

Caderno de modular Larissa Binatti

p1 - 10/10 reposição 17/10

p2 - 10/12 rep. 12/12

livro: Programação modular - Arndt Von Staa

Professor Flávio - email: bevilac@inf.puc-rio.br (trabalha com mercado, analista de sistemas. Sabe muito de mercado e sistemas de informação)

critério de avaliação:

nota do trab 1, 2, 3, 4

50% da nota do trabalho vai ser caderno no pc. Toda aula tem que ser anotada no computador e ser entregue. Caderno é por pessoa e não por grupo.

50% trabalho do enunciado

obs: prof da ponto extra, até +2pt na prova

t1 → arcabouço de teste

t2 + t3 → trabalho do período

t4 → instrumentação

tudo no computador com capa, componentes, etc.

p1 e p2 tem consulta a qualquer coisa: pc, amigo, amigo que já fez a matéria. São provas que valem muito a pena consultar aluno antigo. Provas recorrentes. Mas não podem ter provas iguais ou conteúdo igual ao da internet. Tipo copiar a resposta do amigo ou copiar do wiki. Se faltar prova tem reposição na aula seguinte.

tem lista de exercícios, é por grupo e vale até 2pts extras na prova.

como calcular nota:

$$g1 = (t1 + 2t2 + 2p1) / 5$$

$$g2 = (t3 + 2t4 + 2p2) / 5$$

$$\text{se } g2 \geq 3$$

$$m = (g1 + g2) / 2$$

senão

$$m = (g1 + 3g2) / 4$$

para passar $m \geq 5$

NÃO TEM PF.

como resolver problema de dll de não abrir por não ter dependências(o que não pode acontecer): mandar executável aos poucos toda vez que fizer para o prof para ter certeza de que funciona. Mandar sempre do mesmo pc.

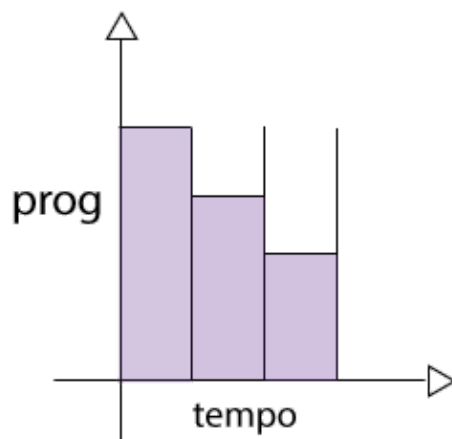
REPROVA POR FALTA. Não tem slide nem matéria online. Pode se ajudar entre grupos só não pode copiar.

Introdução

salvar as coisas em módulos para poderem ser reutilizados. Modularizar para pegar e reaproveitar em outros projetos.

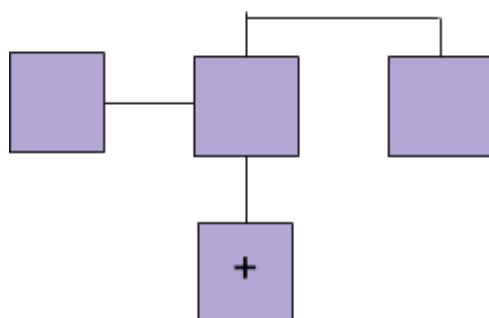
Vantagens da programação modular:

- vencer barreiras de complexidade. (cada pessoa chega até um ponto, quanto mais pessoas chega-se mais longe pois divide-se o trabalho em vários trabalhos menores que cada um faz sua parte)
- Facilita o trabalho em grupo. (questão de separar módulos e cada um pega x módulos e cada um vai fazendo sua parte ao mesmo tempo. PARALELISMO)
- reuso (usar o mesmo código em outros casos)
- facilita a criação de um acervo (= coleção)



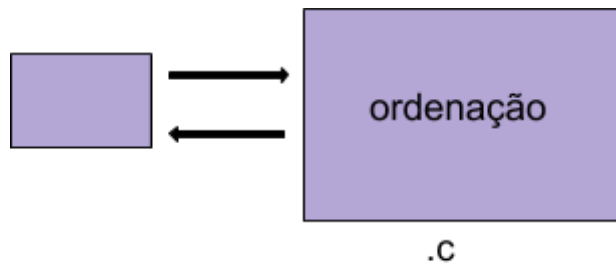
conforme vai se programando se faz um acervo de coisas para se usar depois. Vai salvando-se os módulos e quando se junta eles precisa programar menos a cada vez.

- desenvolvimento incremental (entregas). (incrementa aos poucos, após testes)



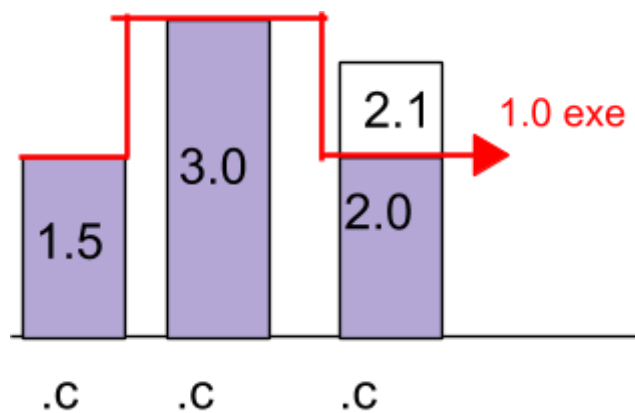
aproveitar as coisas do projeto anterior. Vai fazendo e testando aos poucos e ir mudando o código e ir aprimorando e fazendo aos poucos. Fazer todas as sub coisas bem feitas.

- aprimoramento individual (não precisa recompilar todo o resto, so o modulo necessário)



para aprimorar o programa é só ir no módulo que precisa ser melhorado. Não precisando mudar o programa (aplicação) todo.

- facilita a administração de baselines (controle de versão)



um programa que serve como base para quando modificar saber para qual voltar que não tem erro.

Princípios de modularidade

- 1) **módulo:** (o objetivo é fazer o mínimo de coisa por módulo, cada “e” que faz a mais pior) (só o que importa é a interface do módulo) (cada um nosso vai ser um .c)

definição física: unidade de compilação independente

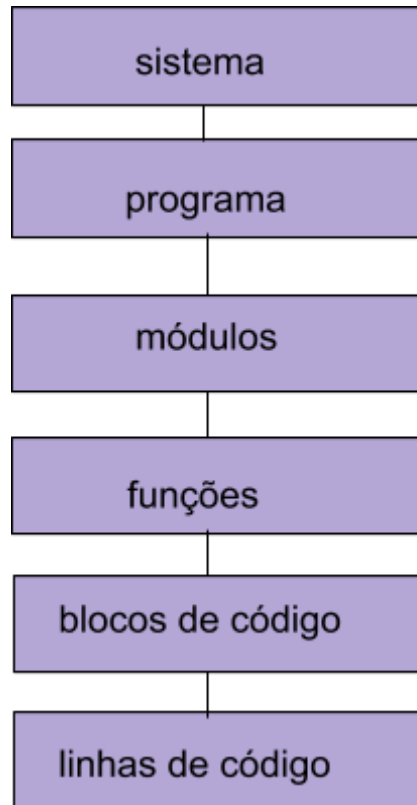
definição lógica: trata-se de um único conceito

quanto mais coisa feita dentro de um módulo mais difícil achar o erro pois não saberá onde está o erro. Objetivo é cada módulo faz 1 coisa só. Porque quando conserta um erro de um conceito pode cagar o outro conceito se estiver no mesmo módulo.

- 2) **Abstração de sistema :** (definir o problema e dizer o que faz e o que não faz parte dele)

abstrair é o processo de considerar apenas o que é necessário em uma situação e descartar com segurança o que não é necessário → ESCOPO!

- níveis de abstração: (hierarquia)



obs: **conceitos:**

1) artefato: item com identidade própria criado dentro de um processo de desenvolvimento. Pode ser versionado!!(controlado por controle de versão)(tipo docs tipo final.doc final2.doc etc)

2) build (construto): resultado apresentável. Algo que possa mostrar da aplicação sem estar final/pronto. Artefato que pode ser executado, mesmo que incompleto.

3) interface:

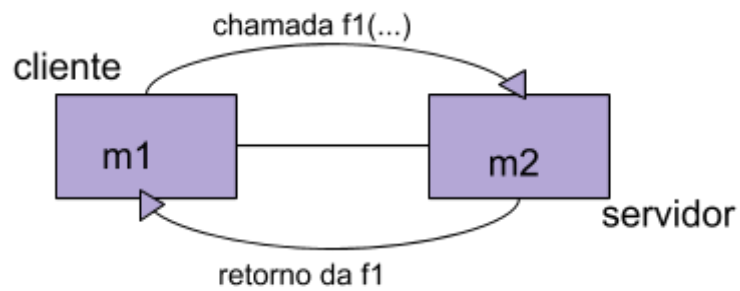
é o mecanismo de: troca de dados, estados e eventos entre elementos de um mesmo nível de abstração.

(módulos se comunicam com funções. Uma função manda o resultado para outra. Variável global comunica blocos de código pois ela muda em vários lugares e blocos de código)

a) exemplos de interface:

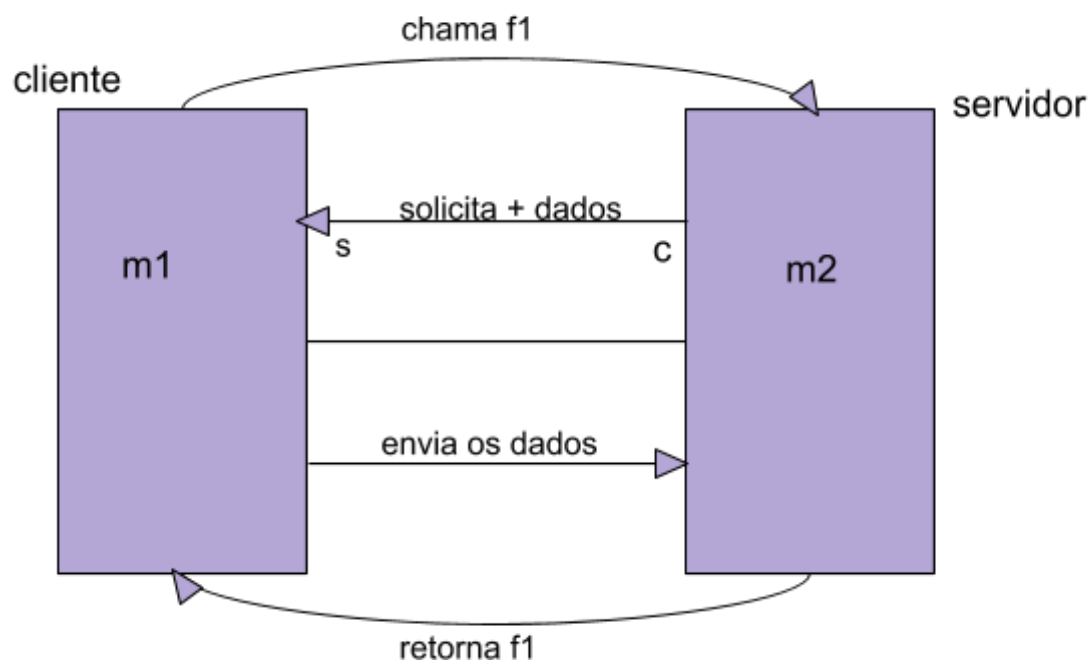
- arquivo (entre sistemas)
- funções de acesso (entre módulos) (no header? .h é interface também)
- passagem de parâmetros
- variáveis globais (entre blocos)

b) relacionamento cliente - servidor:



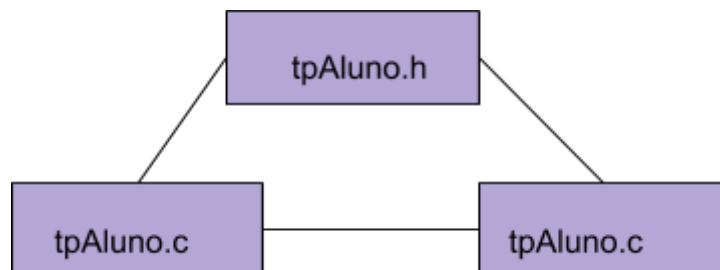
No relacionamento cliente-servidor, o cliente faz uma requisição ao servidor, e o servidor fornece uma resposta. E eles podem trocar de lugar, depende de quem chamar.

caso especial: **callback**



quando um cliente envia um pacote para um servidor com dados incompletos, fazendo o servidor requerir mais dados do cliente para completar a solicitação.

c) interface fornecida por terceiros:



como dois desenvolvedores podem trabalhar em um mesmo módulo ao mesmo tempo isso pode gerar **duplicidade de código** (além de diferenças e incompatibilidade entre arquivos). Como: ter uma struct com mesmo nome em arquivos diferentes e que fazem a mesma coisa. Duplicidade é barreira pra modularização pois qualquer modificação deve ser feita em todos os lugares que o código aparece. Para resolver isso é só usar uma interface fornecida por terceiros (tipo .h) para que o código de ambos os desenvolvedores se comuniquem e tirando duplicidade de código.

Definição: Uma interface é fornecida por terceiros se contém um struct que é utilizado por dois módulos (Definição da aula 3).

d) Interface em detalhe

A passagem de parâmetros entre funções devem ser do mesmo tipo dos parâmetros recebidos(mesmo podendo ter casting automático de tipos, para que a sintaxe fique correta).

- **sintaxe:** respeitar as regras. Determina as regras de como um programa deve ser escrito. ex: se no módulo 1 é int no mod2 não pode ser float.
- **semântica:** respeitar os significados. determina o significado das variáveis e parâmetros passados. ex: se no mod1 o int é idade no mod2 o mesmo int não pode ser quantidade de filhos.

e) Análise de Interface

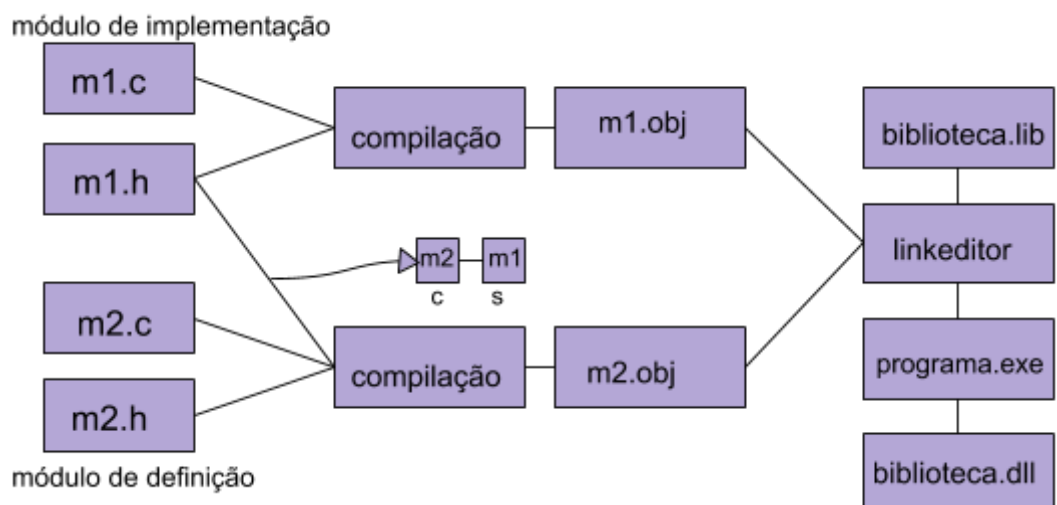
tpDadosAluno* obterDadosAluno(int mat); (assinatura ou protótipo de função de acesso)

Interface esperada pelo cliente: Um ponteiro para os dados válidos do aluno correto ou NULL.

Interface esperada pelo servidor: Um inteiro válido representando a matrícula de um aluno.

Interface esperada por ambos: tpDadosAluno (e int).

