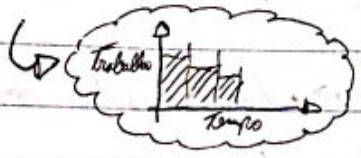
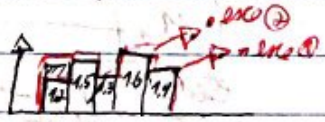
PM

Introdução

- vencer barreiras de complexidade
- distribuição de tarefas em grupo
- reuso – um módulo pode servir p/ multiplas aplicações



- permite trabalhar com baseline (versão) de módulos já testados



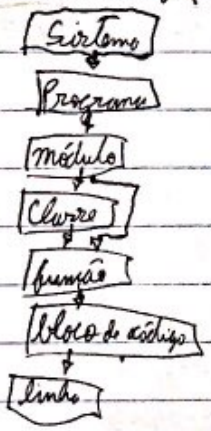
- desenvolvimento incremental
- aprimoramento particular
- reduz o tempo de compilação (não compila tudo sempre)

Princípios de Modularidade

1) Módulo

- física: Unidade de compilação independente
- lógica - possui um único conceito

2) Hierarquia



- Podem ser: blocos de códigos, fragmento de texto de documentação, funções, figuras ou diagramas, seções de documentação, tipos de dados, classes, componentes...
- Artefato: algo que elaborado durante um processo de desenvolvimento e que possui identidade própria.
- Construto: versão de aplicação que pode ser executada mesmo que incompleta

/ /

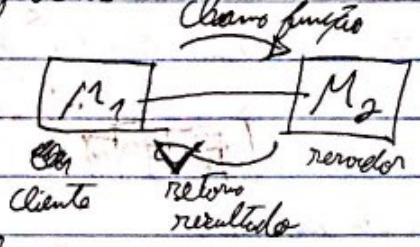
3) Interface

- mecanismo de troca de dados, comandos e eventos entre elementos de programas

A interface sempre ocorre entre elementos de programa do mesmo nível de hierarquia

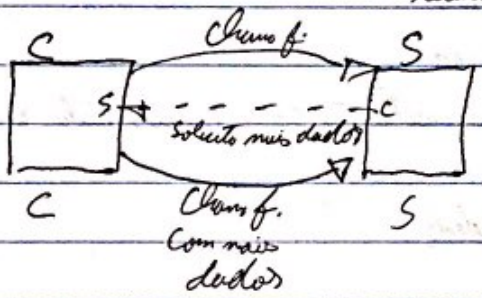
Tipos de interface: - arquivos entre sistemas, funções entre módulos entre blocos; variáveis globais.

Relacionamento cliente - servidor



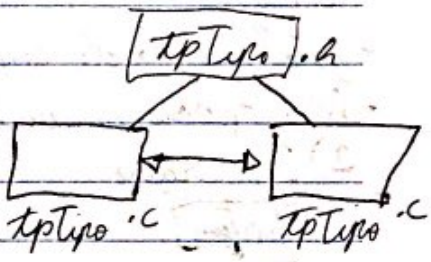
Casos especiais:

Callback



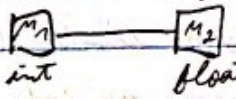
- Interface fornecido por terceiros:

É quando um tipo utilizado em uma interface entre dois módulos não está definido em nenhum dos módulos de implementação e sim num módulo de definição comum aos dois.

