Introdução

Vantagens da Programação Modular

- Vencer barreiras de complexidade
- Facilita o trabalho em grupo (paralelismo)
  - Pessoas podem trabalhar em partes diferentes do programa, ao mesmo tempo
- Reuso, não é necessário fazer a exata mesma coisa várias vezes
- Facilita a criação de um acervo, para reuso
- Desenvolvimento incremental
- Aprimoramento individual
  - É possível melhorar os módulos individualmente, não é necessária a recompilação de tudo a cada mudança.
- Facilita a administração de *baselines*
  - Controlar mudanças, versões e "checkpoints" durante o desenvolvimento
  - "Baseline": o que se congela (versões de módulos) para gerar uma versão de build

Princípios de Modularidade

1) Módulo

Definição Física: Unidade de compilação independente
Definição Lógica: Trata de um único conceito

- Se trata de muitas coisas diferentes, dificulta a manutenção. A chance de você modificar um conceito e "quebrar" outro é maior.

2) Abstração de Sistema

Abstrair: Processo de considerar apenas o que é necessário numa situação e descartar com segurança o que não é necessário. Por exemplo, para construir um formulário de alunos, pode ser necessário ter campos de matricula, nome, nascimento… mas não tamanho do sapato.
O resultado de uma abstração é um escopo: limite do que precisa e do que não precisa.

Nível de Abstração:
Sistema → Programa → Módulos → Funções → Blocos de Código → Linhas de Código

Obs: conceitos

- **Artefato:** é um item com identidade própria criado dentro de um processo de desenvolvimento. É algo que pode ser versionado.
    - Qualquer coisa que se crie no processo de desenvolvimento e possa ser versionado. Por exemplo, uma função não é versionada, então não é um artefato, mas o código ao qual ela percebe é.
- **Construto (build):** um resultado apresentável, um artefato que pode ser executado, **mesmo que incompleto**.

3) Interface

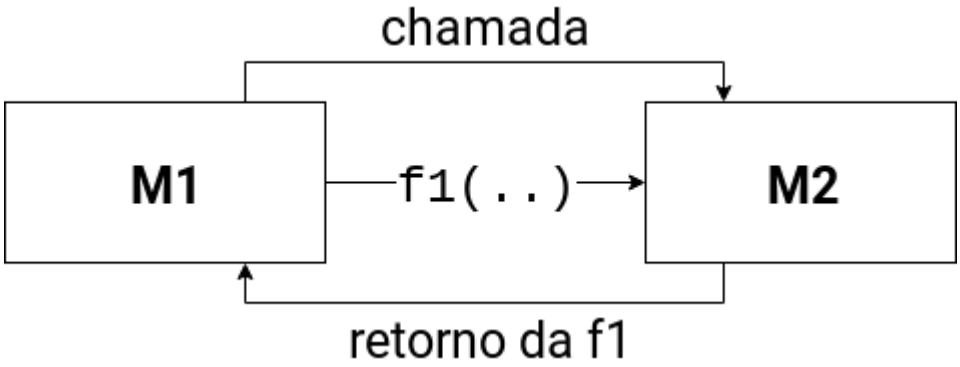
Mecanismo de troca de *dados, estados e eventos* entre elementos de um **mesmo nível de abstração**.

- Interfaces de sistema com sistema, entre módulos…

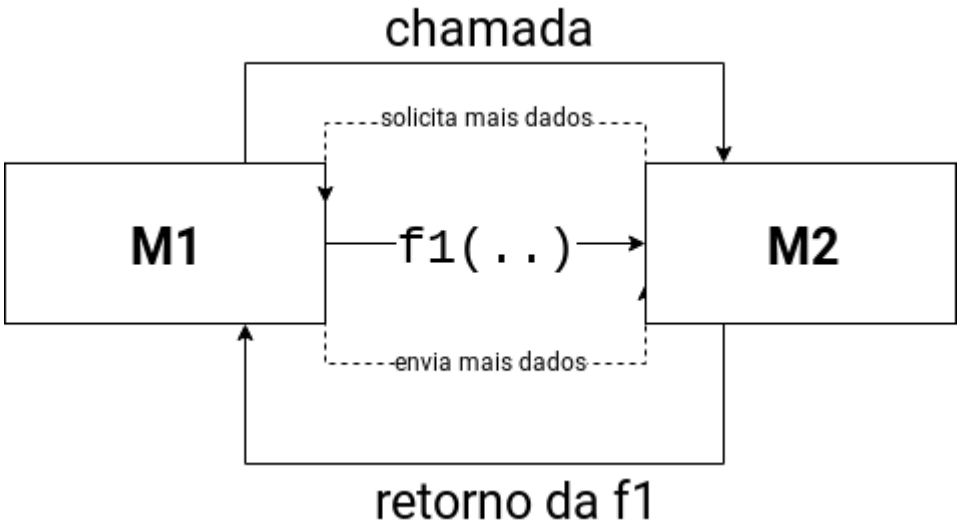
a) Exemplos de Interface

- Arquivo (entre sistemas)
- Funções de acesso (entre módulos)
- Passagem de parâmetros (entre funções)
- Variáveis globais (entre blocos)

b) Relacionamento Cliente - Servidor

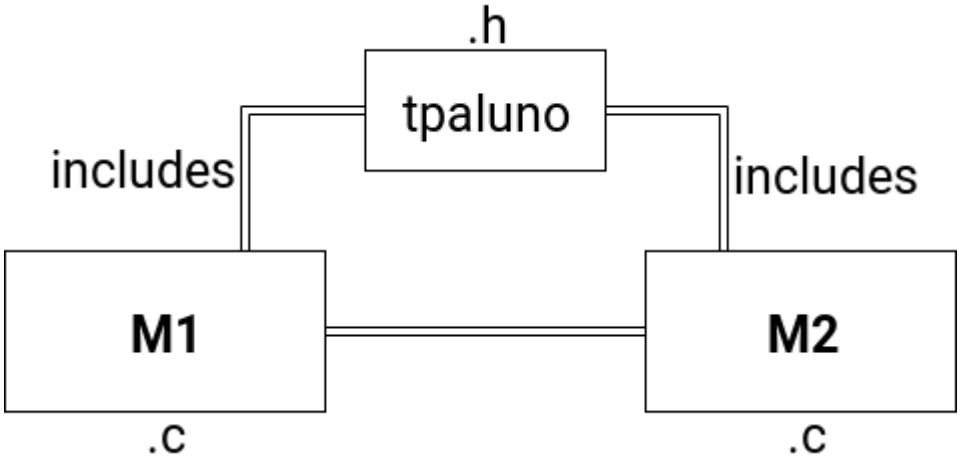


Caso especial: callback



c) Interface Fornecida por Terceiros

Depende de um terceiro componente para fazer dois módulos conversarem.
Se dois módulos usam a mesma estrutura *`tpaluno`*, não se define duas vezes essa estrutura, mas um em um terceiro componente.