4) processo de desenvolvimento



Ao compilar, **módulos físicos de implementação .c** (compilados independentemente) e **interfaces .h** (que são os **módulos de definição**) são traduzidos para assembly pelo compilador e em seguida traduzidos (pelo montador assembly) em **linguagem de máquina** em arquivos **.obj**. Então o LinkEditor junta os .obj em tempo de linkedição, possivelmente com uma **biblioteca estática .lib**. E em tempo de execução o arquivo o **executável (.exe)** gerado pelo linkeditor pode executar com referências para uma **biblioteca de execução dinâmica (.dll)**. Os .h são interfaces entre os módulos, sempre que um módulo quer chamar uma função que está em outro módulo se faz pelo .h (pra incluir a interface toda do .h em um módulo é só usar o #include).

5) Bibliotecas estáticas e dinâmicas

- Estática

Vantagens: A .lib já é acoplada em tempo de link edição à aplicação executável.

Desvantagens: Existe uma cópia desta biblioteca estática para cada executável alocado na memória (e no disco) que a utiliza.

- Dinâmica

Vantagens: Só tem uma instância da biblioteca dinâmica na memória, mesmo que várias aplicações a acessem.

Desvantagens: Não está acoplada ao executável em link edição. Se a .dll não estiver na máquina que tenta executar uma aplicação que dependa dela, será impossível executar a aplicação. A dll precisa estar na máquina para a aplicação funcionar.

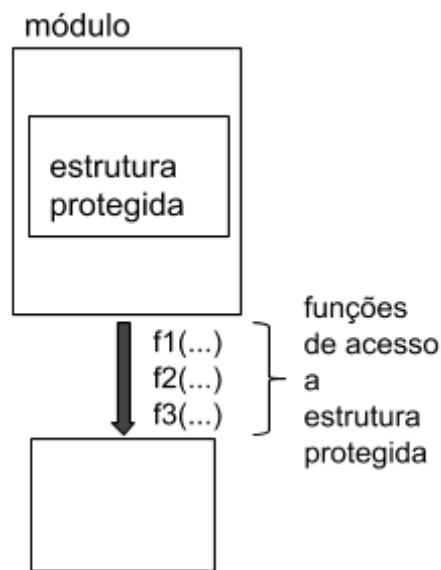
6) Módulo de definição (.h)

- Interface do módulo
- Contém os protótipos (assinaturas) das funções de acesso e interfaces fornecidas por terceiros. (ex: tpDadosAluno do item 3e).
- Documentação voltada para os programadores dos módulos clientes.

7) Módulo de implementação (.c)

- Código das funções de acesso.
- Códigos e protótipos das funções internas (auxiliares).
- Variáveis Internas ao módulo.
- Documentação voltada para o programador do módulo servidor.

8) Tipo abstrato de dados (TAD)



em orientação a objetos:

- atributos
- métodos

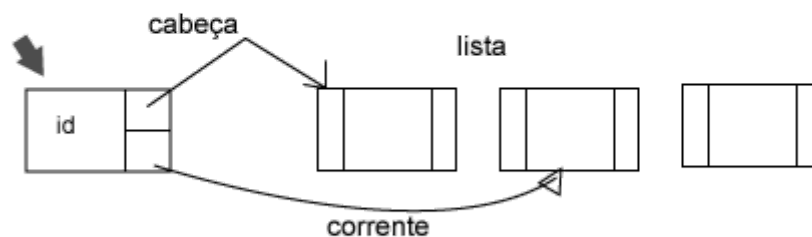
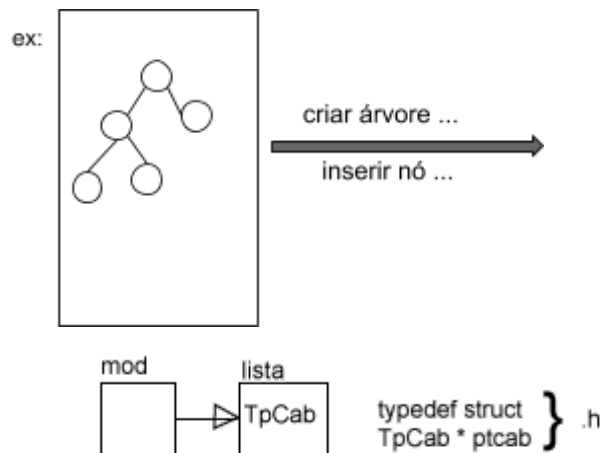
notas:

estrutura encapsulada.

manipula funções de acesso e não os dados e estruturas em si.

uma estrutura protegida por um módulo, você só tem acesso a interface de interação com dados.

definição TAD: estrutura encapsulada em um módulo que somente é conhecida pelos módulos clientes através das funções de acesso disponibilizadas na interface.



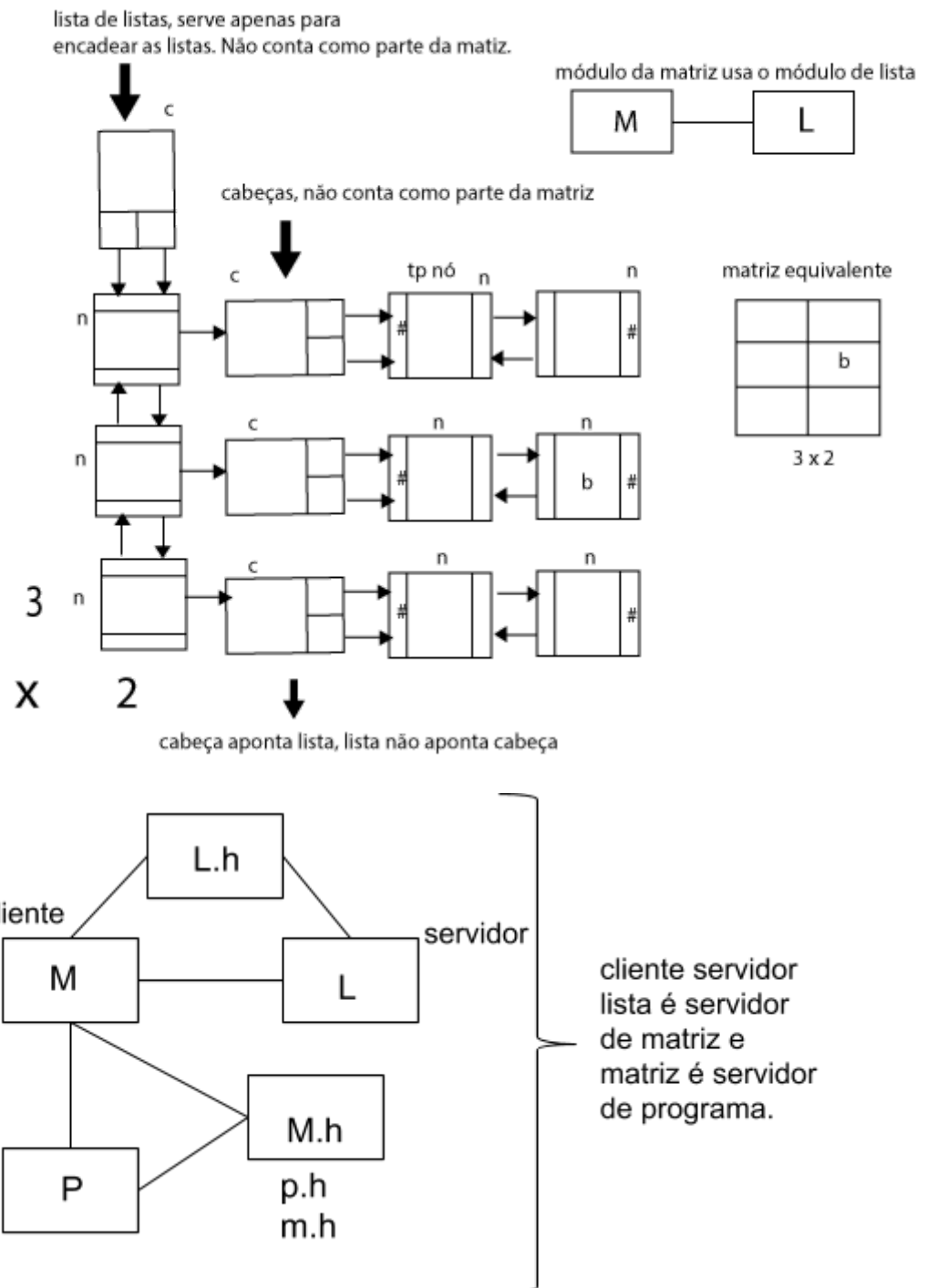
`criarLista(tpCab **plista)` passagem de parâmetro

`criarLista(ptCab *plista)` por referência

`inserirNo(ptCab plista, void *conteudo)` passagem por valor

irprox(ptcab plista)
 obter(ptcab plista no)

o nó cabeça(que serve para nao perder a lista) é uma struct diferente que aponta para os nós que são outra struct. Nó cabeça não conta como nó então você ignora a linha ou coluna do cabeça. Assim como a âncora de listas.

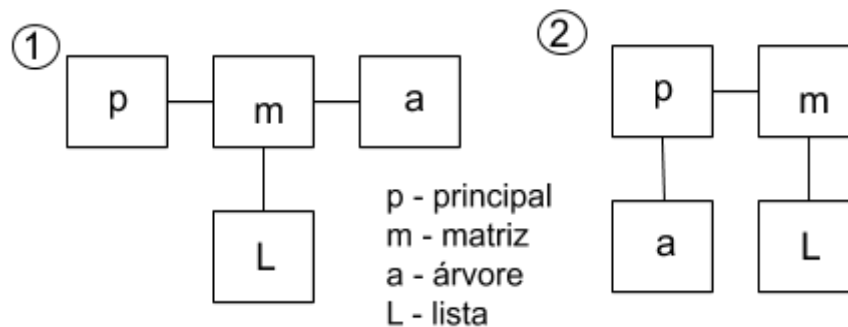


criarMatriz(2, 2)
 criaLista(p1) //lista vertical com cabeça
 criaLista(p2) //lista 1 horizontal c/ cabeca

```

inserirNo( p2, Null) //no da lista 1
inserirNo( p2, Null) //no da lista 1
inserirNo( p1, p2) //põe a lista 1 no primeiro nó da lista vertical
criaLista( p3)
inserirNo( p3, Null)
inserirNo( p3, Null)
inserirNo( p1, p3)

```



Qual dos dois é mais genérico? O 2 é o caso mais genérico.

caso 1: Aqui é uma matriz de árvore, uma matriz com árvores dentro, logo é mais fechado e menos reutilizável. Matriz sempre presa a árvore.

caso 2: Aqui é uma matriz qualquer/ genérica que pode ser usada em árvores.

L (lista) fica sempre com a matriz pois a estrutura da matriz depende de lista.

O programa p da segunda estrutura é mais complexo e faz mais coisas pois ele tem que juntar a matriz com a árvore.

9) Encapsulamento

- Propriedade relacionada com a proteção dos elementos que compõem um módulo.
- objetivo:
 - facilitar a manutenção (precisa apenas olhar o módulo que deu erro)
 - impedir a utilização indevida da estrutura do módulo (restringe acesso aos dados originais por uma interface)

Outros tipos de encapsulamento:

- **de documentação**
 - doc interna → módulo de implementação (.c)
 - doc externa → mod. definição (.h)
 - de uso → manual do usuário
- **de código**
 - blocos de código visíveis apenas:
 - dentro do módulo
 - dentro de outro bloco de código (ex: conjunto de comandos dentro de um for)
 - código de uma função

- de variáveis

private (apenas do objeto. Encapsulado no objeto),
public, global, (public = global só que public é orientada a objeto)
global static,
protected, (estrutura de herança)
static, (classe. Static = encapsulada no módulo)
etc...

10) Acoplamento

Propriedade relacionada com a interface entre os módulos. (a interface precisa ser não redundante e ter tudo, ser completa)

Conector:

- cada unidade interfaceada entre os módulos
- item de interface. Ex: função de acesso, variável global, etc.

Critérios de qualidade: (ver se tudo na interface é necessário e suficiente)

- quantidade de conectores
 - necessidade X suficiência
 - tudo é útil? falta algo?
- tamanho do conector
 - ex: quantidade de parâmetros de uma função.
 - como diminuir o número de parâmetros de uma função?
 - Agrupando em estrutura de dados. Tipo struct endereço com nome, num, ap... tirar coisas desnecessárias também.
- se tem mais de um conceito nos conectores diz que o módulo está com mais coisa do que devia.
- complexidade do conector
 - explicação em documentação
 - utilize mne = mnemônicos (dar bons nomes as variáveis, autoexplicativos)

Acoplamento bom é baixo pois quanto mais acoplado pior. Porque tem menos funções de acesso? Como o acoplamento é relacionado com conector de interface quanto menos coisa pra ligar mais facil é ligar. Caso dos fios: se tem 1 fio ligando é mais facil de ligar do que tendo 10 fios.

11) coesão

Propriedade relacionada com o grau de interligação dos elementos que compõem um módulo. Tudo dentro do modulo tem que estar relacionado, tudo dentro do modulo tem que ter 1 conceito. Quanto mais conceito em um módulo pior pra manter.

- **níveis de coesão:**

- incidental - (não tem lógica e relacionamento dos elementos. Uma coisa não tem a ver com a outra). PIOR COESÃO.
- lógica - elementos logicamente relacionados (porém cada um tem sua lógica. Ex: como uma função calcula que calcula qualquer coisa. Abraça muita coisa apesar de tudo ser cálculo)
- temporal - itens que funcionam em um mesmo período de tempo (tipo fim de semestre que junta várias funções que são usadas na mesma hora. Tipo nota de disciplina e matrícula).
- procedural - itens em sequência (.bat por ex. Interligação em sequência).
- funcional - tudo dentro de uma funcionalidade.(como gerar relatório). Agrupados por funcionalidade que acaba sendo lógica porque lógica é subconjunto de funcional.
- Abstração de dados (melhor)
 - um único conceito. Ex: TAD

Teste automatizado

Criar um **script de entrada** para fazer o teste de forma automática para testar um programa ao invés de fazer manualmente.

Para evitar refazer todos os testes se der erro em alguma parte. Forma de testar todos os programas varias vezes.

Aí fica um log de saída com ok e erro.

Mudar as partes que tem erro para tentar deixar tudo ok.

ler o **log de saída**: chegou no primeiro erro ignora tudo q ta em baixo. Corrige primeiro o primeiro erro e depois disso roda de novo pra saber se funcionou. tudo q ta em baixo do primeiro erro não é confiável.

Qual o objetivo do teste provar que funcionou ou provar que tem erro? Provar que tem erro.

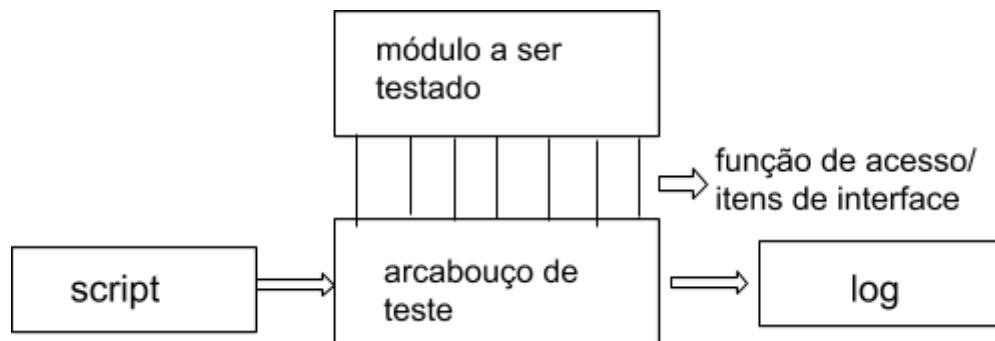
Um teste não prova que um programa tá funcionando pois você nunca sabe se seu teste está completo o suficiente e se conseguiu testar tudo.

1) objetivo

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.

obs: a partir do primeiro retorno esperado diferente do obtido no log de saída todos os resultados de execução em caso de teste não são confiáveis.