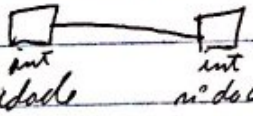


- Interface em detalhe

- Sintaxe: regras



- Semântica: significado



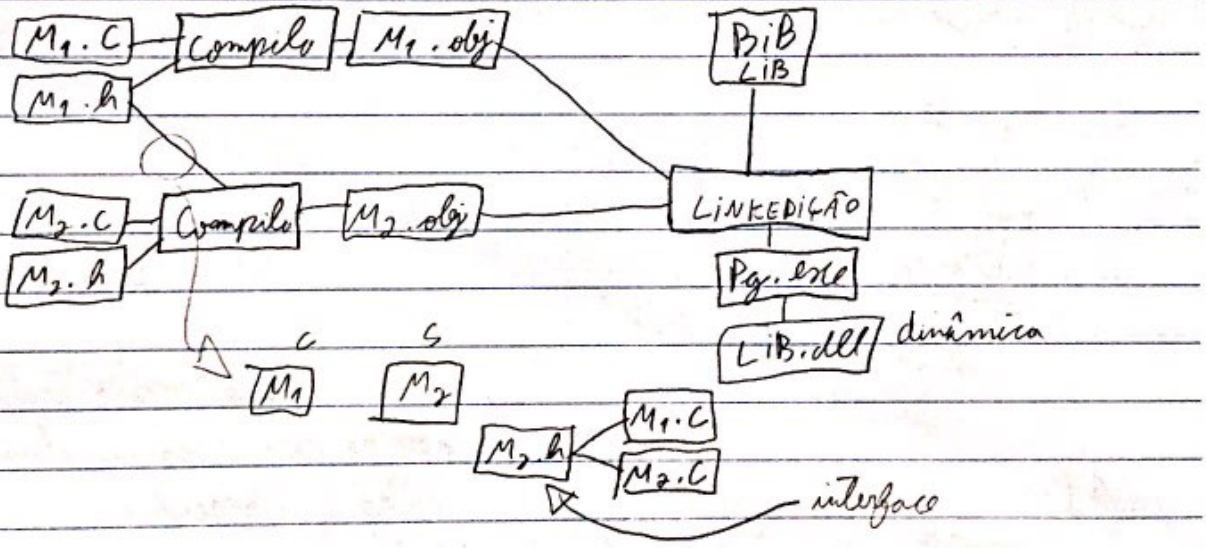
Exemplo de interface:

→ tpDadosAlunos * obterAlunos(int id)

- interface esperada pelo cliente
  - ↳ ponteiro válido referenciando os dados dos alunos (ou null)
- interface esperada pelo servidor
  - ↳ id válida
  - ↳ acesso aos dados do aluno se o mesmo existir
- interface esperada por ambos
  - ↳ tpDadosAlunos (interface fornecida por terceiros)

4) Processo de Desenvolvimento

Protocolo de uso: forma de se utilizar os itens que compõem uma interface para que ela possa ser operada corretamente



5) Módulo de Definição (.h)

- ↳ especificação externa voltada p/ os programadores do módulo cliente.
- ↳ Protótipo ou assinatura das funções de acesso
- ↳ Declarações e códigos públicos ao módulo

/ /

6) Módulo de Implementação (.c)

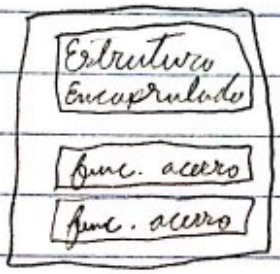
↳ Especificações externas voltadas p/ os programadores do módulo servidor

↳ Protótipo ou assinatura das funções internas

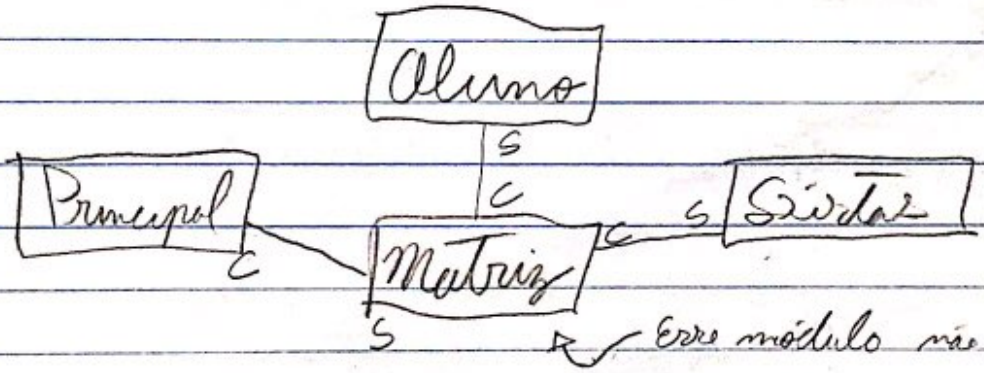
↳ Declarações e códigos encapsulados no módulo