d) Interface em Detalhe

- **Sintaxe**: regras
  `MOD 1 (int) <----> MOD2 (float)`
  Geralmente não funciona (sob o ponto de vista de qualidade de software)

- **Semântica**: significado
  `MOD 1 (int) <----> MOD2 (int)`

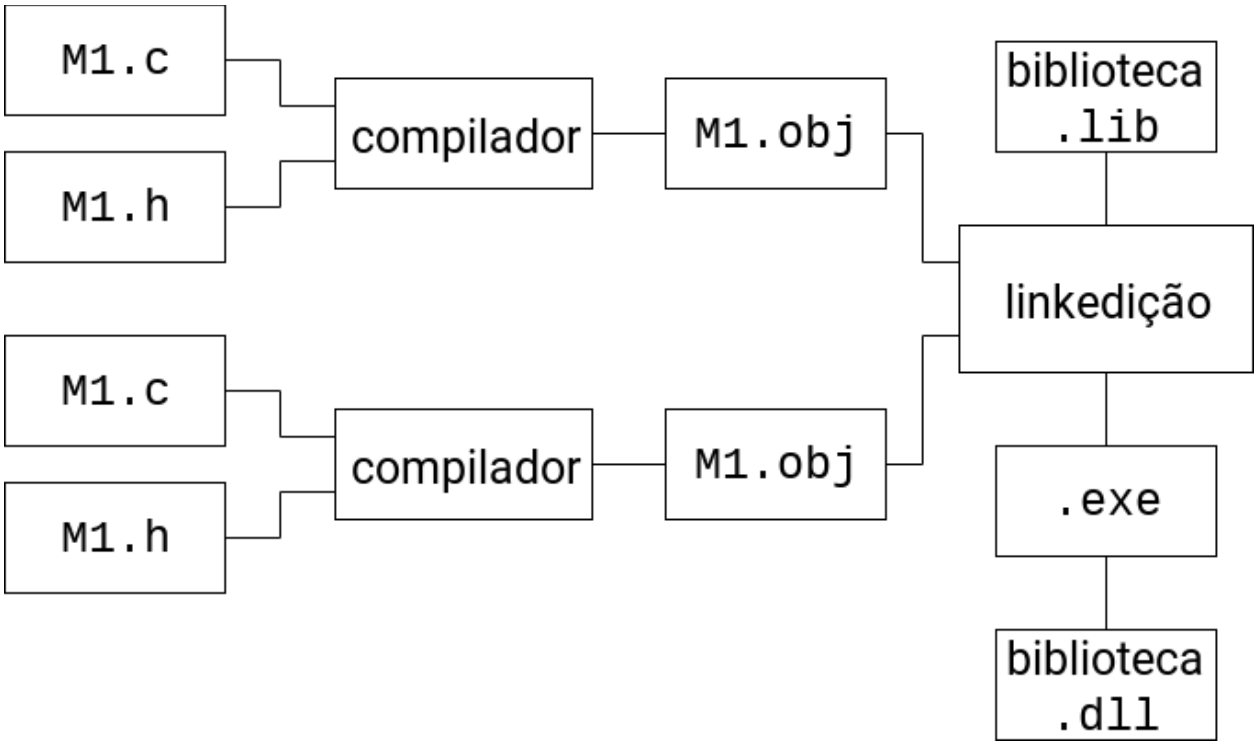
Depende, se um `int` for *idade* e o outro *quantidade de filhos*, há algum problema.

e) Análise de Interface

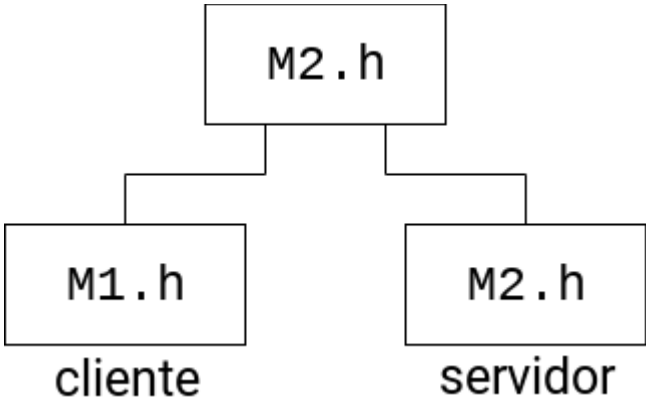
`tpDadosAluno * opterDadosAluno(int mat); // protótipo ou assinatura de função de acesso`

- Interface esperada pelo cliente: um ponteiro para dados válidos do aluno correto ou `NULL`
- Interface esperada pelo servidor: inteiro válido representando a matrícula do aluno.
- Interface esperada por ambos: `tpDadosAluno` (`int` já é conhecido)

4) Processo de Desenvolvimento



`.c` → Módulo de Implementação
`.h` → Módulo de Definição
`.lib` → Biblioteca Estática
`.dll` → Biblioteca Dinâmica



M1 usa as funções de M2. Assim, M1 é o cliente e M2 o servidor. `M2.h` é a interface entre os módulos. (`#include "M2.h"` no M1)

5) Bibliotecas Estáticas e Dinâmicas

Estática

- Vantagens
  - lib já é acoplada em tempo de linkedição à aplicação executável.
- Desvantagens
  - existe uma cópia desta biblioteca estática para cada executável alocado na memória que a utiliza.
  - aumenta o tamanho do programa.

