 1;
        }
}
```

HOTSPOTS (PARTE ESPECÍFICA)

Esquemas de algoritmo permitem encapsular a estrutura de dado utilizada. É correto, independe de estrutura e é incompleto (precisa ser instanciado). Normalmente ocorrem em:

- Programação orientada a objetos;

- Frameworks;

Se o esquema for correto, hotspots com assertivas válidas, então, programa correto!!

XII) Parâmetros do tipo ponteiro para funções:

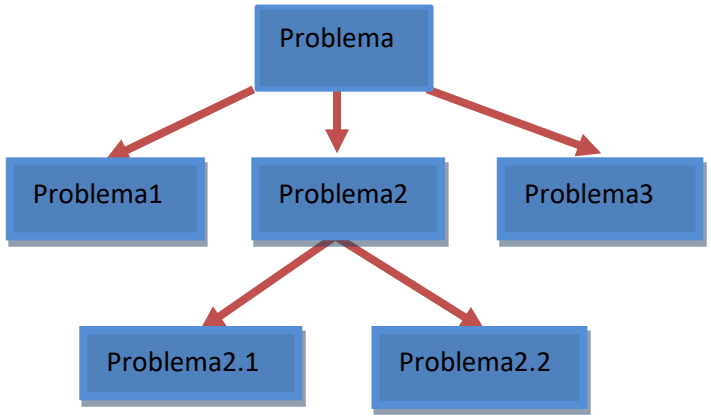
```
float areaQuad(float base, float altura){
        return base*altura;
}
float areaTri(float base, float altura){
        return base*altura/2.0
}
int ProcessaArea(float valor1, float valor2, float (*Func)(float,float)){
        printf("%f\n", Func(valor1, valor2));
}
condRet = ProcessaArea(5,2,areaQuad);
condRet = ProcessaArea(3,3,areaTri);
```

8 – Decomposição Sucessiva

I) Conceito:

- Divisão e conquista: Dividir um problema em subproblemas menores de forma que seja possível resolvê-los.
- Decomposição sucessiva é um método de divisão e conquista.

II) Estruturas de Decomposição:



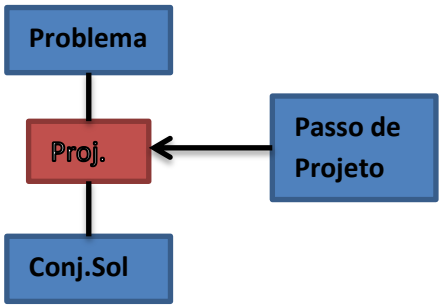
Prob1+ Prob2.1+Prob2.2+Prob3 →Componentes Concretos;
Problema+ Prob2 → Componentes Abstratos;
Prob2.1+Prob2.2+Prob2.3 →Conjunto Solução que é o conjunto de problemas, os quais quando são resolvidos, resolvem o problema raiz.

OBS.: Uma estrutura resulta somente em um conjunto solução. No entanto, para uma solução pode-se precisar diversas estruturas, sendo elas boas ou ruins.

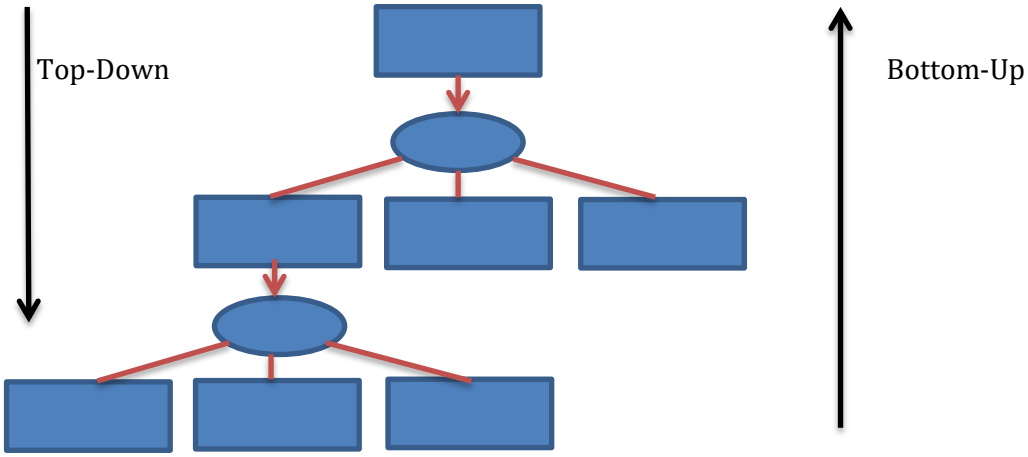
III) Critérios de Qualidade:

- Complexidade;
- Necessidade → Todos os componentes de um conjunto solução são necessários?