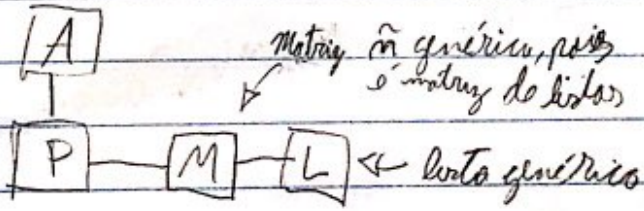
↳ Códigos executáveis das funções

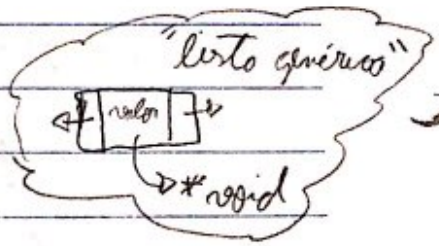
7) Tipo Abstrato de Dados (TAD)

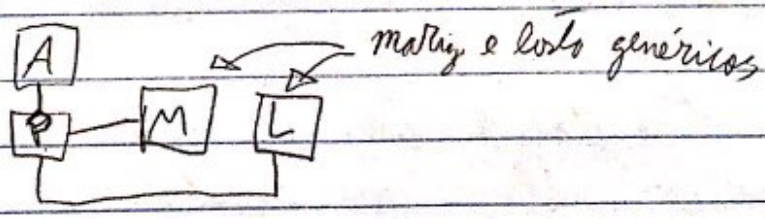


É a estrutura encapsulada que somente é conhecida pelos clientes.









/ /

(8) Propriedades de modularização

- Encapsulamento
- Acoplamento
- Coesão

(9) Encapsulamento

- Propriedade relacionada com a proteção dos dados de um componente de forma que este possa ser utilizado sem perder suas características básicas.
- Vantagens: manutenção facilitada, pois tudo relacionado à estrutura encapsulada está dentro do mesmo módulo.
- Facilita a documentação pois tudo que se encontra documentado trata de um único assunto / conceito.
- Desvantagens: exagero leva a muitos modulos muito específicos
- Tipos de encapsulamento:
  - de código: • função de acesso contido no módulo de implementação, que não é visto pelo módulo do cliente
    - for, while, if, switch, funções, ...
  - de variáveis: • static - classe ou módulo
    - local - bloco de código
    - private - objeto
    - protected - estruturas de herança
  - de documentação: • interno - do (.c) (desenvolvedor)
    - externo - do (.h) (interface)
    - de uso - usuário (ex.: leia-me)

(10) Acoplamento

- propriedade relacionada com a interface entre os módulos
- conector - item de interface
  - protótipo de função
  - arquivo
  - variável global aos módulos
- Critérios de qualidade de acoplamento
  - tamanho do conector
    ex.: func com 10 parâmetros VS func c/ 1 parâmetro
  - quantidade de conectores
    ex.: alocar na .h apenas o que for necessário para o cliente
  - complexidade do conector
    ex.: usar boa documentação p/ facilitar

(11) Coesão

- Propriedade relacionada com o grau de interdependência dos elementos que compõe o módulo (conceito)
- níveis de coesão
  - incidental - bagunça; não há relação entre os vários conceitos que existem no módulo (pior)
  - lógica - os elementos possuem uma relação lógica entre os conceito de uma func possivelmente genérica
  - temporal - os elementos estão relacionados pela necessidade de serem utilizados dentro do mesmo período de tempo
  - procedural - elementos que devem rodar em sequência (ex.: .bat)
  - funcional - os elementos estão relacionados por funcionalidade. Todos os elementos de uma mesma funcionalidade estão no módulo
  - abstração de dados - um único conceito