Dinâmica

- Desvantagens
  - a `dll` precisa estar na máquina para a aplicação funcionar.
- Vantagens
  - só é carrecaga uma instância de biblioteca dinâmica na memória, mesmo que várias aplicações a acessem.

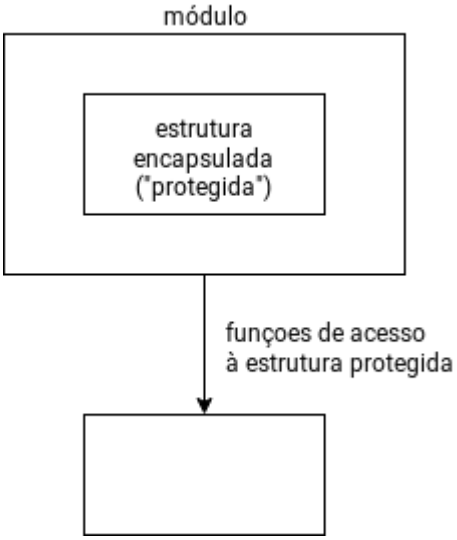
6) Módulo de Definição (`.h`)

- Interface do módulo
- Contém os protótipos das funções de acesso, interfaces fornecidas por terceiros (ex `tpDadosAluno` do item 3e)
- Documentação voltada para o programador do módulo cliente

7) Módulo de Implementação (`.c`)

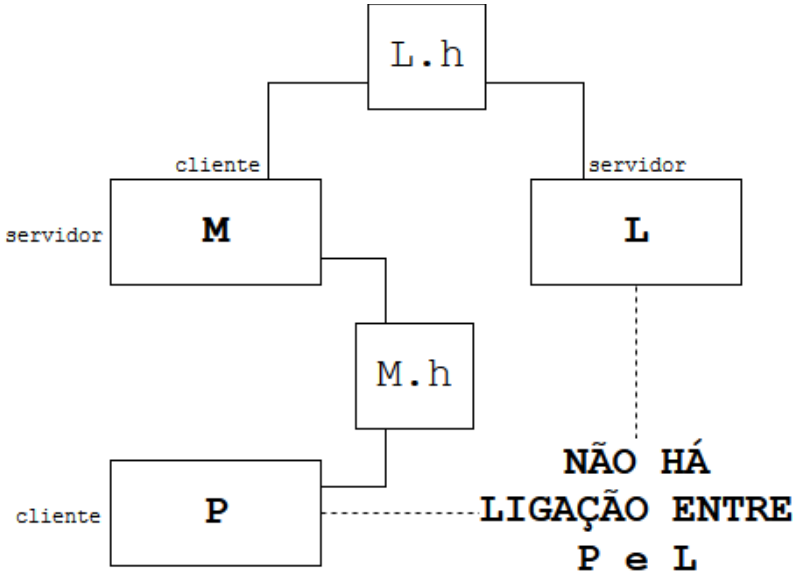
- Código das funções de acesso
- Códigos e protótipos das funções internas
- Variáveis internas ao módulo
- Documentação voltada para o programador do módulo servidor

8) Tipo Abstrato de Dados



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

Por exemplo, se um módulo que manipula uma matriz usa listas para fazer suas colunas e linhas, o esquema abaixo seria viável:



Se o cliente precisar usar mais de uma instância da estrutura fornecida pelo TAD, uma solução é trabalhar com ponteiros para a “cabeça” da estrutura. Para isso, as funções de acesso devem receber esse ponteiro que indica qual estrutura está sendo modificada, e a interface deve disponibilizar o typedef correspondente a ele com `typedef struct tpTipo * ptTipo`.

9) Encapsulamento

Propriedade relacionada com a proteção dos elementos que compõe o módulo.

Objetivo:

- Facilitar a manutenção
  - Mantém erros confinados
- Impedir utilização ou modificação indevida da estrutura do módulo

Outros tipos de encapsulamento

- de documentação
  - documentação interna: módulo de implementação `.c`
  - documentação externa: módulo de definição `.h`
  - documentação de uso: manual do usuário `README`
- 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, public, global, global static, protected, static...`
    - `private`: encapsulado no objeto
    - `static`: encapsulado no módulo (ou na classe no caso de orientação a objetos)
    - local: num bloco de código

10) Acoplamento

Propriedade relacionada com a interface entre os módulos.

Conector: item de interface

- Função de acesso
- Variável global

Critérios de Qualidade

- Quantidade de conectores
  - necessidade x suficiência
    - tudo é útil?
    - falta algo?
- Tamanho do conector
  - quantidade de parâmetros de uma função
- Complexidade do conector
  - explicação em documentação
  - utilização de mnemônios
    - nomes descritivos nas variáveis...

11) Coesão

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

- Níveis de coesão
  - incidental – pior coesão
    - praticamente não há relação
    - aplicação que faz cálculos, insere em árvore e printa “Hello World”
  - lógica – elementos logicamente relacionados
    - calcula, processa, exibe
  - temporal – itens que funcionam em um mesmo período de tempo
  - procedural – itens em sequência
    - .bat
  - funcional
    - inclui dados, gera relatório...
  - abstração de dados – a melhor coesão, trata de um único conceito
    - TAD