- Suficiência → Os componentes que fazem parte do conjunto solução são suficientes?
- Ortogonalidade → Q que um componente faz nenhum outro faz dentro de um conjunto solução

IV) Passo de Projeto:



V) Direção de Projeto:



9 – Argumentação de Corretude

```
INICIO
        IND <- 1
        ENQUANTO IND <= LL FAÇA
                SE ELEM[IND] = PESQUISADO
                        BREAK;
                FIM SE
                IND <- IND +1
        FIM ENQUANTO
        SE IND <= LL
                MSG "ACHOU"
        SENÃO
                MSG "NÃO ACHOU"
        FIM-SE
FIM
```

I) Definição:

- Método utilizado para argumentar que um bloco de código está correto

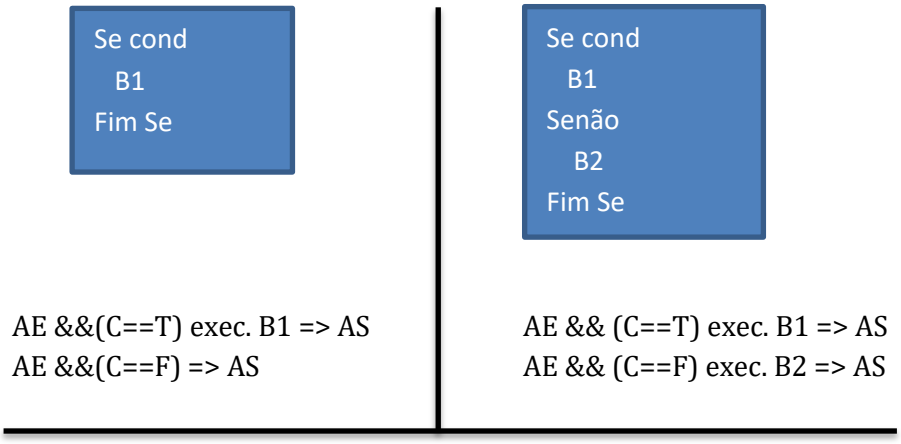
II) Tipos de argumentação:

- Sequencia;
- Seleção : SE
- Repetição: LOOP

III) Argumentação de Sequencia:

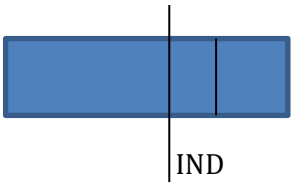
- AEprime: Existe um vetor válido e um elemento a ser pesquisado
- ASprime: Mensagem "ACHOU" se o elemento foi encontrado ou MSG "NÃO ACHOU" se IND>LL
- AI1: O IND aponta para a 1ª posição do vetor
- AI2: Se o elemento foi encontrado, IND aponta para o mesmo. Senão, IND>LL

IV) Argumentação de Seleção:



Seleção:
AE = AI2 & AS = ASprime
- AE &&(C==T) exec. B1 => AS
-- Pela AE, se o elemento for encontrado, IND aponta para o mesmo (e é <= LL). Como (C==T) IND<=LL, B1 apresenta MSG "ACHOU" valendo AS.
- AE && (C==F) exec. B2 => AS
-- Pela AE, IND > LL se o elemento pesquisado não foi encontrado. Como (C==F) IND > LL, neste caso, B2 é executado apresentando MSG "NÃO ACHOU", valendo a AS.

V) Argumentação de Repetição:
AE: AI1 & AS: AI2
Ainv = Assertiva invariante
- Envolve os dados descritores de estado
- Válida a cada ciclo da repetição



- Existem dois conjuntos: a ser pesq e já pesq
- IND aponta para elemento do conjunto a pesq

1 – AE => Ainv

- Pela AE, IND aponta para 1ª elemento do vetor e todos estão em a pesq. O conjunto já pesq está vazio, vale Ainv.

IND
A ser pesq

2 – AE && (C==F)=>AS (Não entra nem completa o ciclo)
- Não entra: Pela AE, IND=1. Como (C==F), LL < 1, ou seja, LL=0 (vetor vazio). Neste caso, vale a AS, pois o elemento pesquisado não foi encontrado.