03/04/

Especificação de requisitos

(1) Requisitos
- o que deve ser feito
- nunca como deve ser feito

(2) Características de requisitos
- curtos e diretos
- linguagem natural

(3) Etapas de especificação
- elicitações (buscas de informações)
- técnicas
  - entrevista
  - brainstorm
  - questionário
  - documentação
- documentação
  - requisitos gerais
    específicos
- Verificação
  - Analisa se a documentação somente possui requisitos computáveis
  - junto com o equipe técnica
- Validação
  - cliente

/ /

(4) Tipos de Requisitos

- funcional
requisitos que devem ser implementados na aplicação relacionados com o negócio.
- não funcionais
propriedades que a aplicação deve possuir e que não necessariamente estão relacionados com o negócio
  a) Segurança
     - login e senha
  b) Disponibilidade
     - ex: 24X7 (24/7)
  c) Backup
  d) Velocidade
     Todas as consultas devem retornar resultado em no mínimo 3 segundos.
- Inverso
que a equipe não se compromete a fazer

(5) Exemplos de requisitos

a) bem formulados
↳ para cada aluno deve ser cadastrado matrícula e nome
↳ o relatório de turmas deve ser disponibilizado no 1º dia de matrícula

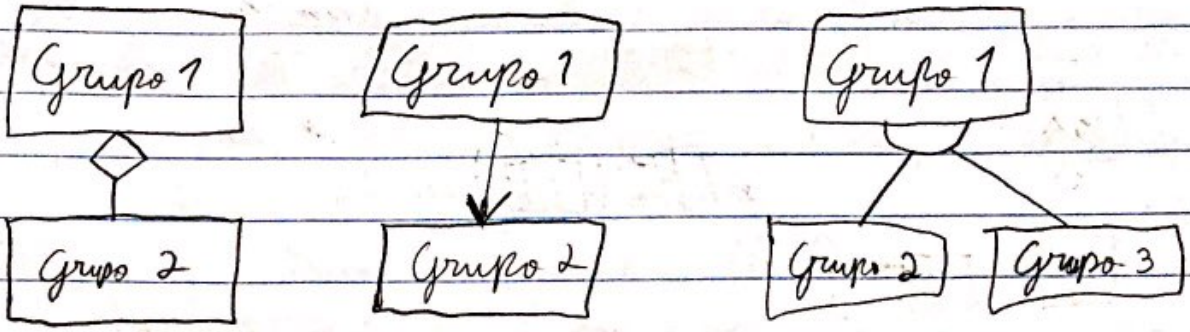
b) mal formulado
↳ a interface deve ser de fácil utilização
↳ o relatório apresenta seus dados mais necessários

08/04/19

Modelagem de Dados

1 modelo → n exemplos

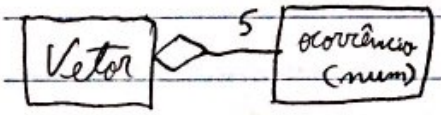
Notação: UML_like



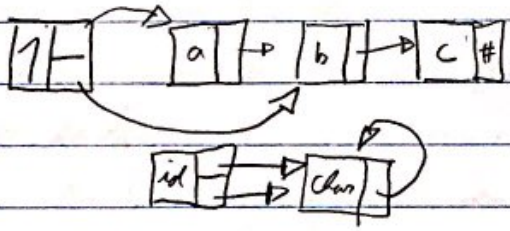
Agregação | referência | Particionamento

Exemplos:

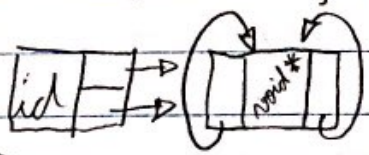
Vetor:

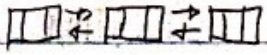


Lista simplesmente encadeada com cabeça

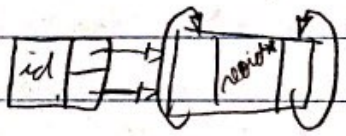


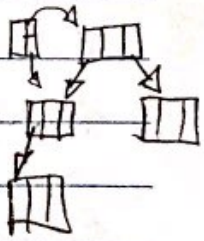
Lista duplamente encadeada genérica com cabeça





Árvore Binária genérica com cabeça



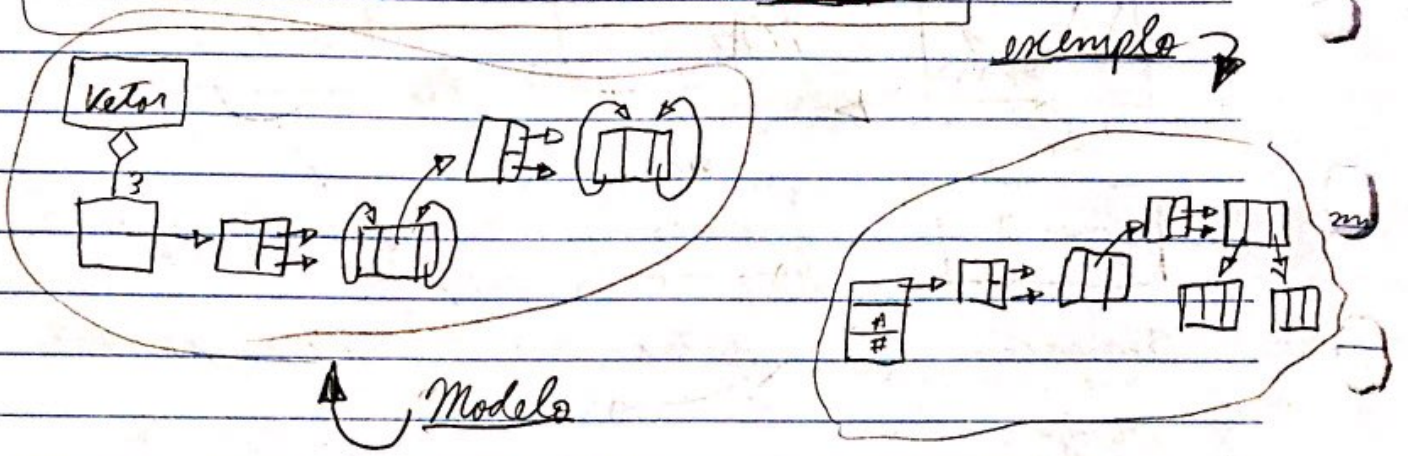


Assertivas Estruturais

LISTA	ARVORE
Se pCorr → pAnt != NULL pCorr → pAnt → pProx == pCorr Se pCorr → pProx != NULL pCorr → pProx → pAnt == pCorr	ponteiro de um nó da subarvore à esquerda nunca aponta para o pai ou para nó da subarvore à direita.

/ /

* alex: entrego do trabalho:
peço modelo → entrego-re: modelo
acrescimos
exemplo



Grafo

Assertivas

→ Regras consideradas válidas ao executar um determinado ponto do programa.

→ São utilizadas em argumentação de corretude

- Assertivas [redacted] Estruturais
  - → são assertivas estruturais regras que complementam os modelos de estruturas de dados.
- Assertivas de entrada e saída
  - → checa entrada e saída de um programa / função:
    ex.: parâmetros e retornos (se são válidos)

obs.: as assertivas precisam estar corretas e completas.

ex.:

- Assertivas de Entrada → • lista existe
  - possui pelo menos 3 nós
  - ponteiro corrente aponta p/ o nó intermediário que se quer excluir
  - valem as assertivas estruturais da lista duplamente encadeada com cabeça.
- func → Excluir nó corrente intermediário de uma lista duplamente encadeada com cabeça
- Assertivas de Saída → • nó foi excluído
  - valem as assertivas estruturais da lista duplamente encadeado com cabeça.
  - ponteiro corrente aponta para o 1º nó da lista.