Teste Automatizado

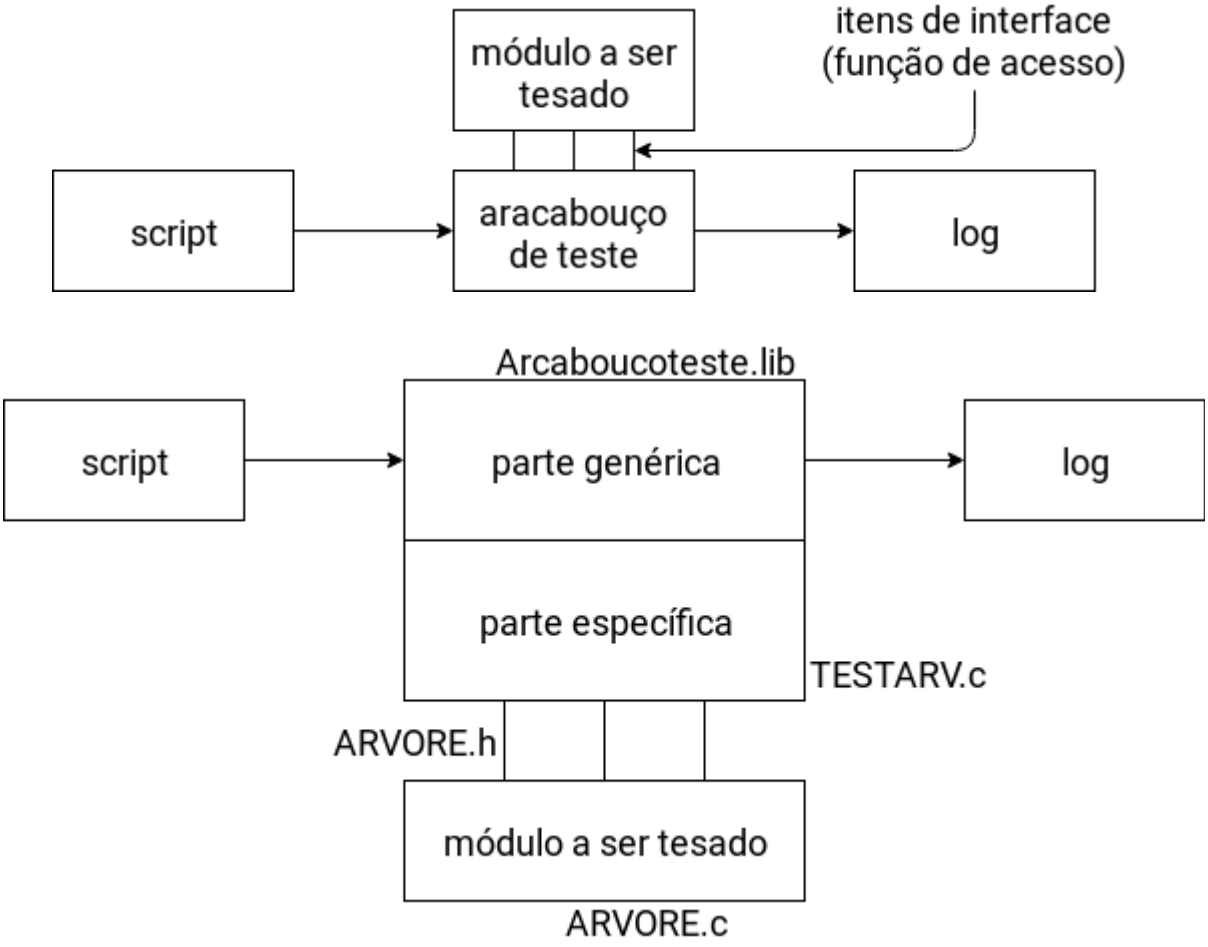
1) Objetivo

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

- Alternativa à forma de teste manual
  - Mais rápido
  - Não precisa da inserção manual de todos os dados
  - Pode testar várias vezes sem muito esforço

obs: a partir do primeiro retorno esperado diferente do obtido no log de saída, todos os resultados de execução de caso de teste não são confiáveis.
obs 2: testes não garantem que o programa funciona, mas podem provar que tem erro.

2) Framework de Teste



3) Script de teste

```
// comentario
== caso de teste -> testa determinada situação
= comando de teste -> associada a uma função de acesso
```

Teste completo: casos de teste para todas as condições de retorno de cada função de acesso do módulo. Aqui no caso, exceto condição de retorno de estouro de memória.

4) Log de saída

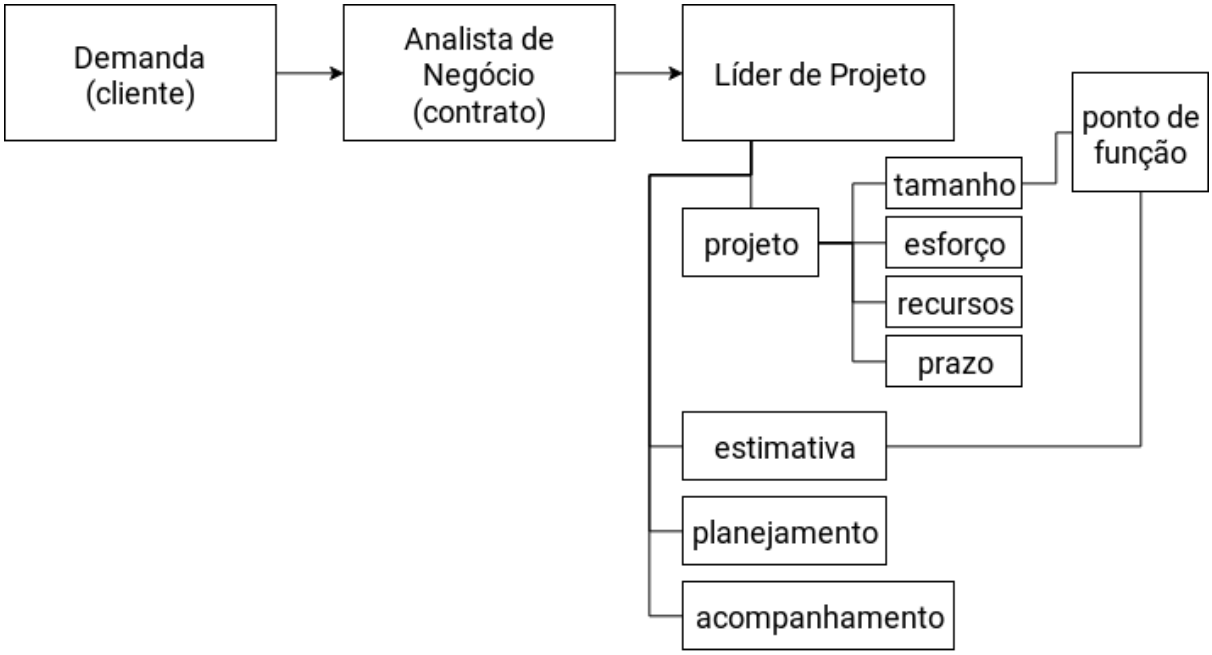
```
== caso 1
== caso 2
== caso 3
1 >> Função erperava 0 e retornou 1
0 << <- =recuperar
```