- Não completa o ciclo: Pela AE, IND aponta para 1º elemento do vetor. Se este for igual ao pesquisado, o BREAK é executado e IND aponta para o elemento encontrado, vale AS.

3 – AE && (C==T) exec. B => Ainv (Primeiro ciclo)
- Pela AE, IND aponta para o 1º elemento do vetor. Como (C==T), este 1º elemento é diferente do pesquisado. Este, então, pass do conjunto a ser pesq para já pesq e IND é reposicionado para outro elemento de a ser pesq vale Ainv.

4 – Ainv && (C==T) exec. B =>> Ainv

- Para que a Ainv continue valendo, B deve garantir que um elemento passe de a ser pesq para já pesq e IND seja reposicionado.

5 – Ainv && (C==F) => AS

- Condição falsa: Pela Ainv, IND ultrapassou o limite lógico(LL) e todos os elementos estão em ja pesq. PESQUISADO não foi encontrado com IND > LL, vale AS.
- Ciclo não completa: Pela Ainv, IND aponta para elemento de a ser pesq que é igual a PESQUISADO. Neste caso, vale a AS pois ELEM[IND] = PESQUISADO

6 – Término:

- Como a cada ciclo, B garante que um elemento de a ser pesq passe para já pesq e o conjunto a ser pesq possui um número finito de elementos, a repetição termina em um número finito de passos.

```
ENQUANTO IND <= LL FAÇA
        SE ELEM[IND] == PESQUISADO
                BREAK
                FIM SE
AI3 - - - - - - - - - - - - - - - - - - >
                IND <- IND + 1
FIM ENQUANTO
```

Argumentação dentro do ENQUANTO:

Sequencia:

AE = Ainv && AS = Ainv;

AI3: O elemento pesquisado não é igual a ELEM[IND] ou o elemento foi encontrado em IND.

Seleção:

Seleção do primeiro bloco SE:

AE = Ainv && AS: AI3

1 – AE && (C == T) exec. B => AS

- Pela AE, IND aponta para um elemento do conjunto a ser pesq. Como (C == T), o ELEM[IND] é igual ao pesquisado e assim o elemento encontrado e IND, valor a AS

2 – AE && (C == F) => AS

- Pela AE, IND aponta para um elemento do a ser pesq. Como (C == F), o elemento apontado não é o pesquisado, valendo a AS.

10 – Instrumentação

I) Problemas ao Realizar Testes:

- Esforço de Diagnose:
  - -- Não saber resolver e ter que buscar a origem do erro;
  - -- Sujeito a muitos erros;
- Contribuem para este esforço:
  - -- Não estabelece com exatidão a causa a partir dos problemas observados
  - -- Tempo decorrido entre o instante da falha e o observado;
  - -- Falhas intermitentes;
  - -- Causa externa ao código que mostra a falha;
  - -- Ponteiros loucos (*wild pointers*);
  - -- Comportamento inesperado do hardware;

II) O que é instrumentação?

- Fragmentos inseridos nos módulos (Códigos & Dados);
- Não contribuem para o objetivo do programa;
- Monitora a serviço enquanto o mesmo é executado;
- Consome recursos de execução;
- Custa para ser desenvolvida.

III) Objetivo:

- Impedir que falhas se propaguem;
- Detecta falhas de funcionamento no programa o mais rápido possível de forma automática;
- Medir propriedades dinâmicas do programa (tempo de execução).

IV) Conceitos:

- Programa Robusto → Intercepta execução quando observa o problema.
  → Mantém o dano confinado.
- Programa tolerante a falhas →É robusto.
  →Possui mecanismos de recuperação.
- Deterioração Controlada →A habilidade de continuar operando corretamente mesmo com uma perda de funcionalidade.

V) Esquema de Inclusão de Instrumentos no Código C e C++

#ifdef_debug
- Código de instrumentação;
- Dados;

#endif
- Debug (Ativa);
- Release (Desativa);

VI) Assertivas Executáveis:

Assertivas em Código (PRECISAM SER COMPLETAS & CORRETAS)

- Vantagens:

-- Informam o problema quase imediatamente após ter sido gerado;
-- Controle de integridade feito pela máquina;
-- Reduz o risco de falha humana

VII) Depuradores:

- Ferramenta utilizada para executar um código passo-a-passo, permitindo que se distribua breakpoints com o objetivo de confinar os erros a serem pesquisados.
OBS.: Deve ser usado apenas como último recurso.

