Aluno: Nagib Moura Suaid 1710839

Introdução:

Vantagens da programação modular:

-Vencer Barreiras de complexidade
-Facilita trabalho em grupo(Paralelismo)
-Reúso de código
-Facilita a criação de um acervo(coleção)
-Desenvolvimento incremental
-Aprimoramento individual
-Facilita a administração de baselines(versões estáveis)

Princípios de Modularidade:

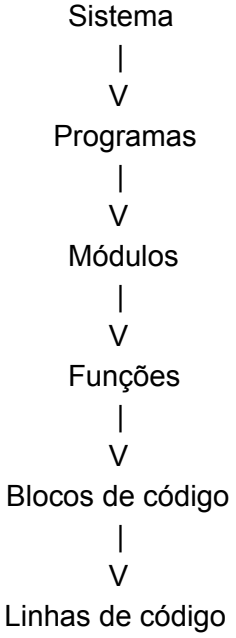
1- Módulo (caixa)

definição física: Unidade de compilação independente
definição lógica: Trata de um único conceito

2-Abstração de Sistema

Processo de considerar apenas o que é necessário em uma situação (definir o escopo)
(Reduzir entidades/objetos às suas características relevantes)

Níveis de abstração:



OBS: **Artefato**: é um item com identidade própria criado dentro de um processo de desenvolvimento. Pode ser versionado.

Construto (build): é um “resultado apresentável”.(necessariamente um artefato).

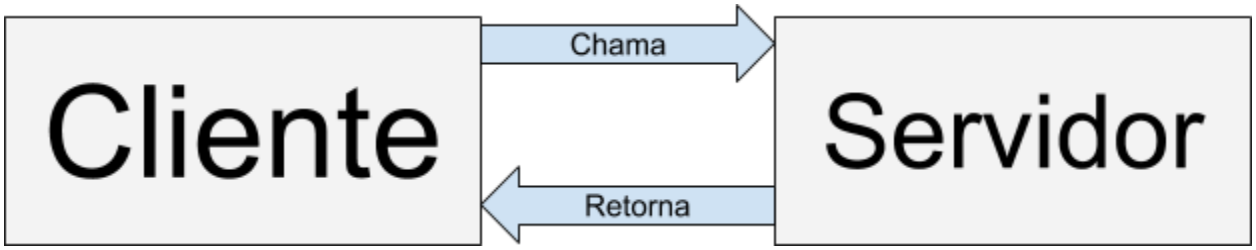
3-Interface

Mecanismo de troca de dados, estados e eventos entre elementos de um mesmo nível de abstração

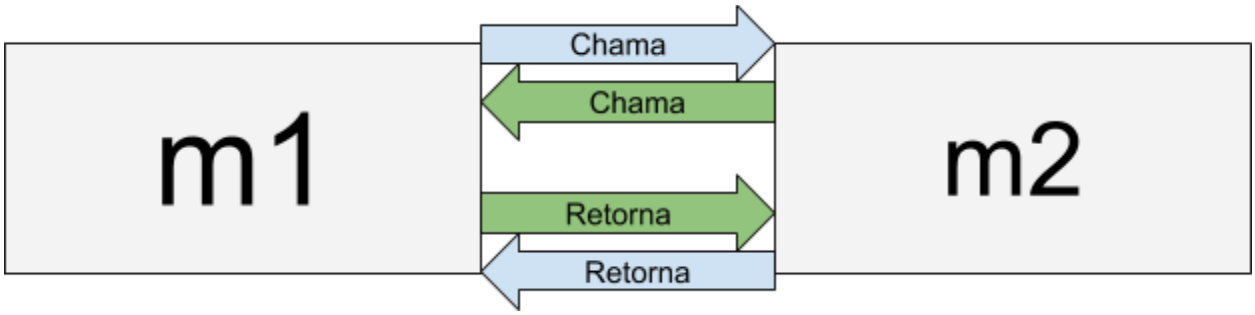
a)Exemplos:

-Entre Sistemas: Arquivos
-Entre Módulos: Funções de acesso
-Entre Funções: Parâmetros
-Entre Blocos: Variáveis globais

b)Relacionamento Cliente-Servidor

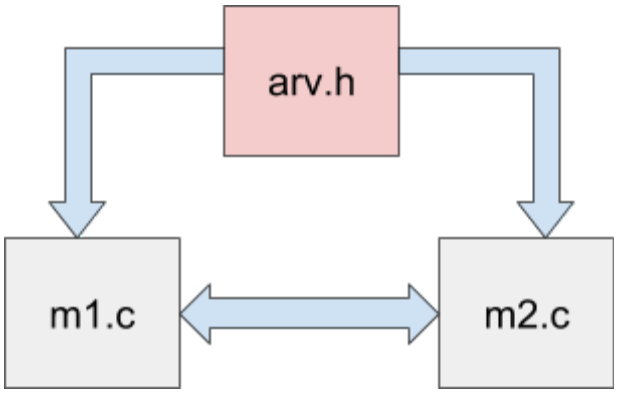


Callback: O servidor precisa de mais informação para realizar a sua tarefa, então a relação cliente-servidor inverte temporariamente:



c)Interface fornecida por terceiros

Módulos que utilizam a mesma estrutura precisam de uma interface fornecida por terceiros



d)Interface em detalhe

-Sintaxe(Regras)

MOD1 ←X→ MOD2
int float

-Semântica(Significado)

MOD1 ←X→ MOD2
int idade int cpf

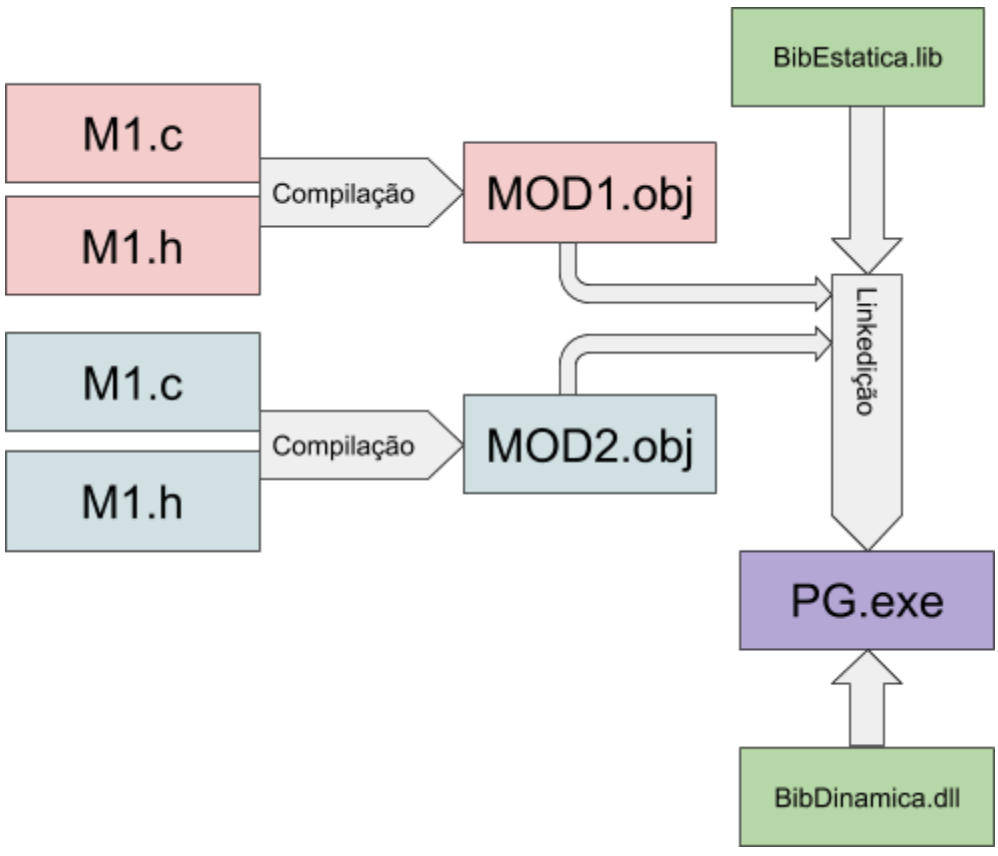
e)Análise de Interface

tpDadosAluno* ObterDadoAluno(int mat); (protótipo da função)