Contém relatório dos resultados dos testes.
Após um erro, nenhum resultado é confiável (falsos positivos, erros derivados)

5) Parte Específica

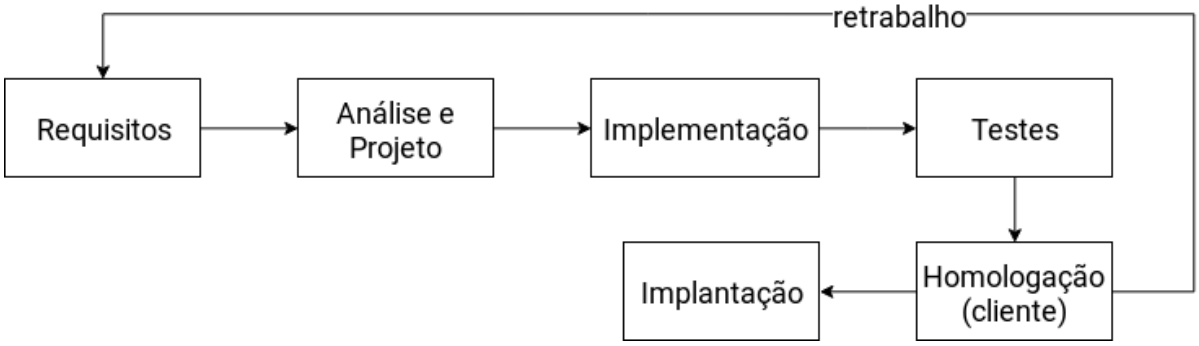
A parte específica que necessita ser implementada para que o framework (arcabouço) possa acoplar na aplicação chama-se hotspot.
Ex: TESTARV.c

Processo de Desenvolvimento em Engenharia de Software



- Líder de Projeto
- Gerência de Configuração
- Qualidade de Software
    - verifica todo o processo de desenvolvimento para manter a qualidade
        - vê se a implementação está funcionando, se os testadores estão fazendo como deviam...

Ponto de função:
Funcionalidade do Software.



Geralmente, são 3 etatpas feitas por pessoas diferentes

Requisitos

- Elicitação
  - entrar em contato com o cliente para ver o que é necessário
- Documentação
- Verificação
  - a demanda faz sentido?
  - é possível na máquina?
- Validação
  - feita pelo cliente

Análise e Projeto

- Projeto lógico
  - modelagem de dados
- Projeto físco

Implementação

- Programas
- Concretiza a linguagem de programação
- Teste unitário

O implementador deve entregar um programa perfeito (da perspectiva dele) para o testador.

Testes

- Teste integrado

O testador deve entregar um programa perfeito (da perspectiva dele) para o cliente.

Homologação

- Sugestão
  - não estava nas especificações
  - conversar com o cliente, adicionar nas especificações
    - retrabalho
- Erro
  - estava nas especificações
  - a equipe não cumpriu

Especificação de Requisitos

1) Definição de Requisto

O QUE tem que ser feito.
Não é COMO deve ser feito.

2) Escopo de Requisito

- Requisitos mais genéricos
  - O sistema cuida de folhas de pagamento
- Requisitos mais específicos
  - Documentação a ser utilizada para programas, funções

3) Fases da Especificação

- Elicitação
  - Captar informações do cliente para realizar a documentação do sistema a ser desenvolvido
  - Técnicas de elicitação
    - entrevista
      - conversar com clientes
    - brainstorm
      - colocar os clientes para decidir o que eles querme
    - questionário
      - lista de perguntas para clientes
- Documentação
  - organizar a bagunça gerada pela elicitação
  - colocar a informação desorganizada em forma de requisitos simples
  - requistos descritos em itens diretos
  - uso da língua natural
    - cuidado com ambiguidade
  - dividir requisitos em seus diversos tipos

Obs: Tipos de requisitos

a) funcionais

o que prercisa ser feito em relação à informatização das regras de negócio

b) não funcionais

propriedades que a aplicação deve ter e que não estão diretamente relacionadas com as regras de negócio

por exemplo:

segurança (i.e: login e senha)

tempo de processamento (i.e: as consultas não podem demorar mais de 5 segundos)

disponibilidade (i.e: 24x7)

c) requisitos inversos

o que não é para fazer – adicionado pelo analista de requisitos

protege de ambiguidades na documentação

deixar explícito que não será feito o que o cliente poderia depois argumentar que está implícito

- Verificação
  - vê se o que está na documentação é viável/computável
  - a equipe técnica verifica se o que está descrito na documentação é viável de ser desenvolvido
- Validação
  - cliente valida a documentação