OBS: Escrever (nos comentários) acima de cada função de acesso que forem desenvolvidas pelo gente as assertivas de entrada e saída.

T2: 1 ponto extra

T3: obrigatório

11

24.04

Implementação da Programação Modular

1) Espaço de Dados

→ São áreas de armazenamento alocadas em um meio
- Possuem um tamanho
→ Possuem um ou mais nomes de referência
ex.: A[i] → iésimo elemento do vetor A.
*ptAux → espaço apontado por ptAux.
ptAux → espaço que contém um espaço

função → ptElementoTabSimb* obterElementoTabSimb (char* ptSimbolo)
(*ObterElementoSimb(char* ptSimb)).Id → subcampo id do elemento retornado pela função
ou
obterElementoSimb(char* ptSimbolo) → Id → subcampo id do elemento retornado pela função

2) Tipos de Dados

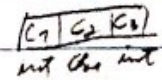
Determinam: • organização
• Codificação → como se interpreta o binário
• Tamanho em bytes
• Conjunto de valores permitido.

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

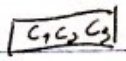
obs 2.: tipos de tipos:
• Tipo computacional → int, char, char*
• Tipos básicos → struct, union, enum, typedef
• Tipos abstratos de dados → estruturas de dados encapsulado

3) Tipos Básicos

- Struct



- Union



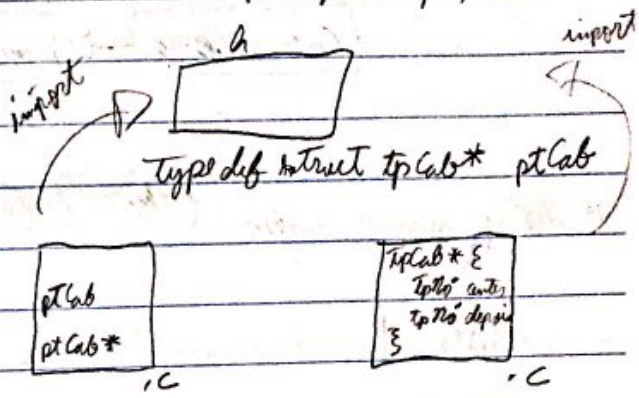

```
{ int C1
  float C2
  char C3
}
```

- Enum

```
{ a → 0,
  b → 1,
  c → 2
}
```

- TypeDef

```
typedef float TpVeloc
typedef float TpTempo
TpVeloc velocidade;
TpTempo Tempo;
```



4) Declaração e definição de elementos (Binding)

Definir: aloca espaço de dados e amarra expressão ao nome
Declarar: atribui tipo ao espaço

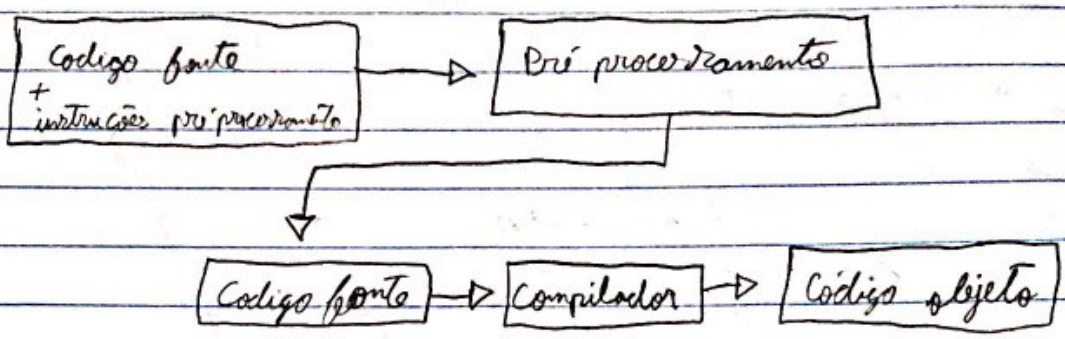
5) Implementação em C e C++

A) declarações e definições de nomes globais exportados pelo módulo servidor. ex.: int a;