2) framework de teste



cada traço é uma função de acesso(itens de interface)

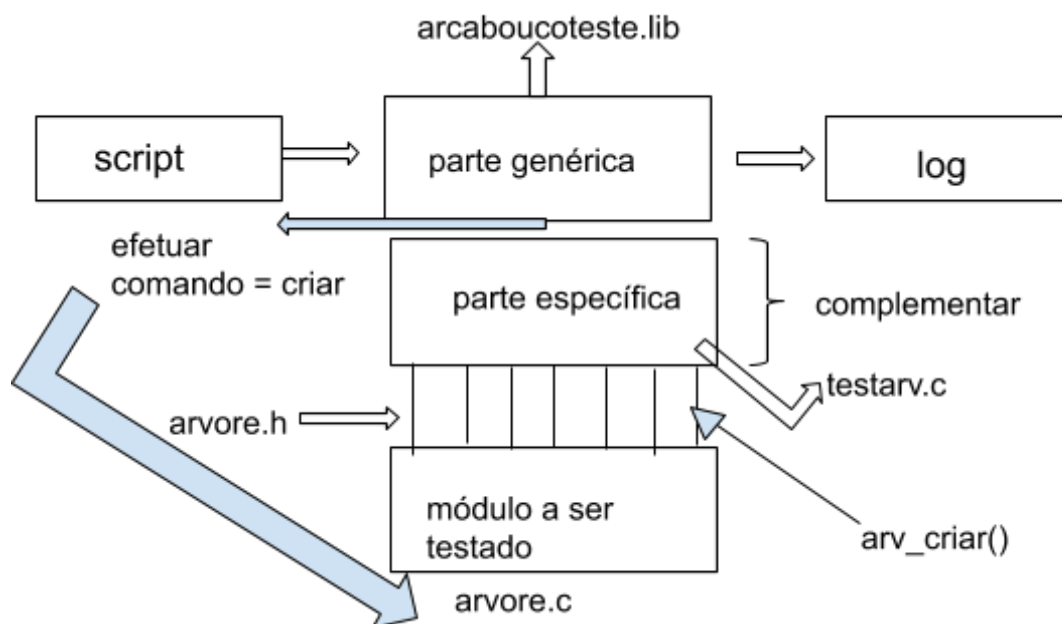
No arcabouço, existe uma parte genérica e uma parte específica. A parte genérica lê o script e produz o log, e a parte específica determina a semântica dos comandos de teste presentes no script, que podem ser diferentes para cada aplicação. A parte específica liga cada comando do script a uma função de teste para o módulo a ser testado, ex: funções "criarArvore", "inserirArvore". A parte específica deve ter acesso, por uma interface, ao módulo a ser testado.

Script tem o comando criar e tem que juntar com a função específica do módulo de criar.

Módulo a ser testado é o módulo árvore que tem o arv criar().

parte genérica toda tá no arcabouço.lib

a parte específica é testaarv.c



3) script de teste

é como se fosse um programinha escrito em uma outra linguagem

// → comentário

== → caso de teste

- testa determinada situação
(tentar algo dar errado numa situação geral. ex: em uma lista vazia tem q excluir um nó.
inserir -> inserir -> inserir -> ir ant -> excluir corrente
5 comandos no total)

= → comando de teste

- associado a uma função de acesso

caso de teste não pode repetir mas pode repetir comando de teste!!!!

você só testa a situação uma vez, não tem porque testar a mesma coisa várias vezes

Mas vc pode usar função de criar por exemplo várias vezes. as funções podem ser usadas várias vezes.

(obs meu: um testador completo é aquele que testa todos os valores de retorno de uma função, se não checar todos é um teste superficial)

obs prof: **teste completo** -> casos de teste para todas as condições de retorno de cada função de acesso do módulo. (exceto condições de retorno de estouro de memória).

4) log de saída

O **log de saída** de um script, imprime todos os comentários e casos de teste e as saídas de cada comando.

esperado(certo)

== caso 1

== caso 2

== caso 3

1>>função esperava 0 e retornou 1 (erro)

2>>

3>>

...

= recuperar (desfaz o último erro)

tipo

1>>

0<< ← = recuperar (ele fica na parte genérica)

ai continua

Se uma condição de teste falha e em seguida o comando '**=recuperar**' é usado, esta falha será ignorada e não será contabilizada na contagem de falhas.

você cria uma função fazendo um erro proposital pra testar o comportamento do arcabouço.

Aí você recupera pra regularizar e tira o erro.

pra que serve isso? um teste para o arcabouço para ver se ele ta funcionando perfeitamente se tem ele retornando errado é pra ver oq o arcabouço faz qd tem um resultado não esperado.

tem o caso de erro não previsto-> vc espera um retorno da função serve para testar o arcabouço e não a aplicação

no final tem um relatório de erros não previstos.

5) parte específica

A parte específica que necessita ser implementada para que o framework (arcabouço) possa acoplar na aplicação chama-se **hotspot**.

ex: testArv.c

```
#define criar_arv_cmd "=criar"
```

tem o num de defines igual o número de funções de retorno

efetuar comando(comando teste)

```
if(strcmp(comandoteste, cria arv cmd)
```

processo de desenvolvimento em engenharia de software

tudo começa com um cliente querendo alguma coisa

demanda (cliente) → analista de negócio(contrato) (aciona a equipe que lida c/ a demanda)

→ líder de projeto (equipe escolhe um líder de projeto)

líder de projeto

→ projeto

- esforço
- recursos
- prazo
- tamanho (ponto de função = técnica para definir o tamanho de uma aplicação. o problema é que ele só avalia a entrada e a saída, o que tá no meio é ignorado o que gera problemas tipo vc fez um bd todo pra rodar uma pequena tela de busca. os pontos de função medem a complexidade o que pode dar errado)

→ estimativa (pto de função)

→ planejamento

→ acompanhamento (acompanhamento dia após dia pra ver se o planejamento está andando)

como saber prazo de algo que nunca fiz?

estimativa é chute? sim, mas tem q ser controlado

como fazer a mágica de acalmar cliente?

relatar dia após dia como está o projeto: tipo: você vai falando olha acho q vou acabar antes, fala isso de novo e pa. Ou então tou me atrasando, po tá dando problema.

como otimizar a expectativa(de tempo)? experiência.

acertar estimativa faz o cliente confiar mais.

-elevar a credibilidade c/ o cliente com acertar a estimativa de prazo.

como faz pra entregar menos e deixar o cliente feliz? negocia mais de uma entrega estipula um mínimo (escopo que ele próprio se auto resolva, mínimo pra funcionar) e depois deixa o resto p/ próxima entrega.

Requisitos

tem q ser incisivo com a regras não pode ter acho. Se tiver acho, para e pergunta o que é. não pode deixar NENHUM acho em especificação. nao tem acho.

- elicitação (entrar em contato c/ o cliente para pegar as especificações necessárias)
- documentação (não pode ter nenhuma ambiguidade. Se tiver ambiguo você fica na mão do cliente, ele pode pedir retrabalho de graça. por isso não pode)
- verificação (isso que o cliente pede faz sentido? feita por uma equipe que desenvolve. ver se o cliente nao ta pedindo o impossível porque rola né)
- validação (pelo cliente, é oq vc se dispõe a entregar pro cliente)

[isso tudo vai para o desenvolvedor que transforma o português para prog? NÃO. Tão tentando transformar implementador em analista]

isso vai para análise e projeto (analista diz o que o implementador tem que programar. o analista que interpreta o documento e transforma para algo programável)
que vai pro implementador (esse só coda mesmo e sabe a linguagem de prog e caga pro negócio. mas se for bom sabe um pouco do negócio)
que vai para testes

requisitos → análise e projeto (1) → implementação (2) → testes (3) → homologação (cliente) (4) → implantação

←

- (1) - projeto lógico (tipo modelagem de dados)
 - projeto físico (tabelas com os dados)
(telas, programas, etc)
transformar texto em modelos

- (2) - programas
 - teste unitário

- (3) - teste integrado

(4) obs: tipo um feedebck do cliente, achar erros do tipo que eram esperados. era esperado algo e tem outro. ai ve se foi erro de especificação ou falha de implementação da equipe.ai é sugestao ou erro respectivamente.

- sugestão
- erro

agora como negocia com o cliente? corrige os erros encontrados, sugestões dependem se devem ser mudados ou não. ai se for necessária vira um retrabalho que precisa ser cobrado. ai só cobra retrabalho se for mudança real, se for algo muito simples como só um label melhor nem cobrar.

caminho crítico: tarefas que não podem atrasar senão atrasa a entrega

- gerência de configuração
- qualidade de software (saber se todo procedimento tá dentro do esperado, fica mensurando o tempo todo se tá tudo mantendo a qualidade durante todo o processo de desenvolvimento)

a boa é sempre ir mostrando aos poucos pro cliente.

Especificação de requisitos

1) definição de requisitos

o que tem que ser feito. (detalhes do que fazer)

Não é **como** deve ser feito.

requisito tem que ser uma frase só. ser objetiva, clara e certa.

(quando você tá discutindo c/ o cliente e fica pensando em como implementar é uma péssima ideia)

2) escopo de requisito

- requisitos mais gerais/genéricos (escopo amplo)
- requisitos mais específicos (coisas a serem usadas em funções, etc)

3) fases da especificação

3.1) elicitação (pegar informação com o cliente)

→ captar informações do cliente para realizar a documentação do sistema do sistema a ser desenvolvido.

Técnicas de elicitação:

- entrevista (o que é necessário para gerar o sistema, a dificuldade é filtrar o que é importante e o q perguntar)(só ajuda o cliente a decidir coisas técnicas, não decidir nada do negócio do cliente)
- brainstorm (precisa saber controlar)
- questionário (péssima opção)/
e isso gera uma **ata**(documento sobre a reunião) caótica coisas que precisam entrar no sistema, coisa que não precisa, coisa pessoal das pessoas.

3.2) documentação

- requisitos descritos em itens diretos

qd vc pega essas informações desorganizadas e põe no formato de requisitos simples.

- uso da língua natural. CUIDADO com ambiguidade.
- dividir requisitos em seus diversos tipos

obs. tipos de requisitos:

- a) requisitos funcionais → o que deve ser feito em relação a informatização das regras de negócio

- b) requisitos não funcionais → são propriedades que a aplicação deve ter e que não estão diretamente relacionadas com as regras de negócio (como por login e senha para segurança) .

Ex:

- segurança(login e senha)
- tempo de processamento (limite de tempo p consulta, nao podem demorar mais do q 5s)
- disponibilidade (24 horas x 7 dias)

- c) requisitos inversos → é o que não é para fazer
quem define os requisitos inversos é o analista de requisitos e não o cliente. pq ele define isso? pq pode dar conflito com outro código, por segurança da documentação/ contrato. Serve para resolver ambiguidade e tirar coisas implícitas. explicitando oq nao faz parte. Aí quando o cliente nota que quer algo explícito que não faria ele pede pra por. se ele nao dizer nada quer dizer que o programador não precisa fazer e se ele reclamar não importa/ dane-se.

3.3) Verificação

a equipe técnica verifica se o que está descrito na documentação é viável de ser desenvolvido.

3.4) validação

- cliente valida a documentação

negócio: negócio do cliente

4) exemplos de requisitos

- a) bem formulados

- a tela de resposta da consulta de aluno apresenta nome e matrícula
- todas as consultas devem retornar respostas em no máximo 2 seg

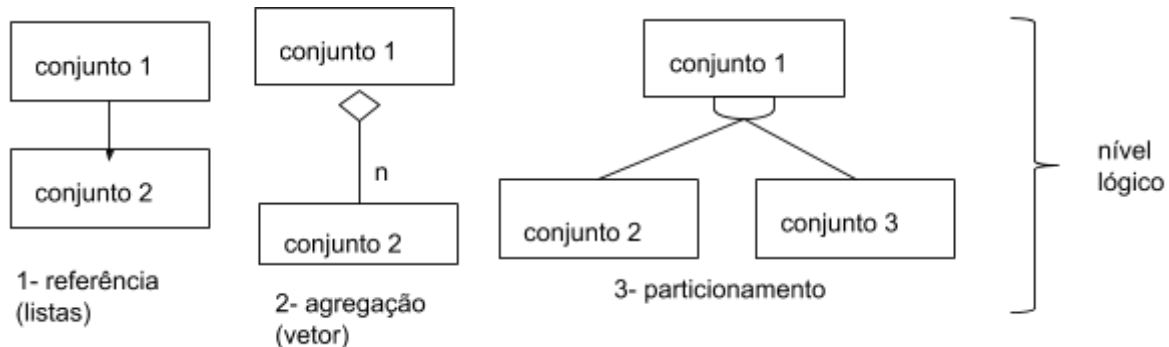
- b) mal formulado

- o sistema é de fácil utilização
- a consulta deverá retornar uma resposta em um tempo reduzido
- a tela mostra seus dados mais importantes

Modelagem de dados

Modelos:

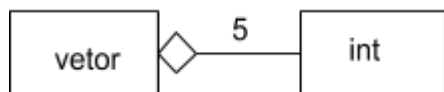
1 modelo \rightarrow n exemplos



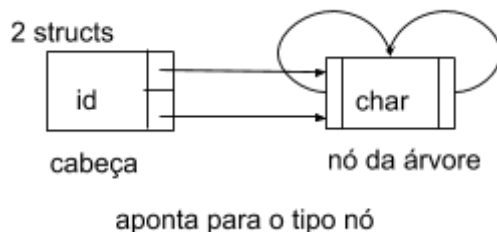
1- ponteiro: conj1 ponteiro pro conj2. \rightarrow = ponteiro (estrutura dinâmica)
2- 1 elemento do conj1 tem n elementos do conj2 (partes do 2) (estrutura estática)
3- ou 1 elem do conj2 ou 1 elem do conj3 (tipo especialização: conj1 eh pessoas do dpt e conj2 eh professor e conj3 eh alunos)
como abstrair uma estrutura para saber o que é importante nela para implementar.

Exemplos:

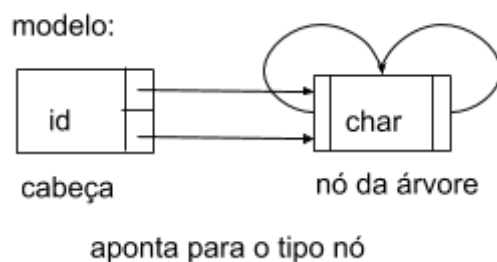
a) vetor de 5 posições que armazenam inteiros



b) árvore binária com cabeça que armazena caracteres



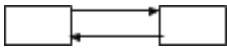
c) lista duplamente encadeada com cabeça que armazena caracteres



árvore binária é o mesmo desenho da estrutura que lista. mesmo modelo representa duas estruturas diferentes, o que não pode acontecer. Como resolve? com assertivas estruturais.

Assertivas estruturais:

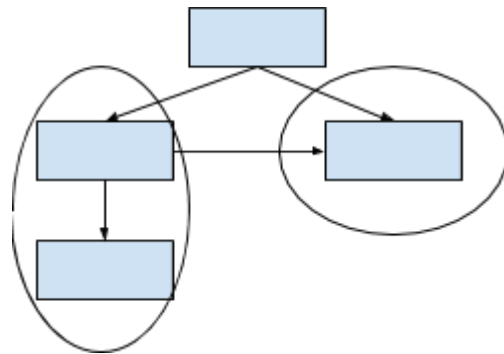
def: São regras utilizadas para desempatar dois modelos iguais. Estas regras complementam o modelo, definindo características que o desenho não consegue representar.