Cliente espera um ponteiro tpDadosAluno*
Servidor espera um inteiro int mat
Ambos esperam a estrutura tpDadosAluno

4-Processo de Desenvolvimento

OBS: .c→ Módulo de Implementação, .h→ Módulo de Definição
OBS2: TODO CLIENTE PRECISA DO .h DE SEU SERVIDOR



5-Bibliotecas Estáticas e Dinâmicas

Estática:

Vantagem:

.lib já é acoplado em tempo de linkedição à aplicação executável

Desvantagem:

Existe uma cópia da biblioteca estática na memória para cada executável que a utiliza

Dinâmica:

Vantagem:

Só é carregada uma instância em memória, independente do número de aplicações que a usam

Desvantagem:

.dll precisa estar na máquina para a aplicação funcionar (dependência externa)

6-Módulo de Definição

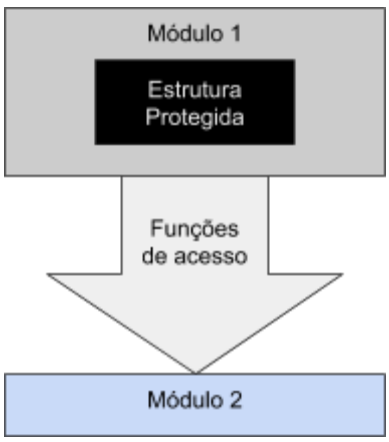
-interface do módulo
-contém os protótipos das funções de acesso, interfaces fornecidas por terceiros
-documentação voltada para o programador do módulo cliente

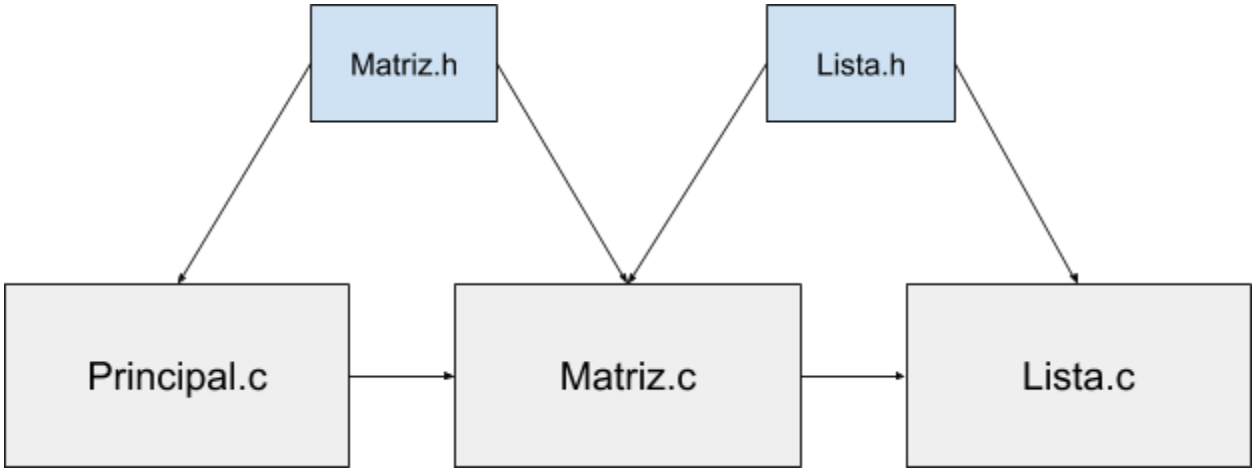
7-Módulo de Implementação

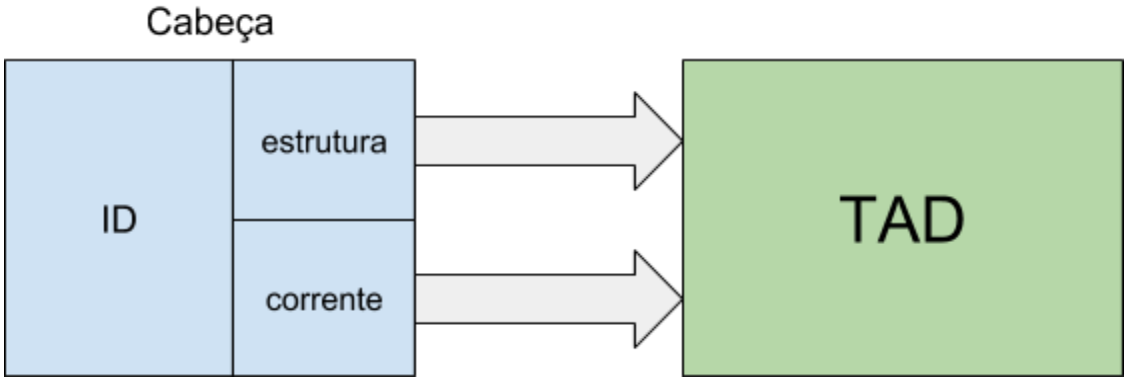
-código das funções de acesso
-protótipo e código das funções internas
-variáveis internas ao módulo
-documentação voltada para o programador do módulo servidor

8-Tipo Abstrato de Dados

É uma estrutura encapsulada em um módulo que somente é conhecida pelos módulos cliente através das funções de acesso disponibilizadas na interface.







Utilização e administração de um tad:
typedef struct ptcab* ptCab

criarLista(ptCab* plista) passagem de parâmetro por referência
inserirNo(ptCab, void *conteudo) passagem de parâmetro por valor

9-Encapsulamento

Propriedade relacionada com a proteção dos elementos que compõem um módulo
Facilita a manutenção e impede a utilização indevida

*encapsulamento de documentação:
- -Documentação interna → módulo de implementação
- -Documentação externa → módulo de definição
- -Documentação de uso → manual do usuário

*encapsulamento de código:
    -dentro de módulos
    -dentro de blocos de código
    -dentro de funções

*encapsulamento de variáveis :
    -private→ objeto
    -public→ mundo (orientado a objeto)
    -global→ mundo
    -static→ módulo/classe
    -protected→ estrutura de herança

10-Acoplamento

Propriedade relacionada com a interface entre módulos

conector==item da interface (ex.: função de acesso, variável global...)

Critérios de qualidade:
    -Quantidade de conectores→ necessidade X suficiência (preciso?, basta?)