4) Exemplos de Requisitos

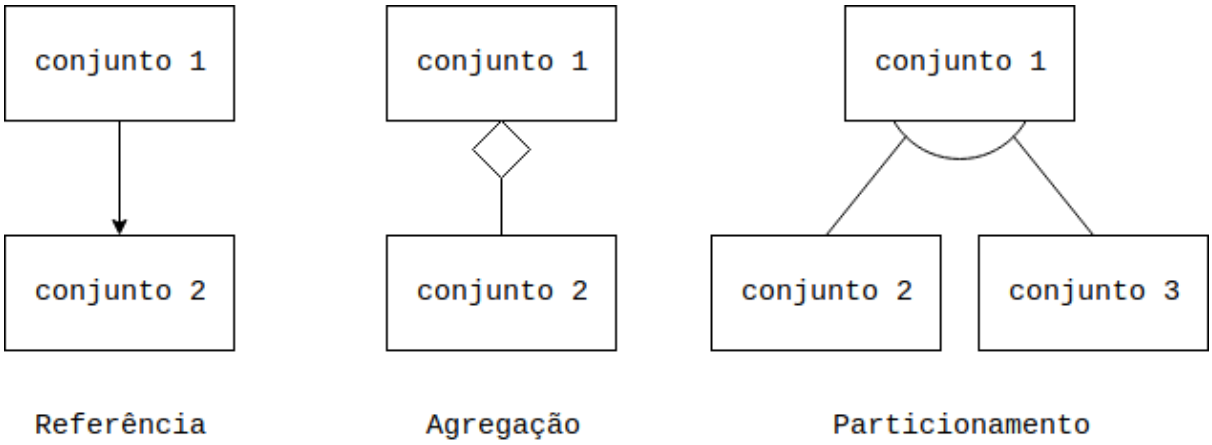
- Bem formulados
  - a tela de resposta da consulda de aluno apresenta nome e matrículo
  - todas as consultas devem retornar respostas no máximo em 2 segundos
- Mal formulados
  - o sistema é de fácil utilização
  - a consulta deverá retornar uma resposta em um tempo reduzido
  - a tela mostra seus dados mais importantes

Modelagem de Dados

1) Modelo -> n exemplos

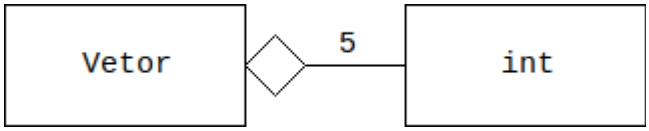
Um modelo deve representar n exemplos desse modelo. Se temos um modelo de lista encadeada, ele deve ser capaz de representar qualquer lista encadeada.

Notação

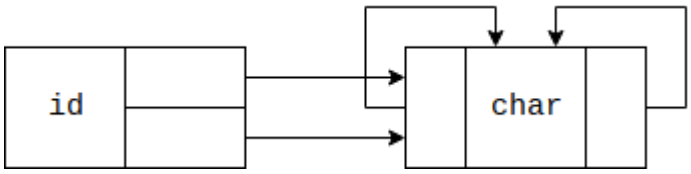


Exemplos

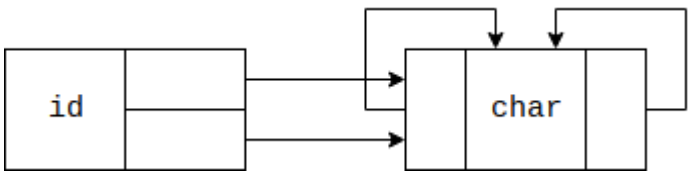
a) Vetores de 5 posições que armazenam inteiros



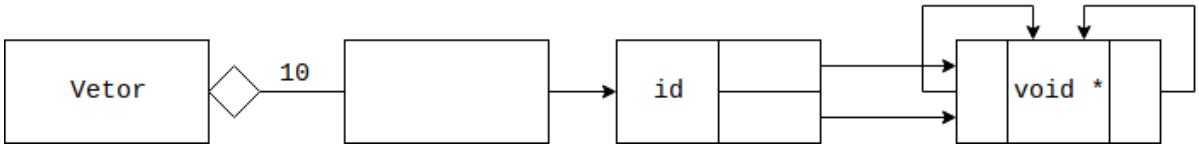
b) Árvore Binária com cabeça que armazena caracteres



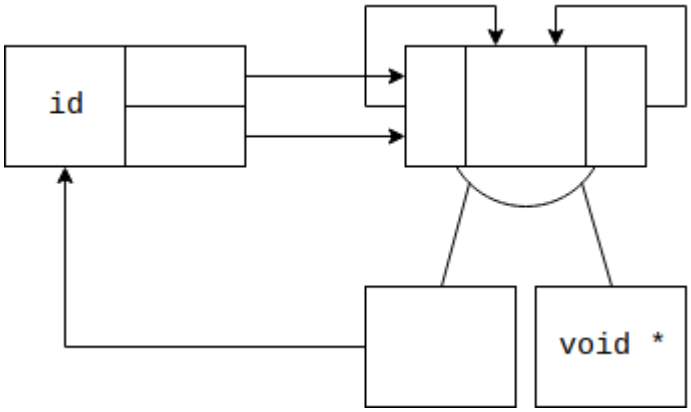
c) Lista duplamente encadeada com cabeça que armazenda caracteres



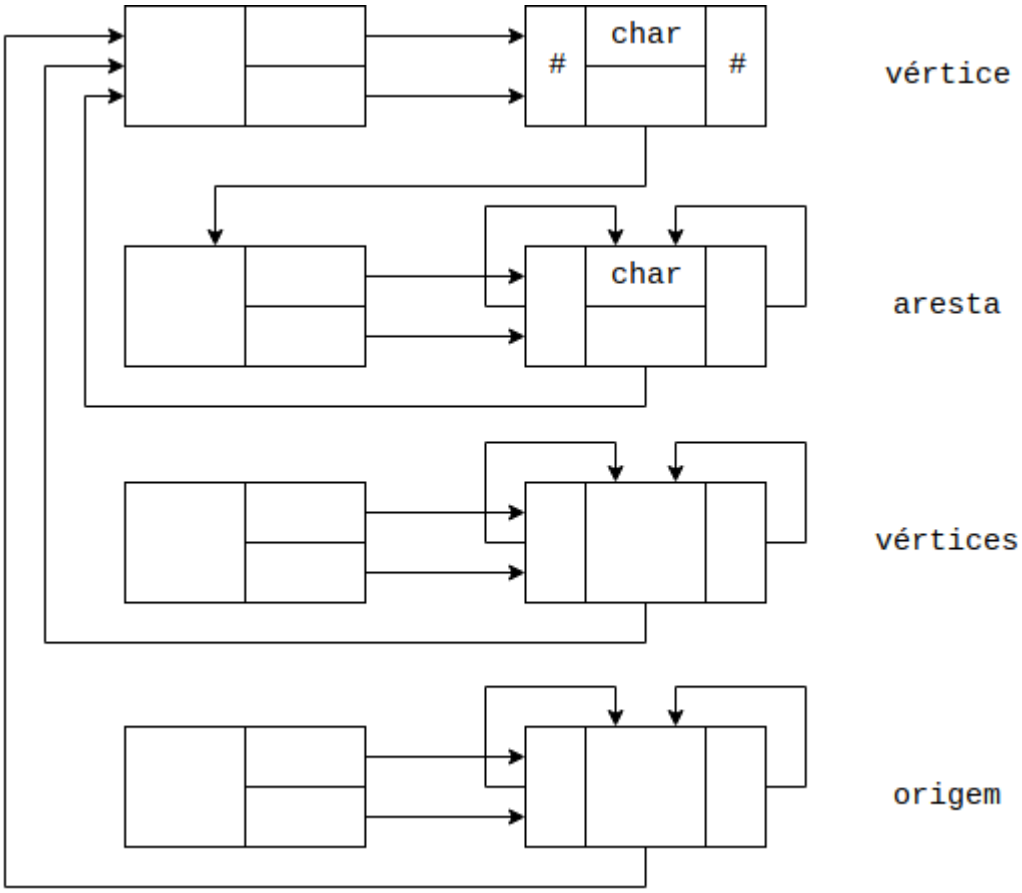
d) Vetor de listas duplamente encadeadas com cabeça e genérica