Lista: 

- se $pCorr \rightarrow pAnt \neq \text{nulo}$ então $pCorr \rightarrow pAnt \rightarrow pProx == pCorr$
- se $pCorr \rightarrow pProx \neq \text{nulo}$ então $pCorr \rightarrow pProx \rightarrow pAnt == pCorr$

Árvore:

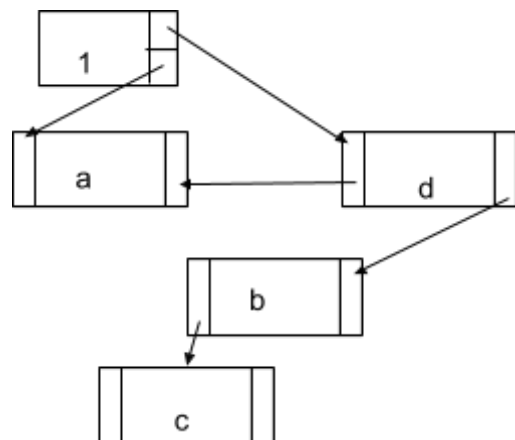
- Um ponteiro de uma subárvore à esquerda nunca referencia um nó de uma subárvore à direita.
- $pAnt$ e $pProx$ de um nó nunca aponta para o pai.



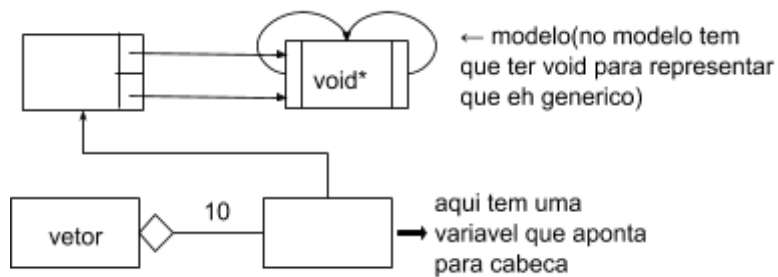
nos trabs 2, 3 e 4:

- modelo(tem que fazer o modelo(desenho) dos TAD)
- assertivas
- exemplo (não pode ter contra exemplo de modelo)

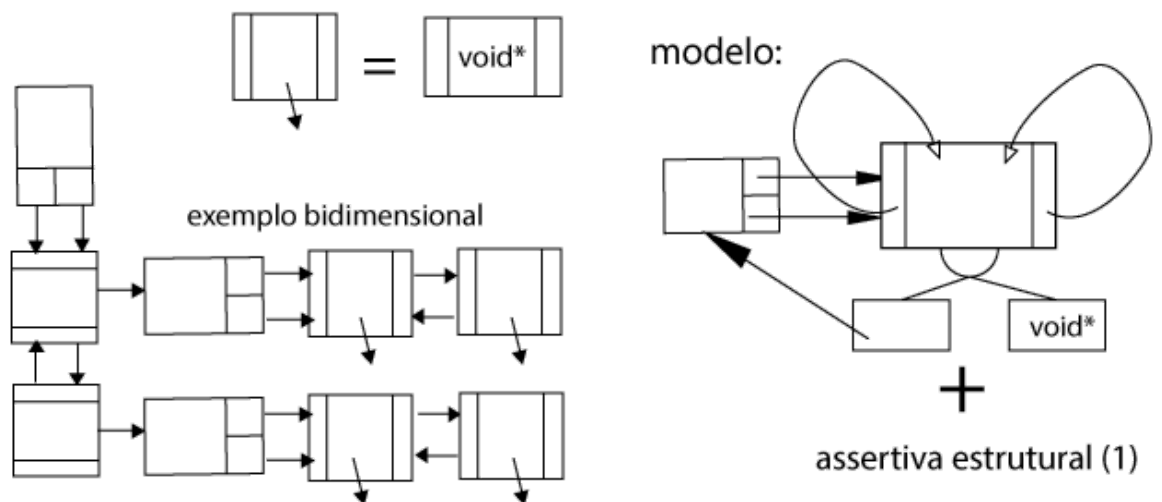
exemplo de desenho struct de árvore:



d) vetor de listas duplamente encadeadas com cabeça e genéricas(void)



e) matriz tridimensional genérica construída com listas duplamente encadeadas com cabeça

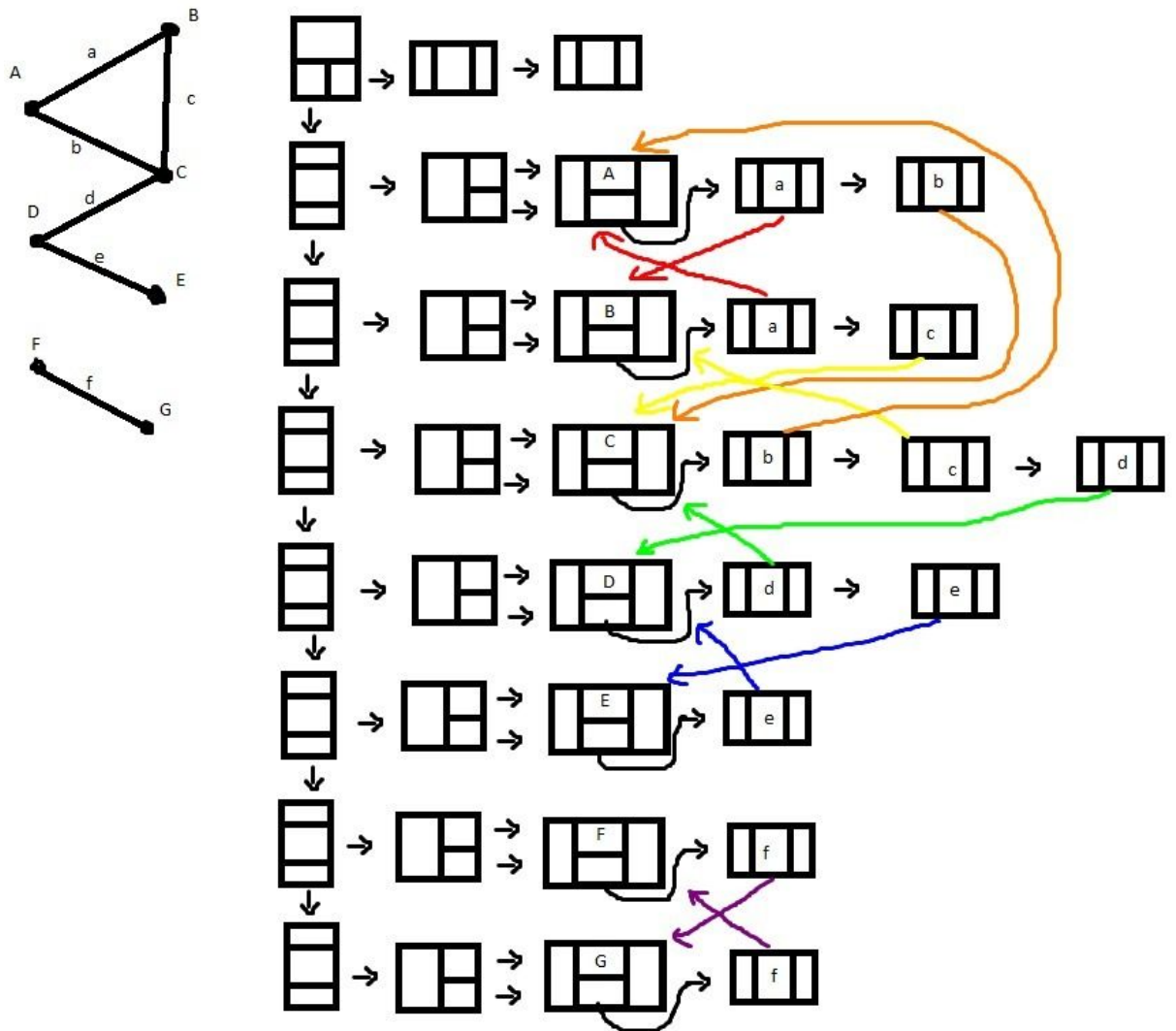


obs: o exemplo de matriz tridimensional é um cubo de cubinhos

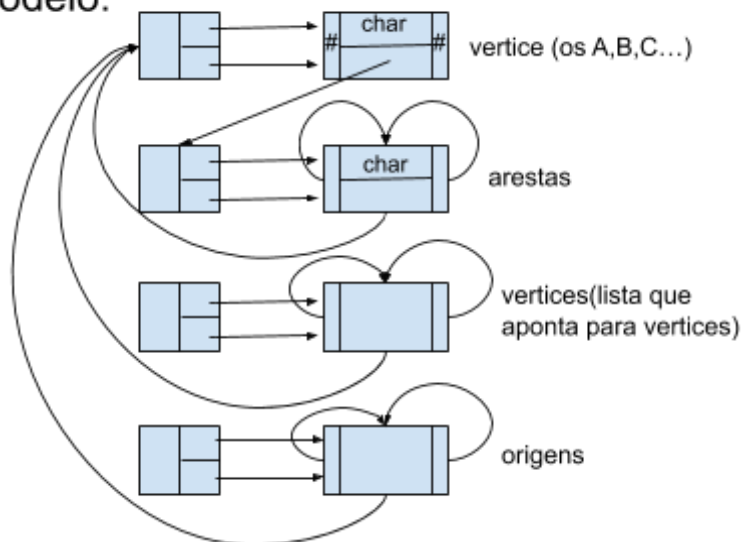
obs2: como dizer quantas dimensões tem a matriz? com assertivas

(1) a matriz comporta apenas 3 dimensões, sendo o nó da lista mais interna preenchido com um void*

f) grafo criado com listas



modelo:



Assertivas

Definição:

- **Qualidade por construção** é a qualidade aplicada a cada etapa do desenvolvimento de uma aplicação.

Assertivas são regras consideradas válidas em determinado ponto do código (tipo: a função recebe 3 parâmetros). Elas são compostas de texto em linguagem natural, mas podem ser implementadas em C por um printf ou um comando assert(ex: se não receber um int aborta).

Onde aplicar assertivas:

- Argumentação de corretude (um bloco está correto quando se pode provar que está)
- Instrumentação (implementar as assertivas [texto] transformando-as em blocos de código para que o programa se auto-verifique)
- Trechos complexos em que é grande o risco de erros.

Assertivas de entrada e saída:

Assertivas de entrada são regras que devem ser verdadeiras antes da entrada em um bloco de código. (tipo: a função recebe 3 parâmetros não nulos)

Assertivas de saída devem ser verdadeiras no final do bloco (ex: função retorna um float).

Exemplos:

- Excluir nó corrente intermediário de uma Lista duplamente encadeada.
Assertiva de entrada: ponteiro corrente referencia o nó a ser excluído e este é intermediário.
Assertiva de saída: nó referenciado foi excluído. Nó corrente aponta para o anterior.

Implementação da Programação Modular

1) Espaço de dados

São áreas de armazenamento:

- Alocadas em um meio (disco, memória, arquivo...)
- Possui um tamanho (malloc(sizeof))
- Possui um ou mais nomes de referência

Ex:

A[j] -> j-ésimo elemento do vetor A.

*(ptAux) -> espaço de dados apontado (referenciado) por ptAux

ptAux -> espaço de dados que contém um endereço

(*ObterElemTab(int id)).Id ou ObterElemTab(int id)->Id

subcampo id presente na estrutura apontada pelo retorno da função

2) Tipos de dados

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

organização → formato do inteiro na memória, tipo bits de float, int, etc (formato das coisas na memória(bits))

codificação → como entender um texto dividido em sub divisões tipo data

dd/mm/aaaa

conjunto de valores permitidos → quais os valores que podem estar em um tipo, tipo char receber letras, int inteiros etc

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

3) Tipos de tipo de dados

- Tipos computacionais: int, char, char*
- Tipos básicos/personalizados: enum, typedef, union, struct
- Tipos abstratos de dados

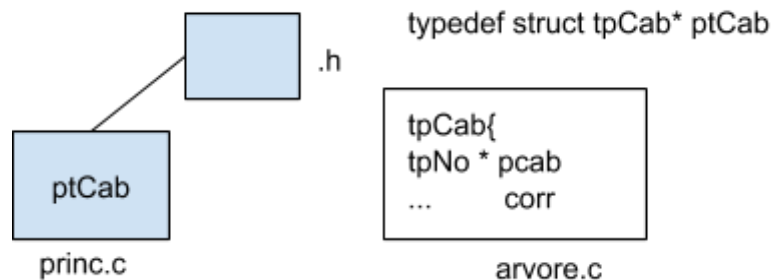
OBS: Imposição de tipos é forçar diferentes interpretações para o mesmo espaço de dados. (typecast)

OBS: Conversão de tipos não é igual a imposição de tipos. Na conversão você muda o tipo, você faz a ser igual a a mas em outro formato. Imposição de tipos mantém o binário mas muda a interpretação do binário.

4) Tipos básicos

a) Typedef

Define um novo tipo explicitamente dando um nome novo a outro tipo.

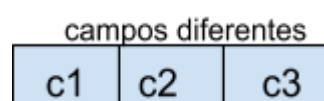


b) Enum

```
enum{
    texto1,
    texto2,
    texto3,
} tpexemplo
```

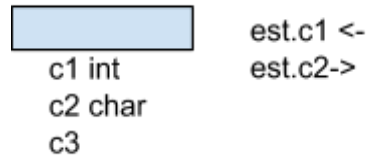
c) Struct

O tamanho do struct é igual à soma dos tamanhos dos seus campos (+padding)



d) Union

é um typecast mais organizado. Mesmo campo que pode mudar a forma de interpretação. Tipo ser char e int



O tamanho da union é o tamanho do maior campo.

5) Declaração e definição de elementos

Definir:

- Alocar espaço.
 - Amarrar o espaço a um nome (binding)
- ex: malloc é só definição

Declarar:

- Associar espaço a um tipo (ex typedef struct quando você define ainda não tem a área com espaço pra dados)

OBS: Quando o tipo é computacional ocorrem simultaneamente a declaração e definição(ex int a).

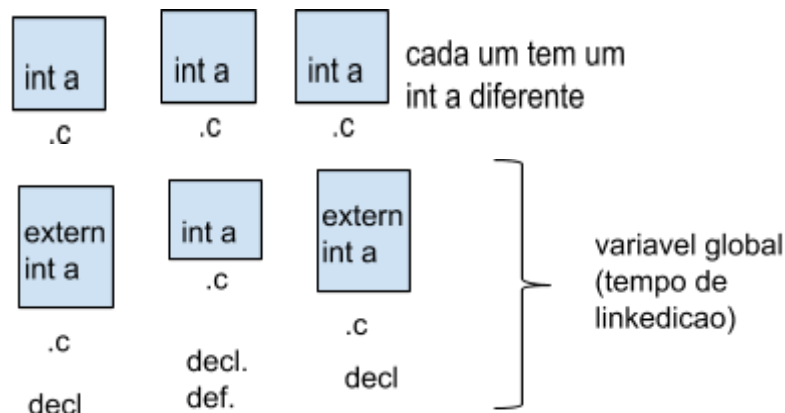
6) Implementação em C/C++

- a) Declarações e definições de nomes globais exportados pelo módulo servidor.

```
int a;  
int Func(int b); //funcao que retorna int e recebe int
```

- b) Declarações externas contidas no módulo cliente e que somente declaram o nome sem associá-lo a um espaço de dados.

```
extern int a;  
extern int Func(int b);
```



static encapsula no módulo

- c) Declarações e definições de nomes globais encapsulados no módulo

```
static int a;  
static int Func(int b);
```

7) Resolução de Nomes Externos

colocar extern em uma variável faz virar global

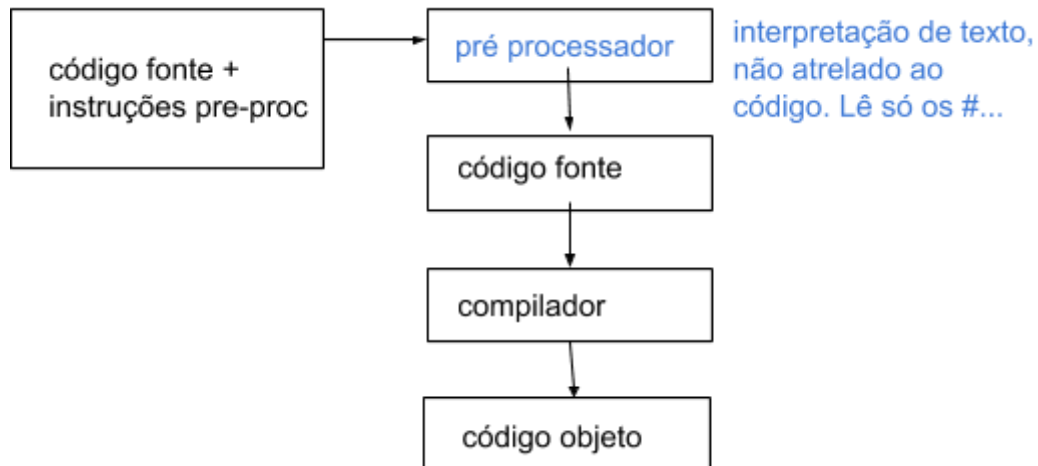
extern int a [int a]

Um nome externo somente declarado em um determinado modo necessariamente deve estar declarado e definido em algum outro módulo.