    -Tamanho do conector→ quantidade de parâmetros

    -Complexidade de conector→ documentação e mnemônicos (nomes autoexplicativos)

11-Coesão

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

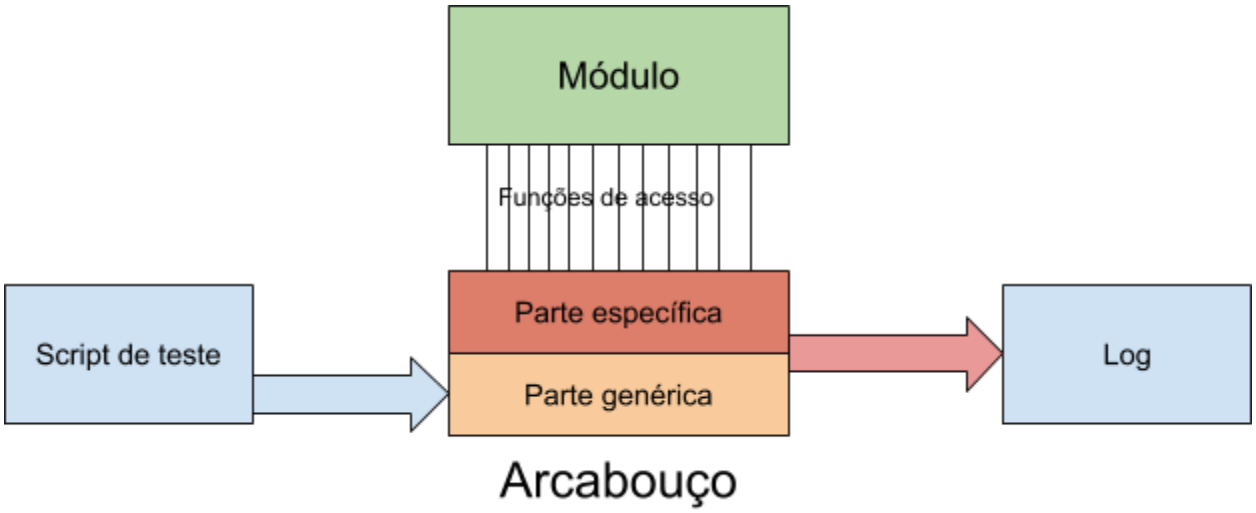
níveis de coesão:
    -incidental→ nada a ver
    -lógica→ relação lógica (ex.: processa)
    -temporal→ tarefas feitas no mesmo período de tempo
    -procedural→ tarefas feitas em sequência
    -funcional→ tarefas semelhantes
    -abstração de dados→ um conceito (o melhor)

Teste Automatizado

1) Objetivo:

Testar de forma automática um módulo, recebendo um conjunto de casos de teste (script) e gerando um log de saída com a análise entre o resultado obtido e o esperado.

2) Framework de teste



3) Script

== caso de teste → testa uma situação(contexto)(pode ter vários comandos)
=comando de teste→ chama uma única função de acesso
=recuperar→ ignora um erro esperado

Um script de teste completo deve testar todas as possíveis condições de retorno de cada uma das funções de acesso.

4)Log de saída

Arquivo de texto listando os casos de teste do script, onde houve algo inesperado e onde tudo ocorreu como previsto:

==caso x → OK
1>> Func esperava 0 e retornou 1 → ERRO

*Toda informação do Log após o primeiro erro não pode ser confiada.

5) Parte específica

Necessita ser implementada para que o framework possa acoplar no app: Hotspot

Ex: TESTEARV.c:

/* Tabela dos nomes dos comandos de teste específicos */

```
#define      CRIAR_ARV_CMD       "=criar"
#define      INS_DIR_CMD         "=insdir"
#define      INS_ESQ_CMD         "=insesq"
#define      IR_PAI_CMD          "=irpai"
#define      IR_ESQ_CMD          "=iresq"
#define      IR_DIR_CMD          "=irdir"
#define      OBTER_VAL_CMD       "=obter"
#define      DESTROI_CMD         "=destruir"
...
 TST_tpCondRet TST_EfetuarComando( char * ComandoTeste )
   {

      ARV_tpCondRet CondRetObtido   = ARV_CondRetOK ;
      ARV_tpCondRet CondRetEsperada = ARV_CondRetFaltouMemoria ;
                                 /* inicializa para qualquer coisa */
      char ValorEsperado = '?'  ;
      char ValorObtido   = '!'  ;
      char ValorDado      = '\0' ;
      int  NumLidos = -1 ;
      TST_tpCondRet Ret ;
      /* Testar ARV Criar árvore */
      if ( strcmp( ComandoTeste , CRIAR_ARV_CMD ) == 0 )
      {

            NumLidos = LER_LerParametros( "i" ,
                                 &CondRetEsperada ) ;
            if ( NumLidos != 1 )
            {
            return TST_CondRetParm ;
            } /* if */

            CondRetObtido = ARV_CriarArvore( ) ;

            return TST_CompararInt( CondRetEsperada , CondRetObtido ,
                                 "Retorno errado ao criar árvore." );

      } /* fim ativa: Testar ARV Criar árvore */
```

Processo de Desenvolvimento em Engenharia de Software

Demanda→ Analista de negócios→ Líder de Projeto

A demanda vem de um cliente.
O Líder deve estimar:

- Tamanho do projeto (ponto de função)
- Esforço para terminar o projeto
- Recursos necessários
- Prazo de término do projeto

Etapas:

- Requisitos (o que o Cliente quer):
  - -Elicitação
  - -Documentação
  - -Verificação
  - -Validação
- Análise e Projeto
  - -Projeto Lógico (Modelagem de dados, ...)
  - -Projeto Físico (Módulos, arquivos, BD, ...)
- Implementação
  - -Programas
  - -Teste Unitário
- Teste Iterado
  - -Teste integrado
- Homologação (Apresentação ao Cliente)
  - -Sugestão-> Retrabalho renumerado
  - -Erro-> Se ferrou, se vira
- Implantação

OBS:

- *Gerência de configuração
- *Gestor de qualidade de software

Especificação de requisito

Definição de requisito:

-O **QUE** DEVE SER FEITO

Escopo de requisito (Abstração):

-requisito genérico

-requisito específico

Fases de especificação

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

-entrevista

-brainstorm

-questionário

-Documentação -> Organização da “ata caótica” em requisitos simples:

-Linguagem natural e sem ambiguidade

-Divisão em tipos:

-funcionais: informatização das regras de negócio

-não funcionais: propriedades do app que não estão diretamente relacionadas com as regras de negócio (Segurança)

-inversos: coisas a não serem feitas (Proteção do analista)

-Verificação -> Verificar se o que está descrito é viável de ser desenvolvido

-Validação -> Cliente valida a documentação

Exemplos de requisito

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.

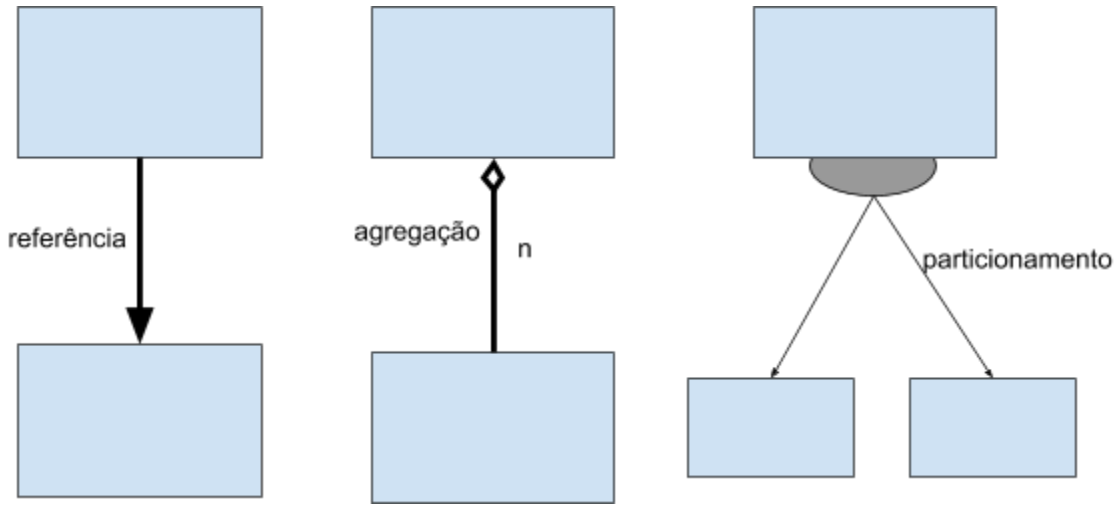
Mal formulados:

-O SIsTEma é dE FácIl UTilIZaçÃo

-A TelA MOStrA seuS dADOs mAiS IMPorTanTes

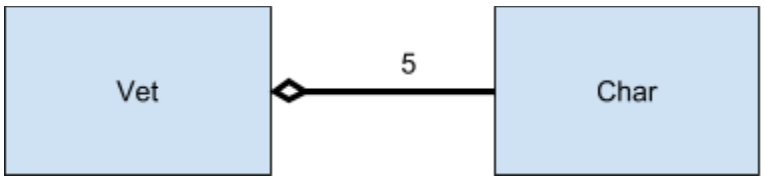
Modelagem de dados

(Modelo deve representar todo e qualquer exemplo possível)

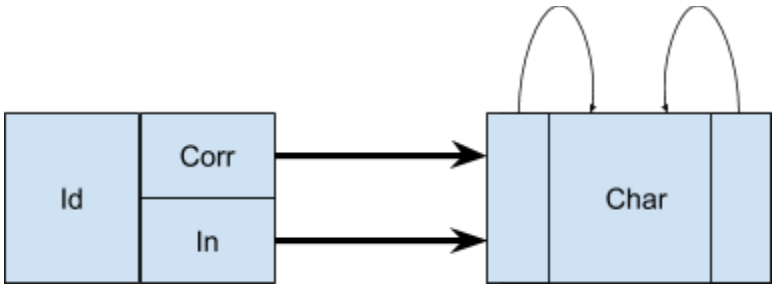


Exemplos:

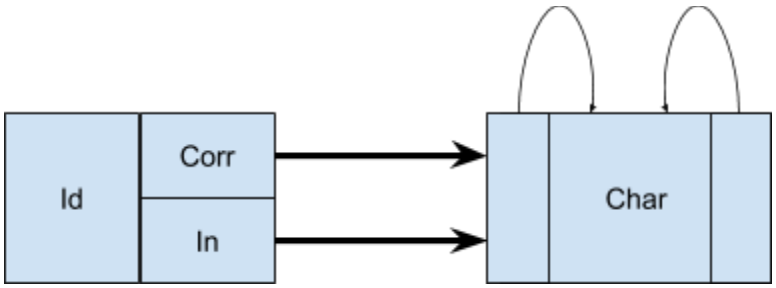
Vetor de 5 posições que armazena char



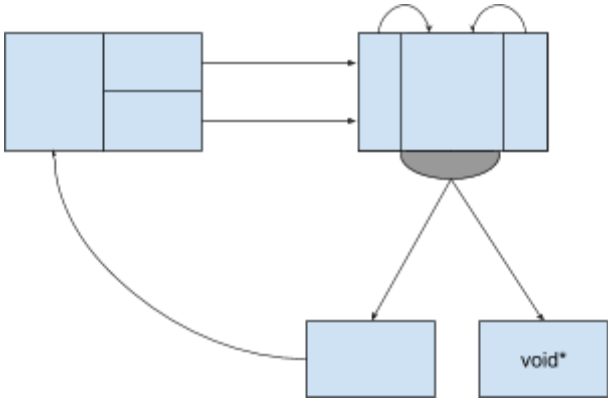
Árvore binária com cabeça que armazena char



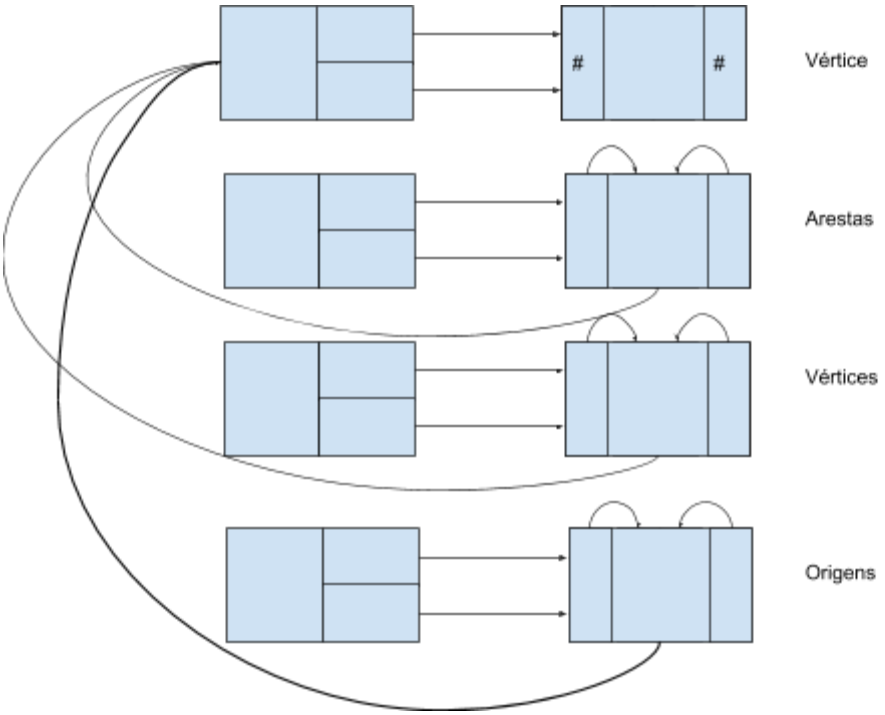
Lista duplamente encadeada de char com cabeça



Matriz 3D genérica construída com listas duplamente encadeadas com cabeça



Grafo criado com listas



OBS: Assertivas estruturais são utilizadas para “desempatar” modelos que são representados da mesma forma (se existir um contraexemplo o modelo não é válido)

Ex:

Lista:

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

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

Assertivas

São regras consideradas válidas em determinado ponto do código

Onde aplicar?:

-Argumentação de corretude

-Instrumentação

-Trechos complexos

Assertivas de entrada e saída:

São assertivas que tratam de como os dados devem estar ao começo do código e que ações foram tomadas (o que mudou)

Exemplo

-Excluir nó corrente de uma lista

AE: ponteiro corrente referencía um nó intermediário

AS: nó foi excluído e ponteiro corrente aponta para outro nó

Implementação de Programação Modular

Espaço de dados:

são áreas de armazenamento:

-alocadas em um meio

-possui tamanho

-possui nomes de referência

A[j] -> j-ésimo elemento de A
ptAUX* -> elemento apontado por ptAUX
ptAUX -> endereço
(*obterElemtab (int id)).Id -> subcampo Id do retorno da função

Tipos de Dados

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

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

Tipos de Tipos de Dados

-computacionais (int, char)

-basicos (enum, typedef, struct, union)

-abstrato de dado (TAD)

CONVERSÃO DE TIPO != IMPOSIÇÃO DE TIPO

(imposição -> typecast)

Tipos Basicos

-typedef -> novo nome

-enum -> enum{ “””” varios typedefs “””

a

b

}

-struct -> varios campos separados

-union -> um campo com varias interpretações

Declarações e definições

Definir:

-alocar espaço e amarrar a um nome (binding)

Declarar:

-associa espaço a um tipo

obs: Quando o tipo é computacional, definição e declaração são simultâneas

Implementação em C e C++

-Declarações e definições de nomes globais exportador pelo módulo servido:

int a

int F(int b)

-Declarações externas contidas no módulo cliente que somente declara sem associá-lo a um espaço de dados

extern int a

extern int F(int b)

-Declarações e definições de nomes globais encapsulados no módulo

static int a
static int F(int b)

Resolução de nomes externos

Um nome externo somente declarado em um determinado módulo necessariamente deve ser definido em um outro módulo

resolução:

-associa declarados a declarados e definidos
ajusta endereços para os espaços definidos

Pré-processamento

#->pré-processamento

#define -> substitui texto

#undef -> desfaz o define

#if defined \
#else ----->divide o código
#endif /

#if !defined -> divide o código

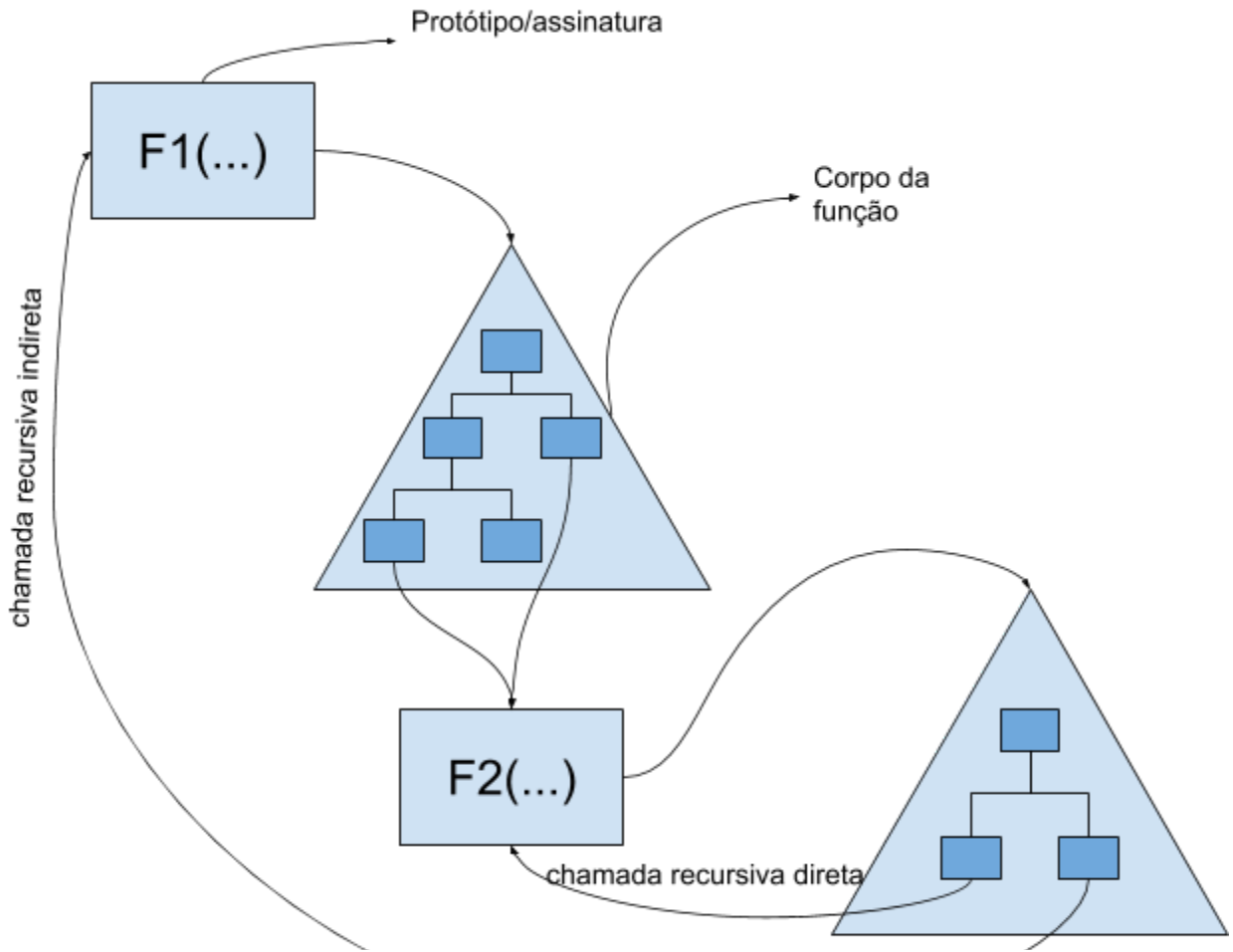
#include<arq>/"arq" -> inclui texto ("copia e cola")

```
#if !defined (EXEMP_MOD)
#define EXEMP_MOD
.
.
.
#endif
```

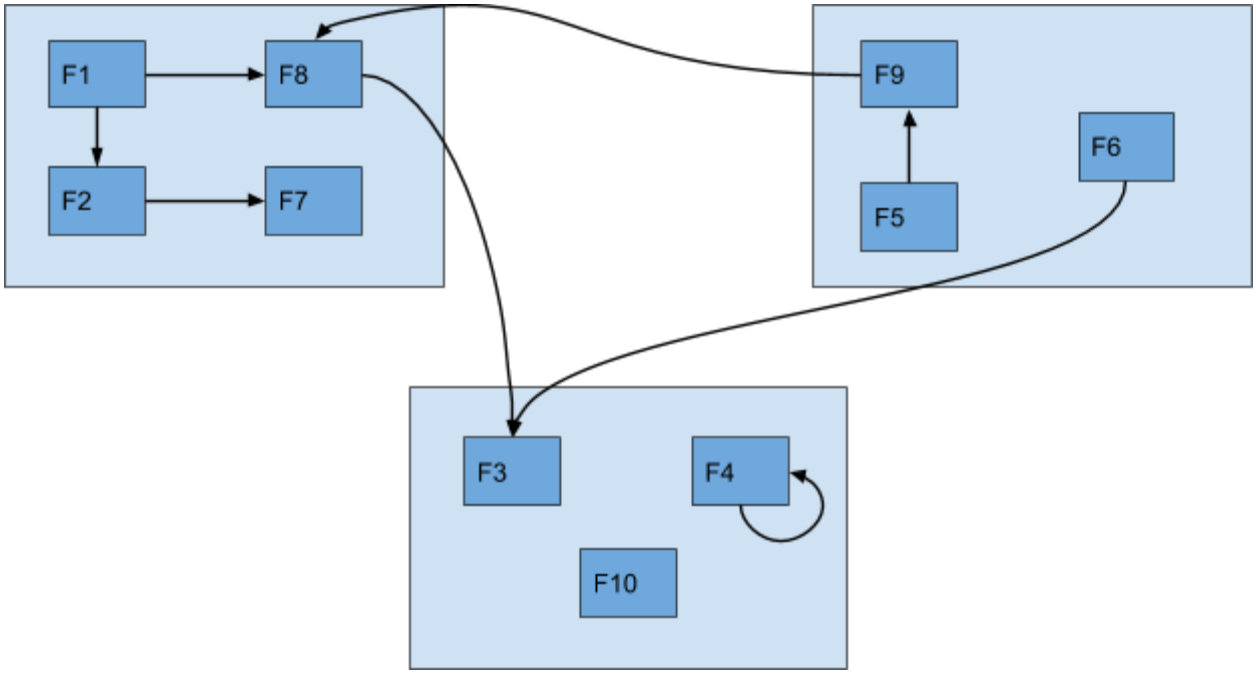
```
#if defined (EXEMP_OWN)
#define EXEMP_EXT
#else
#define EXEMP_EXT extern
#endif
EXEMP_EXT Int vet[5]
#if defined EXEMP_OWN
     ={1,2,3,4,5};
#else
     ;
#endif
```