B) declarações externas contidas no módulo cliente e que somente declara o nome sem atrelá-lo a um espaço de dados
ex.: extern int a;

C) declarações e definições de nomes globais encapsulados no módulo
ex.: static int a; (private)

(6) Pré-Processamento



define nome valor (troca todos até o # undef)

include <nome-arquivo>

if define (nome) ou # ifdef nome
texto V
else
texto F
endif

if ! defined
ou
ifndef

undef nome (para o # define)

```
# if !define (exemp_mod)
# define exemp-mod
    Corpo do .h
# endif
```

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

M1.h

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

```
# define exemp_ouun
# include "M1.h"
# undef exemp_ouun
```

```
int vetor [7] = {1,2,3,4,5,6,7}
```

```
extern int vetor[7];
```

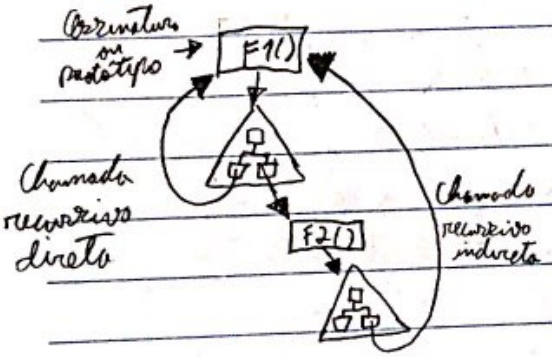
```
# include "M1.h"
```

06/05/19

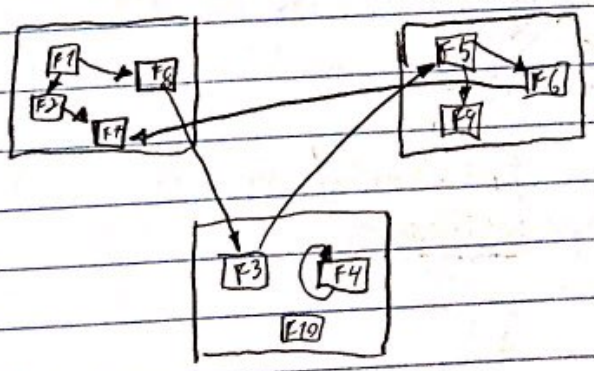
Estrutura de Funções

1) Paradigma
   - forma de programar
     - procedural
       - "receita de bolo"
     - orientado a objetos
       - P.O.O.
       - Programação modular

2) Estrutura de funções



3) Estrutura de chamadas

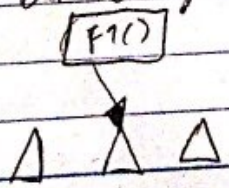


Arcos de chamada

F4 → F4 chamada rec. direta
F9 → F8 → F3 → F5 → F9 chamada rec. indireta
F10 função morta
F6 → F3 → F5 → F6 → F7 dependências cíclicas entre módulos
F1 origem

4) Função

É uma porção autocontida de código. Possui o nome, uma assinatura e corpo de código



tilibra

/ /

5) Especificação de função

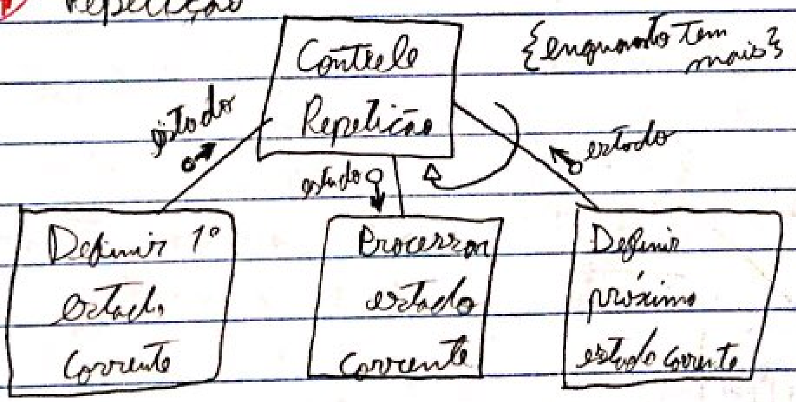
- Objetivo → se o nome da função é autoexplicativo, esse item pode ser ausente
- Acoplamento → parâmetros e condições de retorno
- Condições de acoplamento → assertivas de entrada e saída
- mmmmm
- Interface de usuário → mensagens, imagens e textos (console)
- Requisitos → tópicos que a função faz (descrição)
- Hipóteses → regras consideradas válidas antes da definição da função
- Restrições → regras que restringem as alternativas de solução utilizadas no desenvolvimento de uma aplicação.