A resolução:

- Associa os declarados aos declarados e definidos.
- Ajusta endereços para os espaços de dados definidos.

8) Pré-processamento



#comando → tipo #include não é código fonte. É um pré-processador.

#define nome valor

...

#undef nome

#define nome valor → #undef nome //(1→2 sequencial)

substitui nome por valor. Substituição de texto.

nome não é mais substituído

#if defined(nome) ou #ifdef nome

textoV

#else

textoF

#endif

#if !defined(nome) ou #ifndef nome

#include <arquivo> //inclui arquivo texto

#include "arquivo"

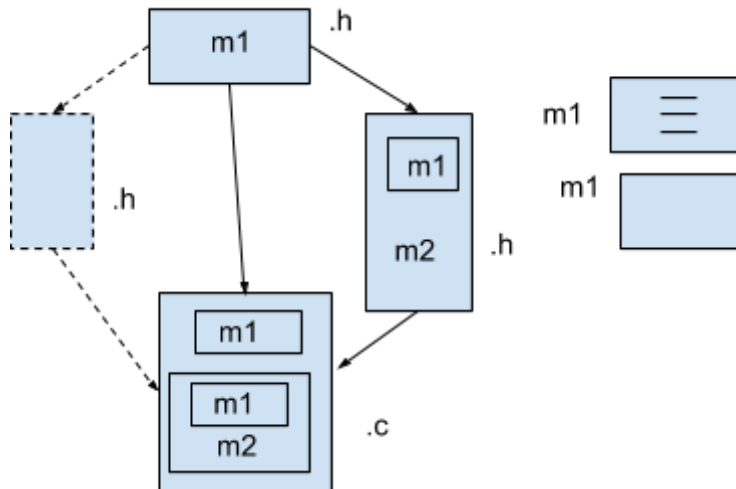
Ex:

#if !defined(EXEMP_MOD)

#define EXEMP_MOD

... (texto do .h)

#endif



```

M1.h
#if !defined(EXEMP_OWN)
    #define EXEMP_EXT extern
#endif
EXEMP_EXT int vetor[7]
#if defined(EXEMP_OWN) = {1,2,3,4,5,6,7};
#else
    ;
#endif

```

```

ou 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

```

<p>M1.c</p> <pre>#define EXEMP_OWN #include "M1.h" #undef EXEMP_OWN</pre>	<p>gera cliente</p> <pre>int vetor[7] = { 1, 2, 3, 4, 5, 6, 7 };</pre>
<p>M2.c</p> <pre>#include "M1.h"</pre>	<p>extern int vetor[7];</p> <p>servidor</p>

Estrutura de funções

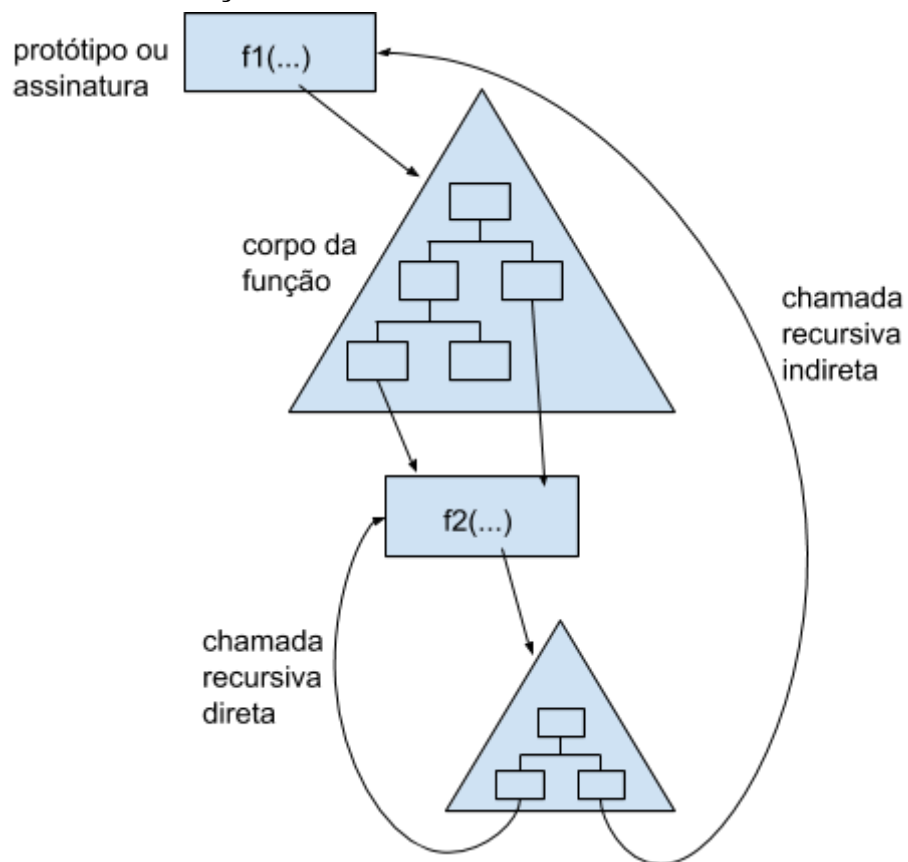
1) Paradigma

conceito de estrutura de funções

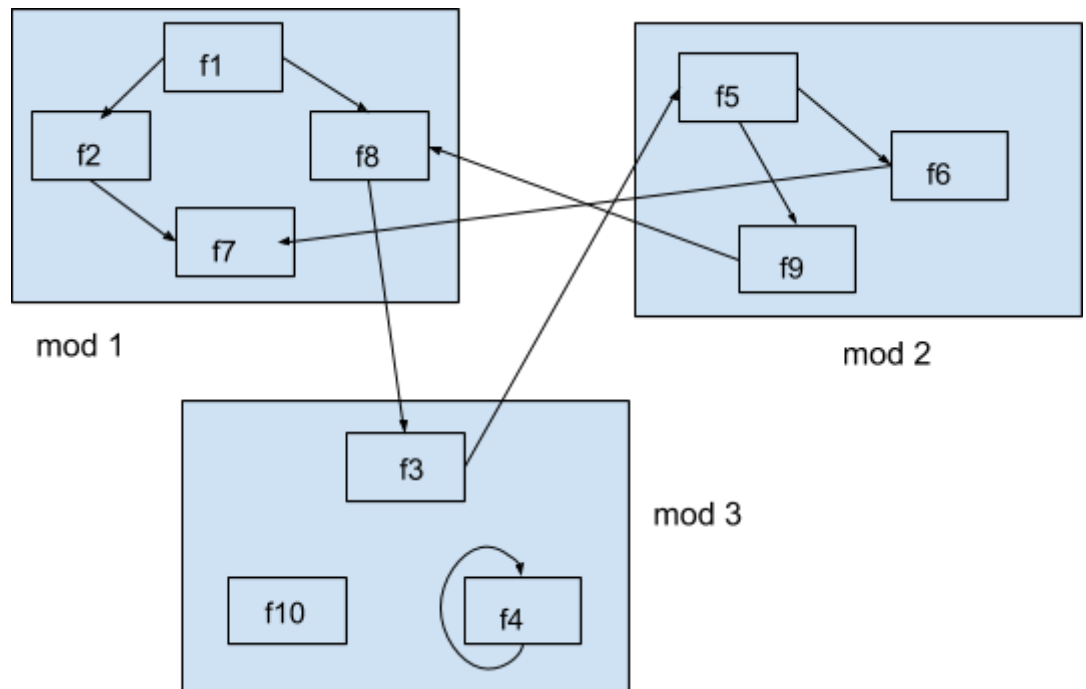
Formas de programar:

- Procedural : passo a passo, como receita de bolo
 - Programação estruturada
- Orientada a Objetos: pensar nas entidades da programação. Classes e agrupar e objetos e como guardar e como interagem
 - P.O.O.
 - Programação modular (mesmo utilizando uma linguagem não orientada a objetos)

2) Estrutura de funções



3) Estrutura de chamadas



F4→F4: Chamada recursiva direta

F9 → F8 → F3 → F5 → F9: chamada recursiva indireta

F10: função morta(função morta é p uma determinada aplicação, mas pode ser útil em outra aplicação. tipo funções de lista no grafo)

F8 → F3 → F5 → F6 → F7: dependência circular entre módulos(pq volta para o mesmo módulo)

4) Função

serve para evitar a repetição de blocos de código

def: é uma porção autocontida de código(tudo que ela precisa p achar um resultado ela tem). Possui:

- um nome
- uma assinatura
- um ou mais corpos de código

parametros de tipo ponteiro de função(tipo uma variavel mas aponta para bloco de codigo).



cada módulo especifica o ponteiro de função para um corpo de codigo mas ela recebe os mesmos parâmetros

pode usar a mesma função para varias coisas

tem que ter os mesmos parametros

5) Especificação da função

- Objetivo: objetivo da função e (n precisa descrever se o nome da função for auto explicativo) pode ser igual ao nome.
- Acoplamento: especifica parâmetros e condições de retorno.
- Condições de acoplamento: assertivas de entrada e assertivas de saída
- Interface com o usuário: mensagens, saídas em tela para o usuário
- Requisitos(assertivas intermediárias): O que precisa ser feito(tipo oq precisa para funcionar)
- Hipótese: São regras pré-definidas que assumem como válida uma determinada ação/proposição ocorrendo fora do escopo, evitando assim o desenvolvimento de códigos desnecessários. (podem ser pré-definidas, p n ter erro a toa. tipo qd alguém vai inserir cpf n deixa o campo livre, deixa so espaços p num, etc. parte-se de uma pre definição, assume-se que tal coisa seja assim)
- Restrições: São regras que limitam a escolha das alternativas de desenvolvimento para uma determinada solução(tipo linguagem a ser usada, tempo, etc)

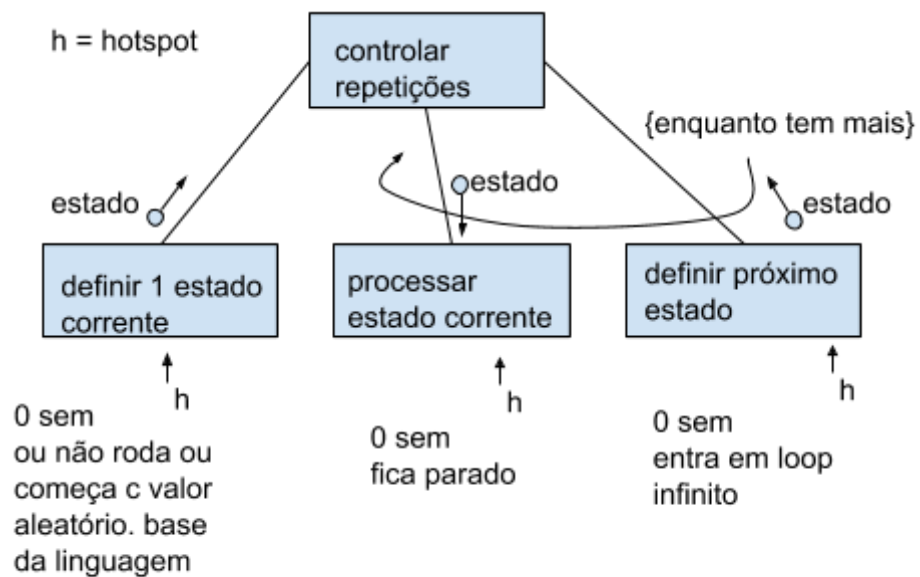
6) Interfaces

- Conceitual → definição da interface da função sem preocupação com a implementação.
inserirSimbolo(tabela, simbolo) => tabela, idSimbolo, condRet
- Física → implementação da conceitual
tpCondRet insSimb(tpSimb* simbolo)
tabela → global static no módulo
- Implícita: dados de interface diferentes de parâmetros e valores de retorno

7) house keeping

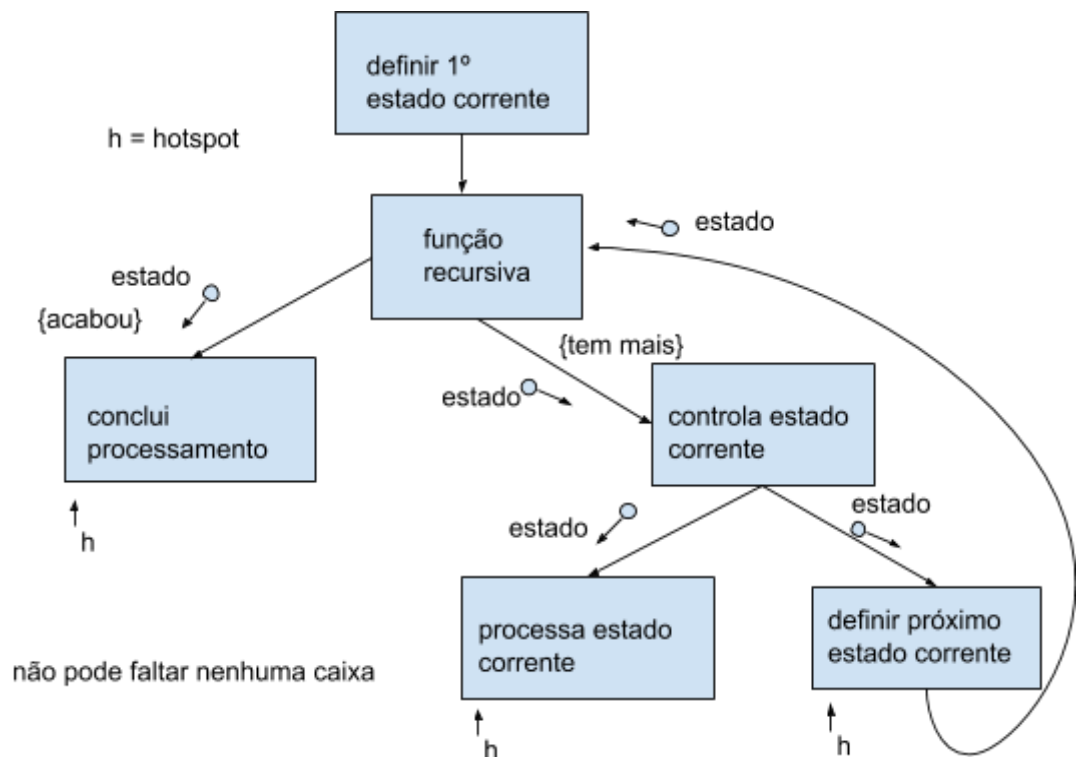
Código responsável por liberar componentes e recursos alocados a programas ou funções ao terminar a execução.

8) Repetição



não pode faltar nenhuma caixa

9) Recursão



10) Estado

passo que está na repetição. Se não muda o passo é pq a repetição deu errado. Se muda ela tá indo.

- def: Descritor de estado:(tipo índice de vetor, ini e fim de busca binária) Conjunto de dados que definem um descritor. ex. índice na pesquisa vetor

- Estado: Valoração do descritor(qd bota valor no descritor)
obs: Não é necessariamente observável, ex: cursor de posicionamento de arquivo.
obs2: não precisa ser único, ex: limite superior e limite inferior de uma pesquisa binária.

11) Esquema de algoritmo

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

parte generica + **parte especifica** = framework

Esquemas de algoritmo permitem encapsular a estrutura de dados utilizada.
 É correto, independe de estrutura e é incompleto (precisa ser instanciado).