Estrutura de Funções

1. Paradigma
   -Forma de programar:
   -Procedural
   -Orientada a Objeto (Inclui Programação Modular!)
2. Estrutura de Funções



3. Estrutura de chamadas



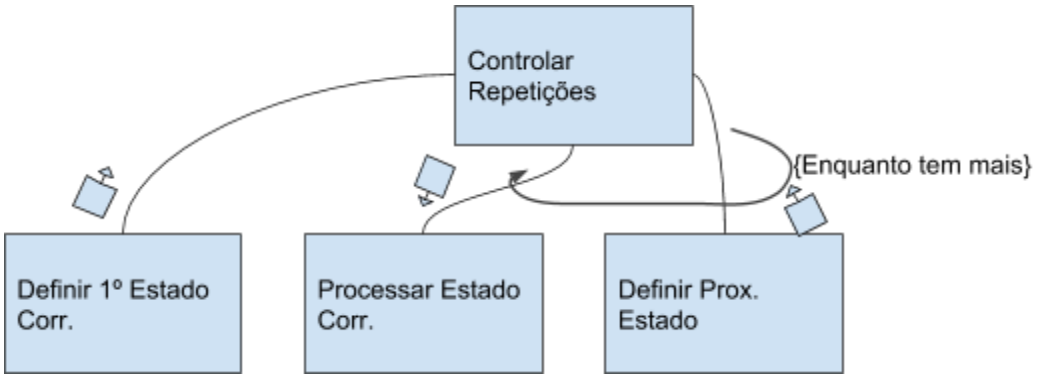
f4 -> f4 - Chamada Recursiva Direta
f9->f8->f3->f5->f9 - Chamada Recursiva Indireta
f10 - Função Morta
f8->f3->f5->f6->f7 - Dependência Circular Entre Modulos

4. Funções
   São porções auto contidas de código. Possuem:
   -Nome
   -Assinatura
   -Corpo(s)
5. Especificação de Função
   -Objetivo
   -Acoplamento:parâmetros e condições de retorno
   -Condições de acoplamento: AE e AS
   -Interface com o usuário
   -Requisitos
   -Restrições (alternativas de desenvolvimento)
   -Hipótese (Regras que podem ser assumidas como válidas)
6. Interfaces
   conceitual: definição da interface da função sem preocupação com a implementação
   inserirSimbolo(Tab, Simb);
   Tabela, IdSimb, condRet
   física: implementação do conceitual
   tpCondRet insSimb(tpSimp* simbolo)
   tabela-> global static
   implicita: dados da interface diferentes de parâmetros e valores de retorno

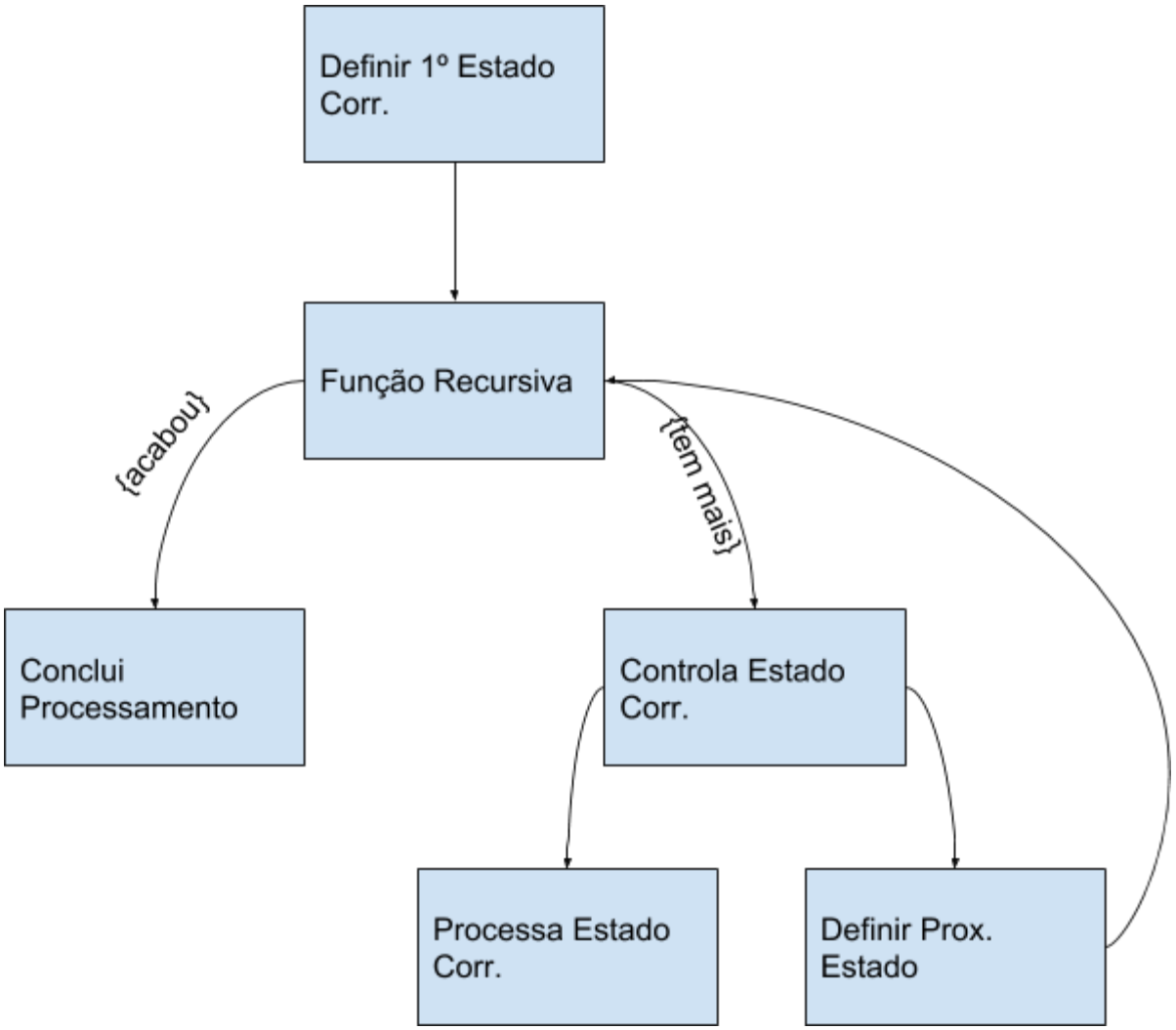
7. Housekeeping

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

8. Repetição



9. Recursão----------



10. Estado
    -Descritor de estado:
        Conjunto de dados que definem um estado
    -Estado:
        Uma valoração do descritor
    OBS: Não precisa ser observável nem único

11. Esquema de Algoritmo

```
inf = ObterLimInf();
sup = ObterLimSup();
while (inf <= sup)
{
      meio = (inf+sup)/2;
      comp = compara(valorProc , obterValor(meio));
      if (comp == igual ) { break; }
      if (comp == menor ) { sup = meio - 1; }
      else (inf = meio + 1;)
}/* while */
```

Tudo => Framework

“Formato”=> Parte genérica

Negrito => Hotspot=>Parte específica

*Permitem encapsular a estrutura de dados utilizada. É correto independente de estrutura e é incompleto (precisa ser instanciado).