VIII) Trace:

- Instrumento utilizado para apresentar uma mensagem no momento que é apresentada

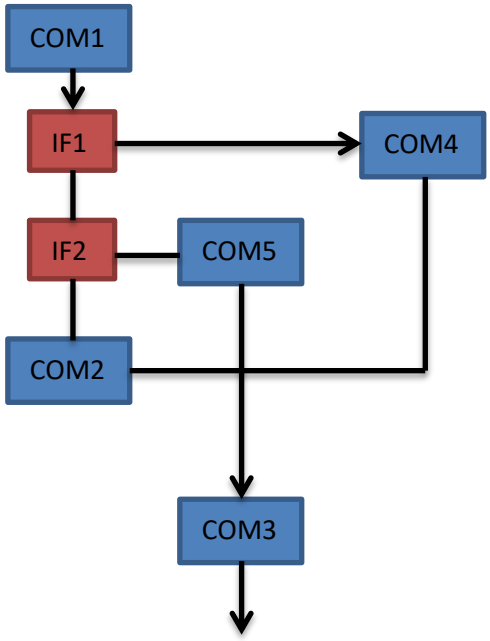
→Trace de Inclusão

Printf;

→Trace de evolução

Apresenta mensagem quando o conteúdo de uma variável for alterada;

IX) Controlador de Abertura:




```
INICIO
        COM1
        IF1 COND
                IF2 COND
                        CNT_CONTAR(CONT4)
                        COM2
                ELSE
                        CNT_CONTAR(CONT3)
                        COM5
        ELSE
                CNT_CONTAR(CONT2)
                COM4
FIM
```

- Instrumento composto de um vetor de contadores que tem como objetivo acompanhar os testes caixa aberta de uma aplicação, monitorando todos os caminhos percorridos.

X) Verificador Estrutural:

- Assertiva (E, S, I) → Código → Assertiva Executável.
- Assertivas Estruturais → Código = Verificador.

+ Modelo

- Verificador: Instrumento responsável por realizar uma verificação completa de estrutura em questão. É a implementação de código relacionado com as assertivas estruturais e modelo.

XI) Deturpador Estrutural:

- Instrumento responsável por inserir erros na estrutura com o objetivo de testar o verificador → Deturpa (tipoDet).

OBS.: A deturpação é sempre realizada no nó controle.

XII) Recuperação Estrutural:

- Instrumento responsável por recuperar automaticamente uma estrutura a partir de um erro observado em uma aplicação tolerante a falhas.

XIII) Estrutura Auto Verificável:

- É a estrutura que contém todos os dados necessários para que seja totalmente verificada.

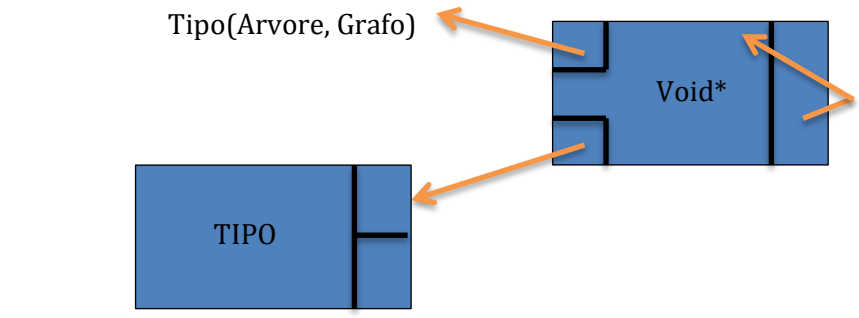
Exemplo: Lista Simplesmente Encadeada.

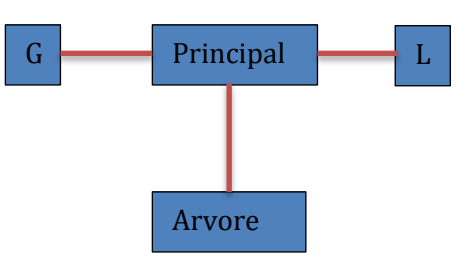
Q1) É possível definir todos os tipos de dados que a estrutura está apontando?

R.: Inclusão de um tipo em cada nó da estrutura e no cabeça.

Q2) É possível acessar qualquer ponteiro da estrutura de qualquer origem?

R.: Inclusão de campo pNoAnterior e pCabeça.






```
INSERE (PT, VALOR
        #IFDEF_DEBUG, TIPO
        #ENDIF)
```