Normalmente ocorre em:

- Programação orientada a objetos.
- Frameworks.

se esquema correto e hotspot com assertivas válidas então programa correto

12) Parâmetros do tipo ponteiro para função

```
float areaQuad(float base, float altura)
{ return base * altura}
float areaTri(float base, float altura)
{return (base*altura)/2;}
```

```
Int ProcessaArea(float valor1, float valor2, float (*Func)(float, float){
    [...]
    printf("%f", Func(valor1, valor2));
    [...]
}
```


condRet = ProcessaArea(5, 2, areaQuad);

condRet = ProcessaArea(3, 3, areaTri);

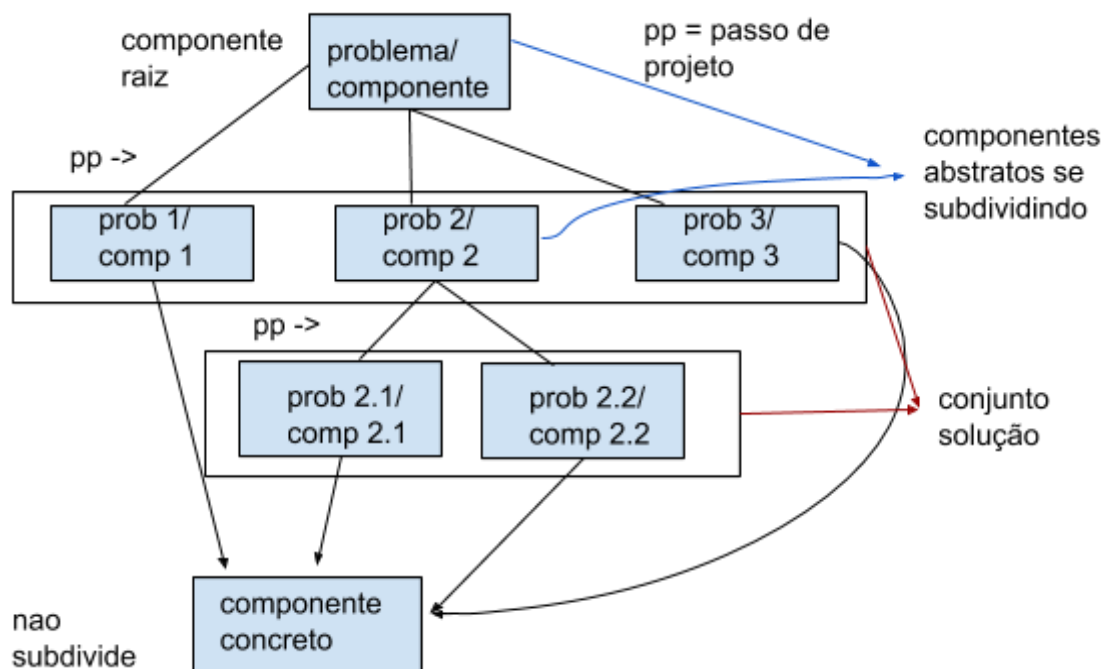
p2

Decomposição sucessiva

1) conceito

- Divisão e Conquista: dividir o problema em sub-problemas menores de forma que seja possível resolvê-los.
- A decomposição sucessiva é um método de divisão e conquista.

2) estruturas de decomposição



o conjunto solução é apenas um nível, o primeiro nível ligado a caixa

1 Estrutura -> 1 Solução

1 Solução -> n estruturas

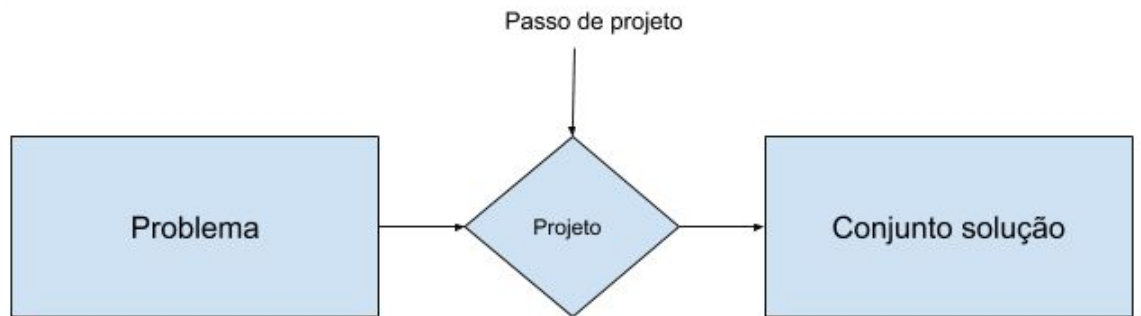
- boas
- ruins

3) critérios de qualidade

- Complexidade(difícil ou fácil de entender)
- Necessidade : (algo que se tirar não muda nada na solução do problema)
Todos os componentes de um conjunto solução são necessários?
(tipo quando tem mudança de versão. Era utilizado em versão anterior mas em atualização não é mais usado)
- Suficiência : (oposto de necessidade) Os componentes que fazem parte do conjunto solução são suficientes

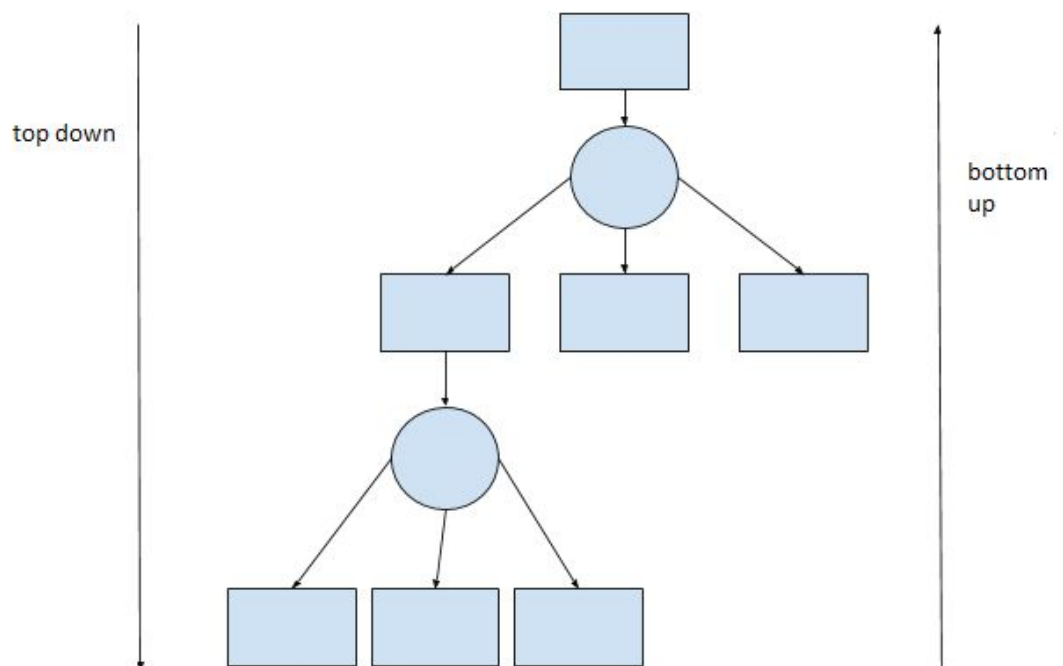
- Ortogonalidade : (dois componentes não realizam a mesma atividade) O que um componente faz, nenhum outro faz dentro de um conjunto solução

4) passo de projeto



tem 1 prob passa por um pp e chega no conjunto solução
 vulgo as linhas q ligam o quadrado da esquerda com seus filhos (conj sol)

5) direção de projeto



bottom up: começa o projeto de baixo para cima. começa pelos problemas menores.
 top down: cria fake(valores fixos para poder ir testando). pensa de cima para baixo, problema principal primeiro

Argumentação de corretude

(AE)

```
Inicio
IND <- 1
(AI1)

Enquanto IND <= LL FAÇA
    Se ELE[IND] = pesquisado.
        break
    Fim-Se
    IND <- IND + 1
Fim-Enquanto
(AI2)

Se IND <= LL
    MSG "ACHOU"
Senão
    MSG "NÃO ACHOU"
Fim-Se
Fim
```

(AS)

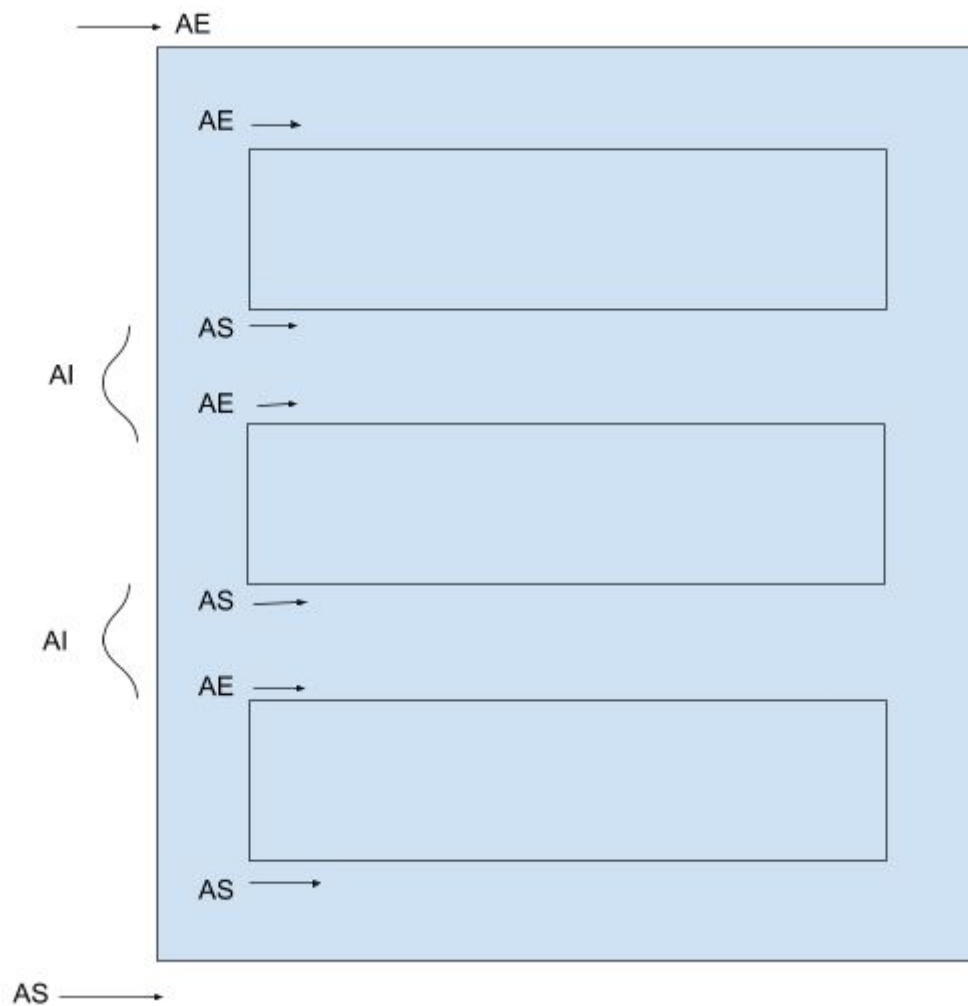
1) Definição

É um método utilizado para argumentar que um trecho de código está correto.

2) tipos de argumentação

- sequencia
- relação
- repetição

3) argumentação de sequência



limite logico de um vetor (LL): a parte preenchida de um vetor. tipo de $n[50]$ só 10 são preenchidos.

Sequência

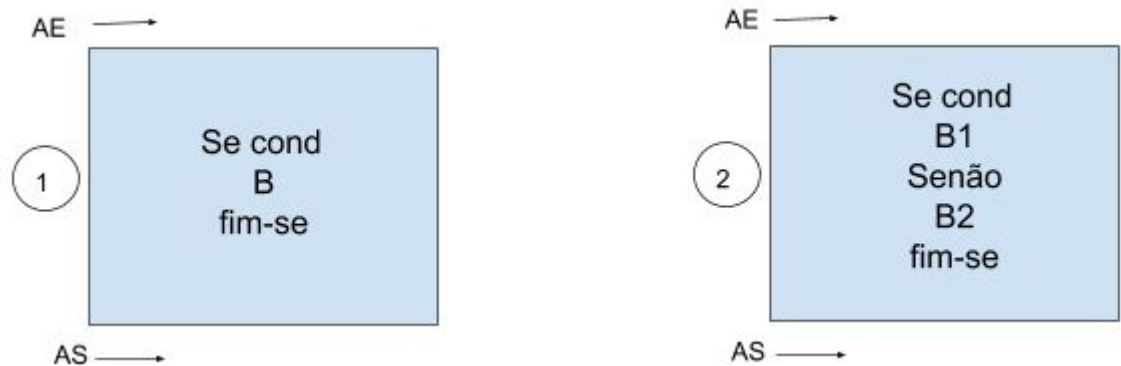
AE : Existe um vetor válido e um elemento a ser pesquisado

AS : MSG “achou” se o elemento foi encontrado (IND aponta elemento encontrado) ou MSG “nao achou” se $IND > LL$

Ai1 : IND aponta para primeira posição do vetor

Ai2 : Se o elemento foi encontrado IND aponta para o mesmo. Senão $IND > LL$

4) Argumentação de seleção



(+) = executando

(1) $AE \ \&\& \ (c == TRUE) \ (+) \ B \rightarrow AS$
 $AE \ \&\& \ (c == FALSE) \rightarrow AS$

(2) $AE \ \&\& \ (c == TRUE) \ (+) \ B1 \rightarrow AS$
 $AE \ \&\& \ (c == FALSE) \ (+) \ B2 \rightarrow AS$

Seleção (assertivas)

AE : Ai2 AS : AS principal

$AE \ \&\& \ (c == TRUE) \ (+) \ B1 \rightarrow AS$

Pela AE, se o elemento for encontrado, IND aponta para o mesmo (e é $\leq LL$)
 como ($c == TRUE$) $IND \leq LL$, B1 apresenta msg “achou” valendo AS

$AE \ \&\& \ (c == FALSE) \ (+) \ B2 \rightarrow AS$

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 AS

5) Argumentação de repetição (while e for)

AE : Ai1 AS : Ai2

AINV \rightarrow Assertiva Invariante

- Envolve os dados descritores de estado
- Válida a cada ciclo de repetição
- Existem dois conjuntos: a pesquisar e já pesquisado
- IND aponta para o elemento do conjunto a pesquisar

(1) $AE \rightarrow AINV$

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

(2) $AE \ \&\& \ (c == F) \rightarrow AS$

- 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 primeiro ciclo : Pela AE, IND aponta para o primeiro elemento do vetor. se este for igual ao pesquisado, o break é executado e IND aponta para o elemento encontrado, vale AS

(3) AE && (c == T) (+) B → AINV

- Pela AE, IND aponta para o primeiro elemento do vetor. Como (c == T), este primeiro elemento é diferente do pesquisado. Este então passa do conjunto a pesquisar para o já pesquisado e IND é reposicionado para outro elemento de a pesquisar, vale AINV

(4) AINV && (c == T) (+) B → AINV

- Para que a AINV continue valendo, B deve garantir que um elemento passe de a pesquisar para o já pesquisado e IND seja reposicionado

(5) AINV && (c == F) → AS

- Condição falsa : Pela AINV, IND ultrapassou o limite lógico e todos os elementos estão em já pesquisado. Pesquisado não foi encontrado com IND > LL, vale AS
- Ciclo não completou : Pela AINV IND aponta para elemento de a pesquisar que é igual a pesquisado. Nesse caso vale a AS pois ele[IND] = pesquisado

(6) Término

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

Sequência

AE : AINV

AS : AINV

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

Seleção

AE : AINV

AS : AI3

a) AE && (C == T) + B → AS

Pela AE (AINV), IND aponta para elemento do conjunto a pesquisar. Como (C == T) o ele[IND] é igual ao pesquisado e assim o elemento foi encontrado em IND, valendo a AS

b) AE && (C == F) → AS

Pela AE (AINV), IND aponta para um elemento do conjunto a pesquisar. Como (C == F) o elemento apontado por IND não é o pesquisado, valendo a AS.