Ocorre nomalmente em P.O.O. e frameworks.

Se Esquema é correto e Hotspot com assertivas válidas então o Programa está correto.

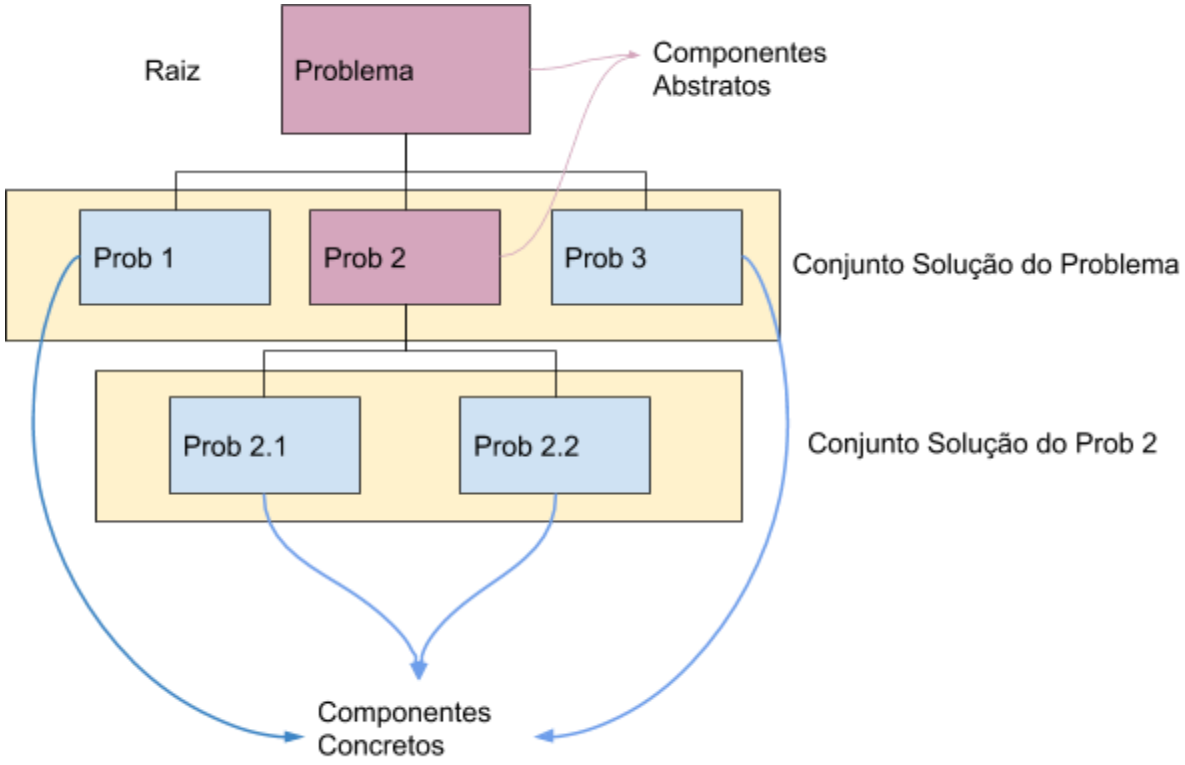
12. Parâmetros do tipo ponteiro para função
    Precisam ter o mesmo retorno e mesmos parâmetros!
    float AreaQuad (float base, float alt) {return base*alt;}
    float AreaTri (float base, float alt) {return base*alt/2;}

    int ProcessaArea(float v1, float v2 float (*FuncArea)(float,float))
    { … aux=FuncArea(v1,v2); …}

Decomposição sucessiva

1) Conceito
   -Divisão e conquista
   divisão de problemas em subproblemas “resolvíveis”
   -Decomposição sucessiva É UM método de divisão e conquista

2) Estrutura de Decomposição



   1 Estrutura --- 1 Solução
   1 Solução -- N Estruturas

3) Critério de qualidade
   -Complexidade
   -Necessidade
   -Suficiência
   -Ortogonalidade

4) Passo de Projeto

5) Direção de Projeto
   -Bottom up
   -Top down

Argumentação de Corretude:

-É um método utilizado para argumentar que um código está correto.

Tipos de argumentação:

-sequência
-seleção
-repetição

```
INÍCIO
     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
```

Argumentação de sequência:

-Divisão do bloco de código em sub blocos.

-Determinar as assertivas que separam os blocos (AE, AS AI's)

ex:

AE: Existe um vetor valido e um elemento a ser pesquisado.

AS: MSG “achou” se pesquisado existe no vetor e MSG “não achou” caso contrário.

AI1: IND aponta para a primeira posição do vetor.

AI2: Se o elemento foi encontrado, IND aponta para o mesmo. Caso contrário IND>LL

Argumentação de seleção:

-Dois tipos: SE e SE-SENÃO

SE:

AE && (c == T) + B => AS

AE && (c == F) => AS

SE-SENÃO:

AE && (c == T) + B1 => AS

AE && (c == F) + B2 => AS

ex:

AE==AI2, AS==AS

Pela AE, se o elemento foi encontrado IND aponta para o mesmo. Como a condição é verdadeira IND<LL. B1 (o bloco correspondente) apresenta MSG “achou”.

Pela AE, se o elemento não foi encontrado IND>LL. Como a condição é falsa IND>LL. Neste caso B2 é executado e é apresentado MSG “não achou”

Argumentação de repetição:

-Determinar a AINV(invariante):

-envolve descritores de estado

-válida todo ciclo

-DEVE-SE ARGUMENTAR:

-AE => AINV

-AE && (c == F)(condição falsa OU ciclo incompleto) + B => AS

-AE && (C == T) + B => AINV

-AINV && (C == T) + B => AINV

-AINV && (C == F) => AS

-Término da repetição

ex:

AINV: Existem dois conjuntos: a pesquisar e já pesquisado. ind sempre aponta para um elemento do conjunto a pesquisar.

AE => AINV: Pela AE, IND aponta para o primeiro elemento do vetor e todos estão em a pesquisar. O conjunto já pesquisado está vazio, vale AINV.

AE && (C == F) => AS:

-Condição Falsa: Não entra: Pela AE, ind = 1. Como (C = F), LL < 1, ou seja, LL= 0. Neste caso, vale a AS pois o elemento pesquisado não foi encontrado.

-Não completa o 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 o elemento encontrado, tornando AS válida.

AE && (C == T) + B => AINV: Pela AE, ind aponta para o 1º elemento do vetor. Com (C == T), este primeiro elemento é diferente do pesquisado. Este passa do conjunto a pesquisar para o já pesquisado e IND é reposicionado para outro elemento de “a pesquisar”, tornando válida AINV.

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

AINV && (C == F) + B => AS:

-Condição falsa: Pela AINV, ind ultrapassou limite lógico e todos os elementos estão em já pesquisado. Pesquisado não foi encontrado com ind > LL, tornando AS válida.

-Não completa o ciclo: Pela AINV, ind apontou para elemento de a pesquisar, que é igual a pesquisado. Neste caso, AS é válida pois Ele[ind] = pesquisado.