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



f) Grafo criado com listas



Assertivas Estruturais

São regras utilizadas para “desempatar” dois modelos iguais (Por exemplo os exemplos b, c e e anteriores). Estas regras complementam o modelo, definindo características que o desenho não consegue representar.

São necessárias sempre que for preciso complementar o modelo desenhado.

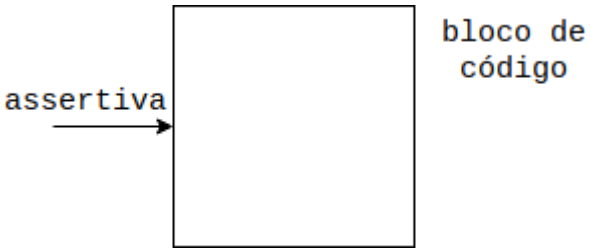
- Lista (b)
  - se `pCorr->pAnt != NULL` então `pCorr->pAnt->pProx == pCorr`
  - se `pCorr->pProx != NULL` então `pCorr->pProx->pAnt == pCorr`
- Árvore (c)
  - um ponteiro uma subarvore à esquerda nunca referencia um nó de uma subarvore à direita
  - `pEsquerda` e `pDireita` de um nó nunca apontam para o pai
- Matriz (e)
  - a matriz comporta apenas 3 dimensões, sendo o nó da lista mais interna preenchida com um `void *`

Assertivas

1) Definição

- qualidade por construção
  - qualidade aplicada a cada etapa do desenvolvimento de uma aplicação

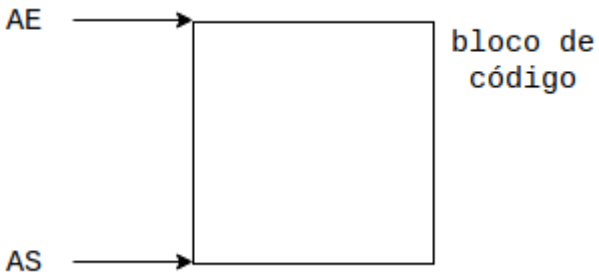
Assertivas são regras consideradas válidas em determinado ponto do código.



2) Onde Aplicar Assertivas

- argumentação de corretude
  - método para argumentar que um determinado bloco de código está correto
- instrumentação
  - transformar assertiva em trechos de código para que o próprio código verifique se ele está correto
- trechos complexos em que é grande o risco de erros

3) Assertivas de Entrada e Saída



obs: a assertiva deve tratar de regras envolvendo dados e ações tomadas

4) 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.
  - a lista existe

    - ...
- Assertiva de Saída
    - nó foi excluído
    - corrente aponta para o anterior
    - ...

Implementação da Programação Modular

1) Espaço de Dados

São áreas de armazenamento:

- alocadas em um meio
- possui um tamanho
- possui um ou mais nomes de referência

Exs:

- `A[j]`
  - j-ésimo elemento do vetor A
- `ptAux *`
  - espaço de dados apontado por ptAux
- `ptAux`
  - espaço de dados que contém um endereço
- `(*obterElemenTab(int id)).Id`, `obterElemenTab(int id)->Id`
  - é o 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.

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

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
forçar diferentes interpretações para o mesmo espaço de dados
ex: typecast `(int *) malloc(...);`

obs: conversão de tipos
não é igual a imposição de tipos
ex: transformar `1` em `“1”`

4) Tipos Básicos

a) `typedef`

- “sinônimo”

b)
```
enum {
     texto1,
     texto2,
     texto3
} tpExemplo;
```

- enumeração, tpExemplo só pode ser texto1, 2 ou 3 ou 0, 1, 2.

c) struct

- junção (organização) de vários espaços

d) union

- várias interpretações para o mesmo espaço

5) Declaração e Definição de Elementos

- Definir
  - alocar espaço
  - amarrar espaço a um nome (binding)
- Declarar
  - associar espaço a um tipo

obs:

- quando o tipo é computacional ocorrem simultaneamente a declaração e definição
- malloc só define
- typedef struct só declara