Instrumentação

gira em torno de teste. ver se tá funcionando.

1) Problemas ao realizar testes

Esforço de Diagnose: procurar a origem de um problema(erro) de código pode demorar muito tempo e não se conseguir encontrar uma solução(quebra a cabeça para achar o problema).

- grande
- muito sujeito a 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 (sem um padrão para achar/ aparecer o erro. acontece em certos momentos e não em outros. instável)
- Causa externa ao código que mostra a falha. Ex: ponteiros "loucos", comportamento inesperado do hardware.

2) O que é instrumentação

- Fragmentos inseridos nos módulos. (testa blocos de código. código de controle p saber se o programa esta funcionando)
 - código
 - dados
- Não contribui para o objetivo do programa (ex: printf para debugar)
- Monitora o serviço enquanto o mesmo é executado
- Consome recursos de execução
- Custa para ser desenvolvido

3) Objetivos

- Detectar falhas de funcionamento do programa o mais cedo possível de forma automática
- Impedir que falhas se propaguem
- Medir propriedades dinâmicas do programa (tempo de execução)

4) conceitos

- Programa Robusto: Intercepta a execução quando observa um problema. Mantém o dano confinado (marca o inicio e o fim de onde deve estar o problema, qt menor a distância entre eles mais chance de achar o erro)
- programa tolerante a falhas:
 - É robusto
 - Possui mecanismos de recuperação (corrige seu próprio erro)
- Deterioração controlada: (funciona o app mesmo com seus recursos não funcionando)

- Habilidade de continuar operando corretamente mesmo com a perda de funcionalidades

5) Esquema de inclusão de instrumentos no código em C e C++

```
#ifdef _DEBUG
```

- Código da instrumentação.
- Dados.

```
#endif.
```

- Debug: ativa (compilação da versão debug. código de controle só em versão debug)
- Release: desativa (em versão release não pode ter)

6) Assertivas executáveis

Assertivas -> Código (pega as assertivas e transforma em código)

Vantagens:

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

Precisam ser: (se não for correta não vai testar certo)

- Completas (gasta tempo pra fazer)
- Corretas (gasta tempo pra fazer)

7) Deturpadores (debugger)

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 utilizado apenas como último recurso.

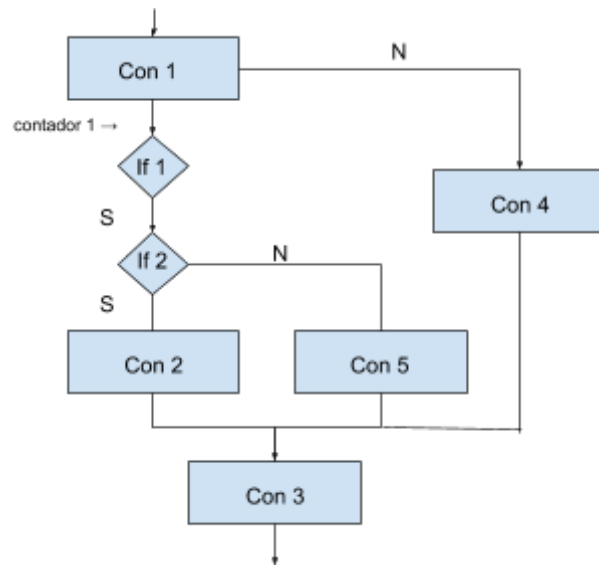
pq demora mt tempo para achar o lugar certo para por breakpoint sendo demorado

8) Trace (rastrear)

- Instrumento utilizado para apresentar uma mensagem no momento em que é executado (printf)
- 2 tipos:
 - 1) Trace de instrução: printf
 - 2) Trace de evolução: apresenta mensagem apenas se o conteúdo de uma variável for alterado

9) controlador de cobertura

- teste caixa : fechada / preta
vc nao sabe oq esta dentro mas sabe como usar tipo saber interface. tem q saber do negócio
- teste caixa: aberta/branca
vc sabe oq ta no cod. testa condições de retorno, funcionalidades e todos os caminhos possíveis



inicio

con1

// cnt_contar("contador1") //modulo conta do arcabouço

if1 cond

if2 cond

→

con2

else

→

con5

fim-if2

else

→

con4

fim-if1

con3

fim

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

vetor de contadores

label	qtd
contador1	1

vetor de contadores

label	qtd
if1 não	0
if2 sim	6
if2 não	7

tem que achar o menor numero de contadores necessarios para ver onde o caminho passa, no anterior eh : desenho → lugares com contador
entre if1 e if2 n precisa pois para chegar no com5 ou com2 ele obrigatoriamente passa por if2

se um teste que teve não passou por algum caminho o teste ta incompleto
tipo como o if1 nao passa nenhuma vez o teste nunca testou ele e está incompleto

10) verificador estrutural

(entrada, saida e intermediaria)

assertiva → codigo → assertiva executavel

assertivas estruturais + modelo → codigo = verificador

aquela estrutura de cabeça pra no (pcoor -> <- anterior bla bla bla)

instrumento responsavel por realizar uma verificação completa da estrutura em questão. é a implementação de codigo relacionado com as assertivas estruturais e modelo.

como faz pra testar um verificador:

dar as estruturas erradas pra ele entretanto vc ta fazendo coisas do codigo ao mesmo tempo do verificador

11) deturpador estrutural

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

deturpa(tipoDet) → é uma função

obs:a deturpação é sempre realizada no nó corrente.

o numero de deturpadores é maior ou igual ao numero de de verificadores

=criar estrutura

=deturpa 1

=verifica

=deturpa 2

-----> gravar contador anterior

=verifica → voou(abend)

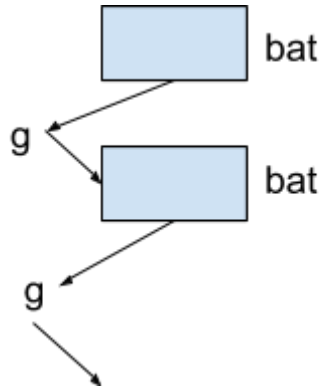
=deturpa 3

...

=estatística de contadores (por um contador em cada verificador pq ai ve tudo

)

como vc vai deturpando pode voar, então ou escolhe blocos de código pra testar, para antes do q der erro e refaz. ou da p fz o bag todo no msm script.



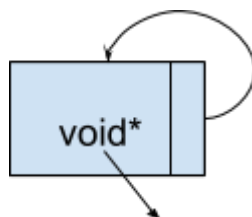
12) recuperador estrutural

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

13) estrutura auto verificável

tudo dentro de uma estrutura pode ser consertado por ela msm

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

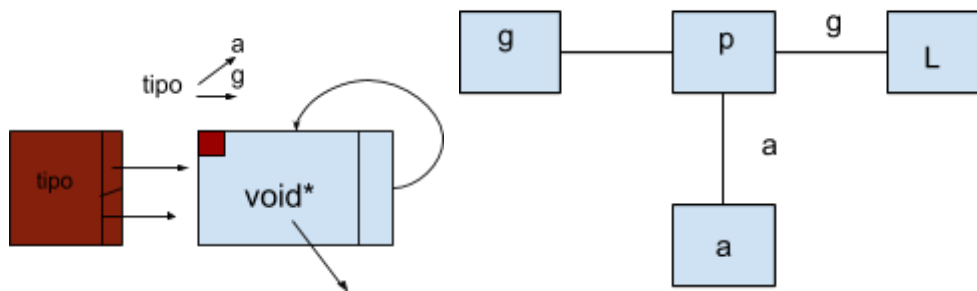


caso 1: é possível definir todos os tipos apontados pela estrutura?

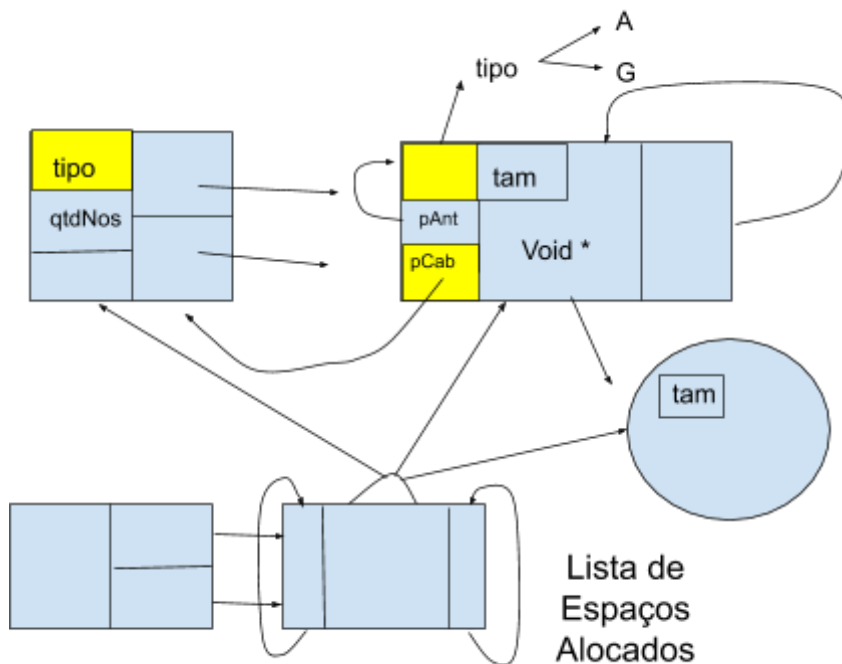
nao, precisa dizer oq pode ser em cada campo eu acho

resposta: tem que incluir um campo tipo(no modelo precisa ter outra cor) em cada um dos nós da estrutura e no cabeça

resp2: inclusao de campo p anterior no nó e de campo p cabeça tb no nó



tem que dizer oq é cada no
 insere(pt, valor #ifdef _DEBUG , tipo #endif)



é possível definir o tamanho total da estrutura?

inclusão de um campo qtd de nós na cabeça e campo tamanho no nó e na estrutura apontada

é possível definir se todos os espaços alocados estão ativos? Utilizando uma estrutura chamada Lista de espaço alocado (LEA)

L.E.A: Lista de Espaço Alocado é uma lista encadeada cujos nós apontam para áreas de memória alocadas para o programa. Para verificar se existe vazamento de memória, podemos percorrer, a L.E.A., ignorando os ponteiros que sabemos ativos da aplicação, e verificar se algum ponteiro desta lista está inativo. Neste caso podemos também excluir a área de memória inativa.

OBS: Vazamento de memória (memory leak) ocorre quando existe um espaço alocado e inativo(a cada malloc dado)

Teste de software

1) definição

São técnicas de controle de qualidade baseadas na execução de experimentos controlados.

(testes para chegar no resultado certo: resultado que o cliente espera. Prov n chega de antemão)

Antes de realizar os testes: (preparação/escolha de parametros do que quer testar)

- Critério de seleção de casos de teste
- Cenários de testes (tudo que precisa pro teste funcionar):
 - modo de uso de artefato
 - Organização necessária
 - Ferramentas (as vezes cria um banco de dados usado pra teste)
 - Definição e preenchimento do banco de dados
 - Massa de teste (todos os testes necessarios p testar tipo o script).Ex: script de teste
 - Resultados esperados
 - Pessoas envolvidas no teste

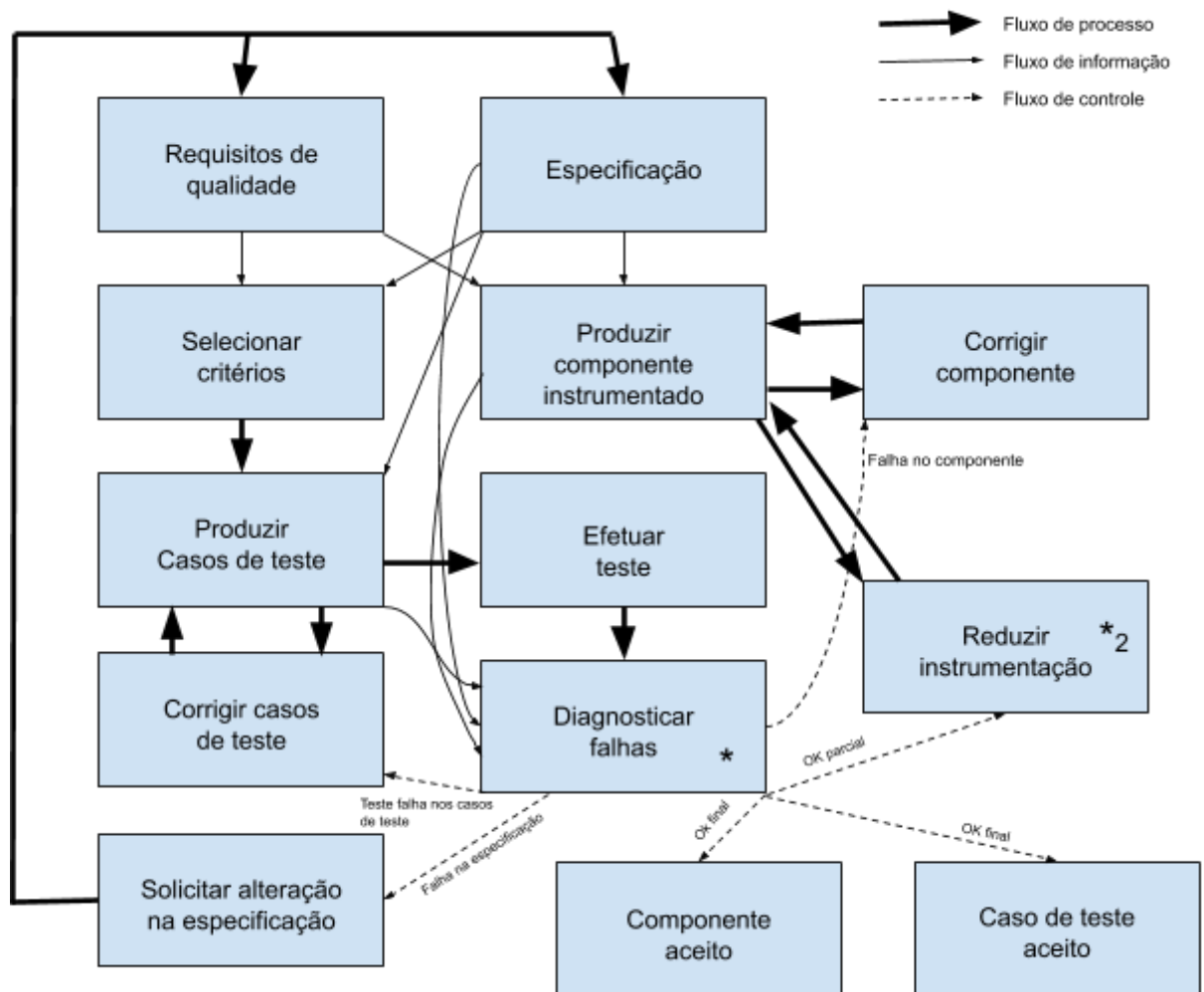
OBS: Testes são utilizados para detectar a presença de erros, nunca a ausência de erros (Testes orientados à destruição). Não é o objetivo ver se funciona.

Teste x Objetivo:

Teste mais rigoroso: serviços com elevado valor e grandes riscos. (tipo qd o risco é mt alto, tipo programas financeiros de banco)

Teste menos rigoroso: Testes com menor valor e riscos pequenos

2) processo de teste de programa



* → casos: código errado, especificação errada e teste errado

*2 → tipo tirar os printf de debug.

começa a pensar em teste na especificação de requisitos

req de qualidade: mais próximo do que o cliente quer

3) registro de falhas

fazer documentação de todas as falhas que acontecem e suas soluções

É uma planilha. Segue abaixo uma lista de sugestões de campos:

- Data.
- Identificação do teste.
- Sintoma (o que está acontecendo no pc)
- Data de correção.
- Artefatos alterados.
- Classe da falta/ conjunto da falta. (tipo de falta. tipo: rede, documentação, código, teste)
- Correção realizada.