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

Instrumentação

-Problemas Ao Realizar Testes:

Esforço de Diagnose

-grande

-sujeito a erros

Contribuem para esse esforço:

-não estabelece com exatidão a causa dos problemas observados

-tempo entre falha e erro

-falhas intermitentes (nem sempre acontece)

-causa externa ao código que mostra a falha

Ex:

Ponteiros loucos

Comportamento inesperado do hardware

-O que é Instrumentação:

-fragmentos inseridos nos módulos:

código e dados

-**não** contribuem para o objetivo do programa

-monitora o serviço enquanto o mesmo é executado

-consome recursos de execução

-custa para ser desenvolvido

-Objetivo:

-detectar falhas de funcionamento do prog ASAP automaticamente

-impedir que falhas se propaguem

-medir propriedades dinâmicas do programa

-Conceitos:

-programa robusto:

intercepta a execução quando observa um problem e mantém o dano confinado.

-programa tolerante:

robusto e corrige a falha

-deterioração controlada:

habilidade de continuar operando corretamente mesmo com perda de funcionalidade

-Esquema de inclusão de instrumentação no código em C e C++:

```
#ifdef _DEBUG
    .
    .
    .
#endif
```

-Assertivas Executáveis

assertiva -> código

Vantagens:

-informam problema quase imediatamente após ter sido gerado.

-controle de integridade feito pela máquina

-reduz risco de falha humana

precisam ser completas e corretas.

-Depuradores

Ferramenta utilizada para executar um código passo a passo permitindo breakpoints com o objetivo de confinar os erros a serem identificados. **ÚLTIMO RECURSO.**

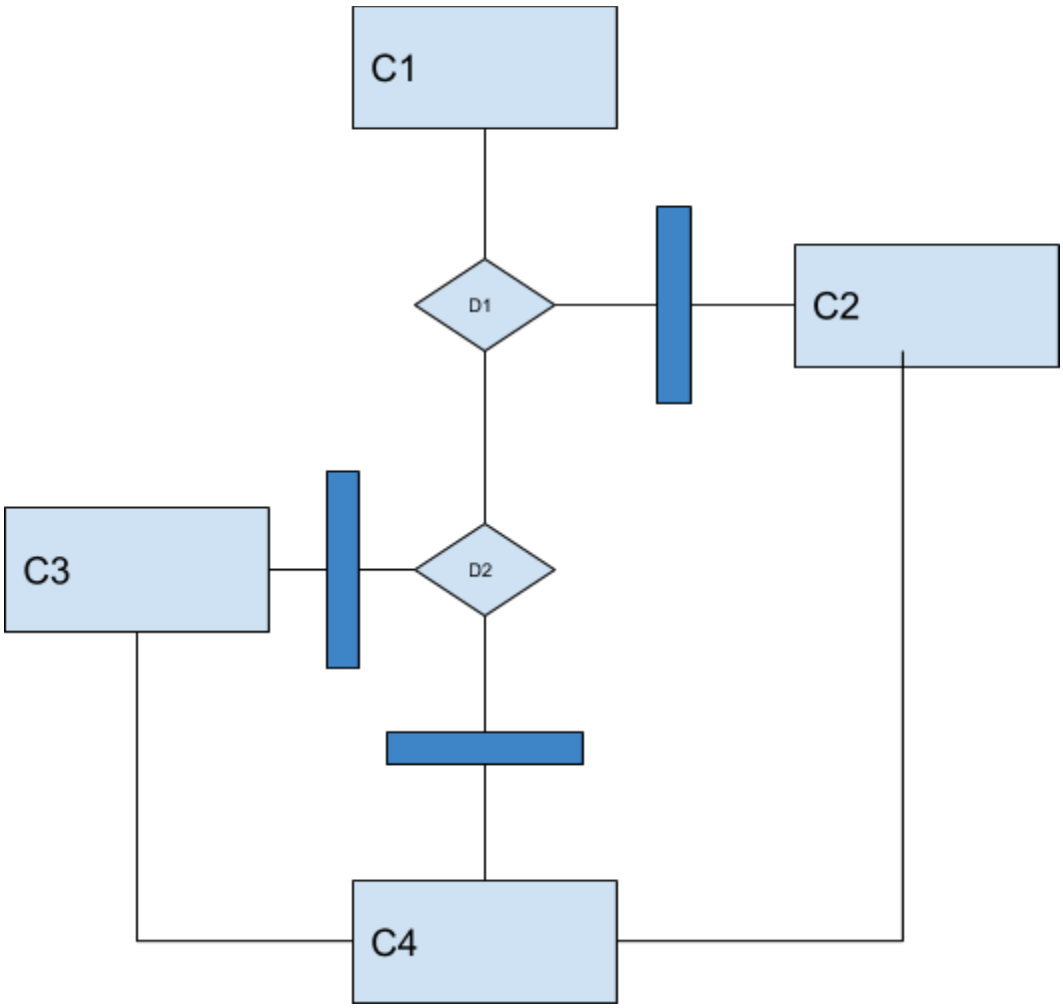
-Trace

Instrumento para apresentar uma mensagem no momento que é executado

Trace de intrução: printf

Trace de evolução: quando variáveis mudam

-Controlador de Cobertura



-Para testes caixa aberta

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

-Verificador Estrutural

(Assertivas Estruturais + Modelo) => Código

Instrumento responsável por realizar uma verificação completa da estrutura em questão.

É uma função dentro da estrutura

-Deturpador Estrutural

Instrumento responsável por inserir erros na estrutura com o objetivo de testar o verificador

Outra função dentro da estrutura:

deturpa(tipo);

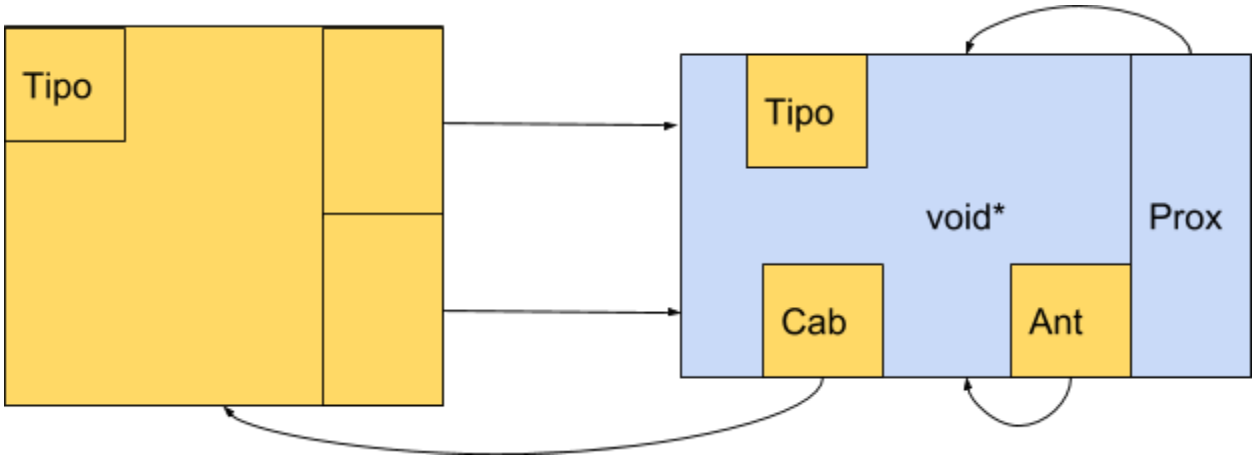
obs: Sempre realizada no nó corrente!

-Recuperador Estrutural

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

-Estrutura Autoverificável

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



-É possível definir todos os tipos apontados pela estrutura?

incluir campo tipo na estrutura E inclusão de cabeça com capo tipo

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

incluir campo pCabeca (e pAnt)