6) Housekeeping

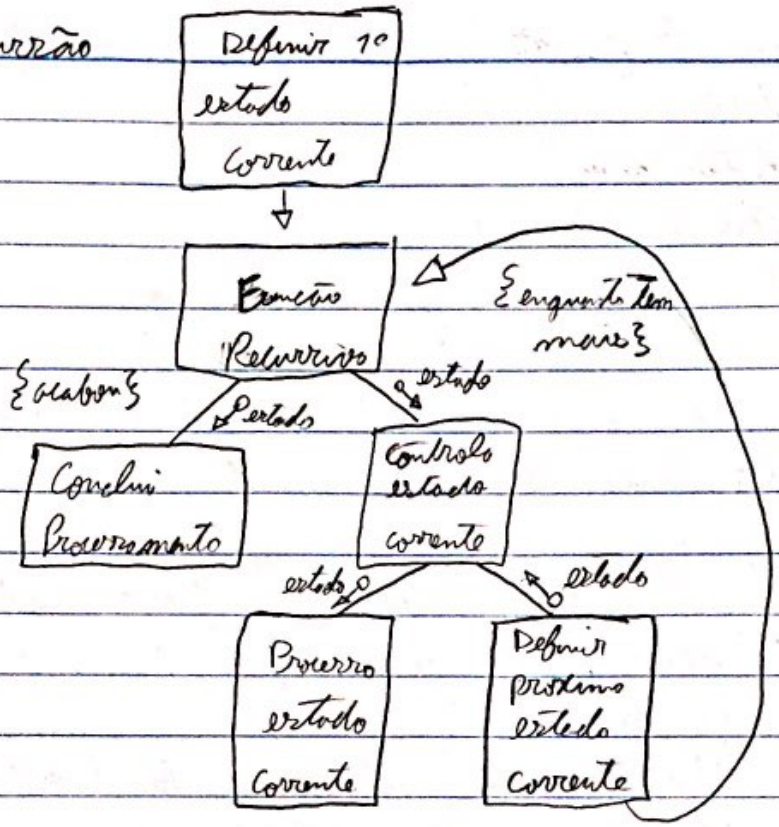
→ dar free p/ cada malloc que der

ou seja: módulo ou bloco de código responsável por liberar recursos alocados, programas, componentes ou funções, ao terminar termino execução.

7) Repetição



(8) Recursão



(9) Estado

descritor de estado

- variável ou variáveis que definem um estado
  ex.: pesquisa sequencial → índice; pesquisa bin. → inf e sup

estado

- valorização do(s) descritor de estado
  ex.: index = 0

(10) Esquema de Algoritmo

```
{ inf = obterLimInf();   <- hotspot
  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; }
  }
```

/ /

obs.: Esquema de algoritmo permitem encapsular a estrutura de dados utilizada. É correto, é incompleto e precisa ser instanciado

Normalmente ocorrem em:

- POO
- Framework

→ Se esquema correto e bem feito com assertivas válidas, então programa correto

(11) Parâmetros do tipo ponteiro para função: ~~float~~

```
float areaQuadrada (float base, float altura)
    { return base * altura; }

float areatri ( float valor1, float valor2, float (*func)(float,float))
  { printf("%f", func(valor1, valor2) }

CondRet = ProcessaArea(5, 2, areaquad);
CondRet = ProcessaArea(3, 2, areatri);
```