Exemplos de classe: especificação, projeto, código, rede, plataforma, outros, etc...

falta = problema

falha = aparece o problema

4) Critérios de seleção de casos de teste

- Teste caixa fechada
-> Casos de teste gerados a partir das especificações
- Teste caixa aberta
.-> Casos de teste gerados a partir da estrutura do código
- Teste de estrutura de dados
-> Casos de teste gerados a partir dos modelos das estruturas de dados utilizadas

Critério é :

- Válido : Acusa falha quando existe falta no artefato (diz q tem erro qd há erro)
- Confiável : Acusa falha que independe da escolha de dados e ações (não mascarar erros)
- Completo : Cobre todas as as condições definidas por um padrão de completeza/ completude
- Eficaz : Quanto mais falta encontrar
- Eficiente : Quanto menos recurso gastar

5) Processo de geração de caso de teste

1 analisa oq precisa ser testado

- Caso de teste abstrato
. EX : A repetição deve ocorrer 3 vezes (testar oq precisa ser testado)
. O que deve ser testado

2 como vou testar que a rep ocorreu 3 vezes?

- Caso de teste semântico
. EX : Para que a repetição ocorra 3 vezes, é preciso que o arquivo possua 3 registros
. Como deve ser testado (como testar)
- Caso de teste valorado / Caso de teste (por valor no caso de teste semântico)
. EX : Inserir 3 registros válidos no arquivo e executar o teste para confirmar que as repetições estejam ocorrendo corretamente
. Colocar valor no caso de teste semântico

6) Recomendações básicas para valoração de caso de teste

testar nos limites que mudam de um teste para outro para aparecer mais erros. limite smp tem mais erros. limites e extremos.

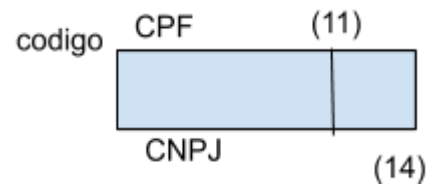
A) Testar $a \geq b$

$A > B$	$A = B$	$A < B$
$A = B + 1$	$A = B$	$A = B - 1$

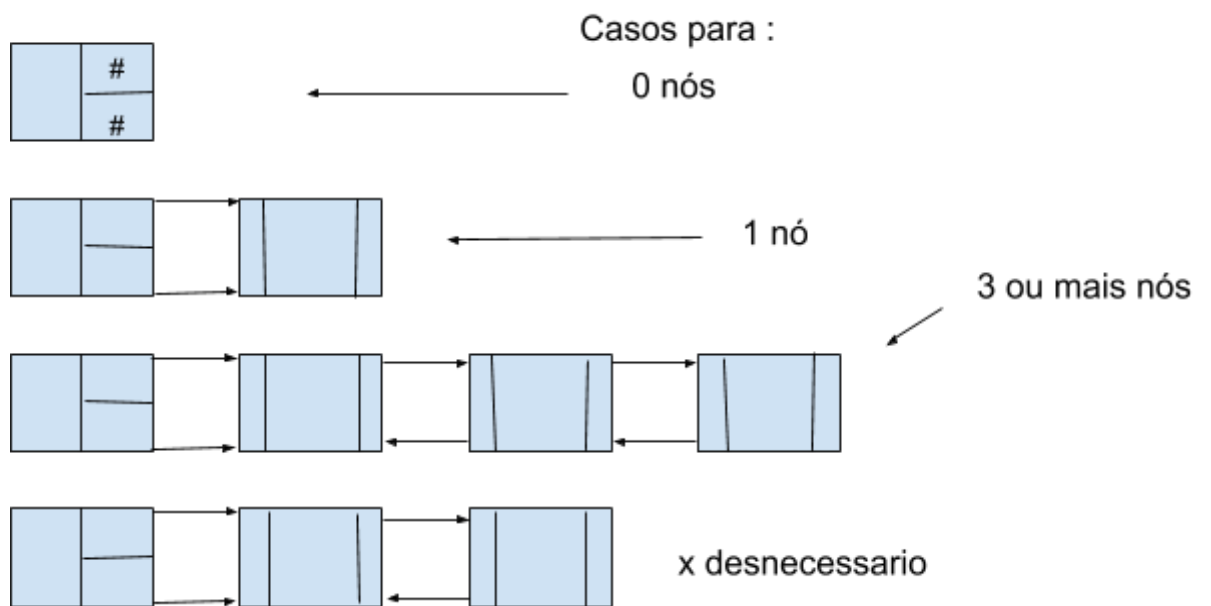
OBS : O objetivo é maximizar a probabilidade de encontrar problemas. É no limite que ocorrem mais problemas
precisa de 3 testes p ver se n tem sequela no código

B) Valores de entrada de tamanhos variados

- . tamNulo
- . tamMin
- . tamMax
- . tamMed
- . tamMin - 1
- . tamMax + 1
- . caracteresValidos

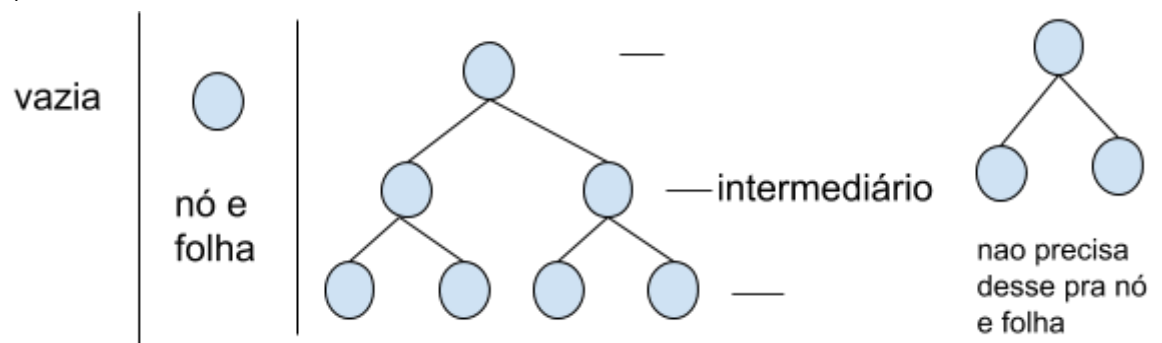


C) Valores pertencentes a estruturas criadas dinamicamente



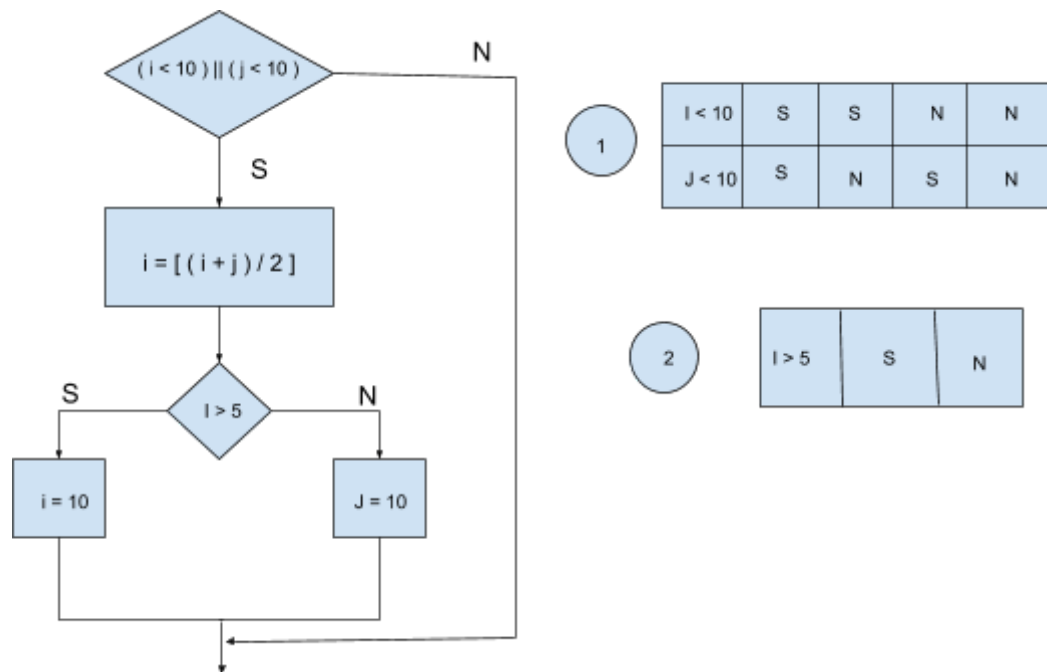
3 ou + nós -> inicio, fim, meio, não encontra <- testar esses casos

D) Árvore



7) Critérios de cobertura

- Cobertura de decisões



caso	1	2	3	4	5	6	7	8
i	4	1	2	1	10	10	10	
j	9	9	10	10	2	1	10	
(1)	SS	SS	SN	SN	NS	NS	NN	NN
(2)	S	N	S	N	S	N		

descarta a 8 pq o 7 já tem o NN

i e j -> valores escolhidos são arbitrários que testem todos os casos

- cobertura de arestas

(1)	$(i < 10) \parallel (j < 10)$	S	N
-------	-------------------------------	---	---

(2)	$i > 5$	S	N
-------	---------	---	---

Casos	1	2	3	4
i	4	1	10	
j	9	10	10	
(1)	S	S	N	N
(2)	S	N		

descarta a 4

- Cobertura de Comandos

(1)	(i < 10) (j < 10)	S	
-------	--------------------------	---	--

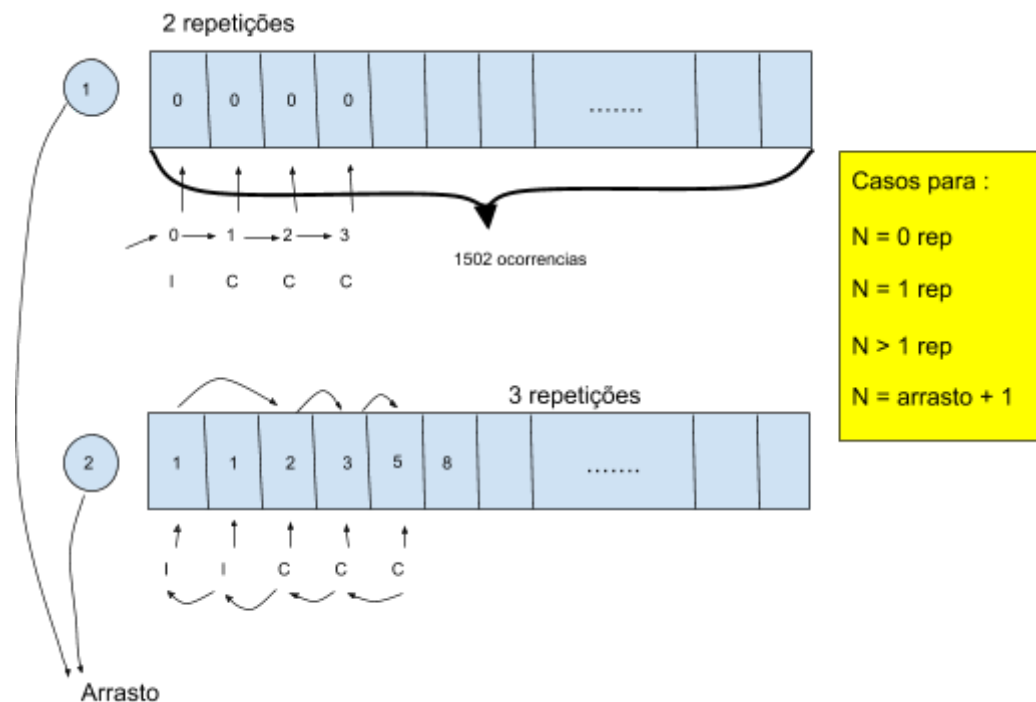
(1)	(i < 10) (j < 10)	S	N
-------	--------------------------	---	---

Casos	1	2
i	4	1
j	9	10
(1)	S	S
(2)	S	N

só usa o 1(cob decisões) qd tem uma expressão mt complexa no if
condição composta muito complexa, qd tem que ver se esta avaliando logicamente
da forma esperada. a diferença é que vc n testa na mão

8) Cobertura de repetições

- Qual o menor número de testes necessários para saber se uma repetição está funcionando



o 0 é inicializado

1 repetição para q a proxima seja calculada apenas

- Arrasto é o menor número de iterações necessárias/repetições para que a próxima iteração utilize apenas valores calculados em iterações anteriores de forma a progredir corretamente a repetição a ser testada.

9) teste caixa fechada

- Método : Partição em classes de equivalência
- Classe de equivalência : Cada caso exercita em uma situação que não é verificada em qualquer outro caso de teste. É um conjunto de todos os casos de teste que possuem um mesmo objetivo

pesquisa em vetor:

- Passo 1 :

Tipos estruturados	Resultado esperado
Vetor vazio	Achou
Vetor 1 elemento	Não achou
Vetor 3 elementos	

- Passo 2 :

Caso	Vetor	Valor	Achou	N~Achou
1	0	A		X
2	A	A	X	
3	A	B		X
4	ABC	A	X	
5	ABC	C	X	
6	ABC	B	X	
7	ABC	D		X

- Passo 3 :
 == caso 1
 =criar 0<- ok
 =pesq A 1<- nao achou

== caso 2
 =inserir A 0<- ok
 =pesq A 0<- achou

== caso 3
=pesq B 1

== caso 4
=inserir B 0
=inserir C 0
=pesq A 0

== caso 5
=pesq C 0

== caso 6
=pesq B 0

== caso 7
=pesq D 1

-

10) testes estatísticos

testes gerados de forma aleatoria e tem os testes de cobertura

- É o módulo responsável por gerar casos de teste com dados de forma aleatória garantindo uma total automação.
- Vantagem : Geração automática dos casos de teste
- Desvantagem : Inclusão de casos de teste repetidos

OBS : É possível verificar a completude do teste utilizando controladores de cobertura

Qualidade de artefatos

1) objetivo

elevada qualidade x qualidade satisfatória

qualidade satisfatória : qualidade q deixa o cliente satisfeito

qualidade > quando o cliente recebe oq ele queria, mais prox do q o cliente quer elevada qualidade é por um controle de qualidade qualquer(tipo coisa top de tecnologia) e não necessariamente oq o cliente quer

a ideia é vc tentar fazer o melhor q vc consegue desde o inicio -> qualidade por construção

2) qualidade por construção

atingida em virtude do trabalho disciplinado:

- padrões
- ferramentas adequadas
- saber onde eu quero chegar: qualidade que satisfaça o usuario -> qualidade requerida
- identificar faltas o mais cedo possível
- vantagem: reduz custo de retrabalho (da qualidade por construção)

3) qualidade vs controle de qualidade

controle de qualidade segura a qualidade requerida ou não? não, o controle de qualidade pode ser mal feito.

- qualidade não pode ser assegurada através de um controle de qualidade
- oq é controle de qualidade? controla oq o artefato tem mas não tem como indicar oq precisa fazer.
- controle -> indicador do grau de qualidade que o artefato possui e não do que deveria ser. Ex: teste

4) propriedades do grau de qualidade

- nem todas as propriedades relevantes são mensuráveis. Ex: facilidade de uso
- fatores básicos não podem ser medidos com precisão. Ex: confiabilidade e segurança

5) tipos de qualidade

- qualidade requerida: especificada para o artefato(oq o cliente quer)
- qualidade assegurada: técnicas, ferramentas, métodos e processos conseguem assegurar por construção (tipo usar o arcabouço de teste pra assegurar um controle de qualidade. ai dps vc tem que entender o resultado do arcabouço-> qualidade observada)
- qualidade observada: observado pelo controle de qualidade
- qualidade efetiva: é a que o artefato efetivamente possui. A mais baixa.

6) garantia de qualidade

assegurar que o processo de desenvolvimento geralmente conduza a artefatos de qualidade satisfatória