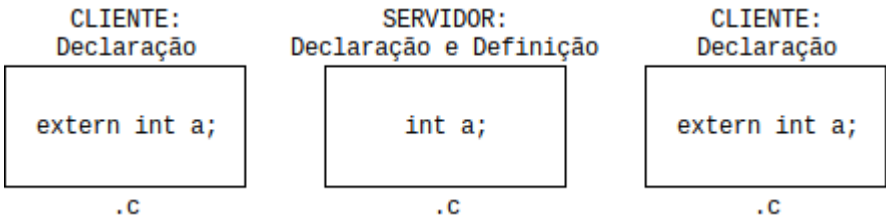
6) Implementação em C e C++

a) Declarações e definições de nomes globais exportadas pelo servidor

- `int a;`
- `int F(int b);`

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

- `extern int a;`
- `extern int F(int b);`



c) Declaração e definição de nomes globais encapsuladas no módulo

- `static int a;`
- `static int F(int b);`

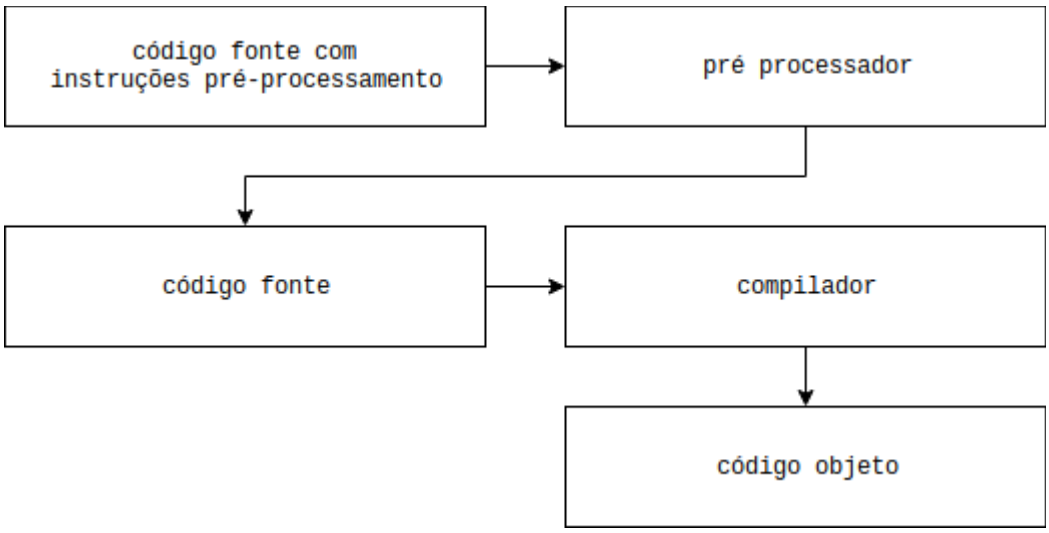
7) Resolução de Nomes Externos

obs: um nome externo somente declarado em determinado módulo necessariamente deve estar declarado e definido em algum outro módulo

A resolução:

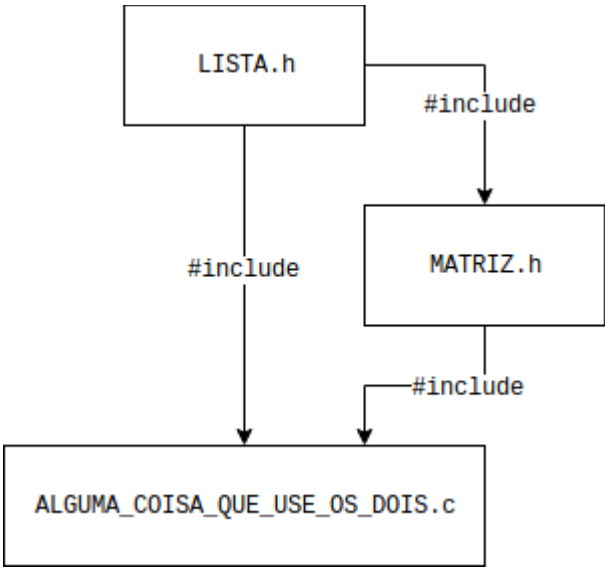
- associa os declarados aos declarados e definidos
- ajusta endereços para os espaços de dados definidos

8) Pré-Processamento




```
#define nome valor
     // substitui nome por valor

#undef nome

#if defined(nome) ou #ifdef nome
     textoV // insere esse texto se tiver definido
#else
     textoF // insere esse texto se não
#endif

#if !defined(nome) ou #ifndef nome
     // faz o contrário

#include <arquivo>
         “arquivo”
  // inclui o conteúdo (texto) do arquivo (por exemplo,
  // todo o conteúdo de stdio.h)
```

Por exemplo, guarda de includes, que previne múltiplas inclusões do mesmo arquivo num caso como o abaixo:



Outro exemplo:

```
-- M1.h --
#if defined (EXEMP_OWN)
     #define EXEMP_EXT
#else
     #define EXEMP_EXT extern
#endif

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

-- M1.c --
#define EXEMP_OWN
#include “M1.h”
#undef EXEMP_OWN

-- M2.c --
#include “M1.h”
```

Resultado:

M1.c

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

M2.c

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

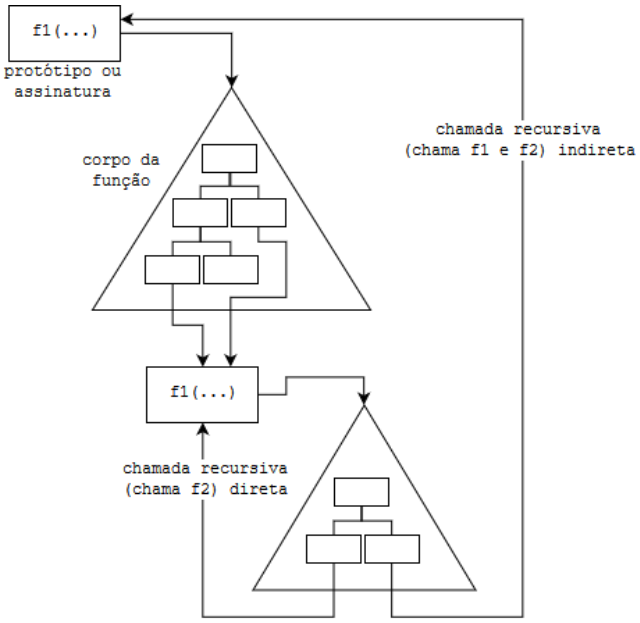
Estrutura de Funções

1) Paradigma

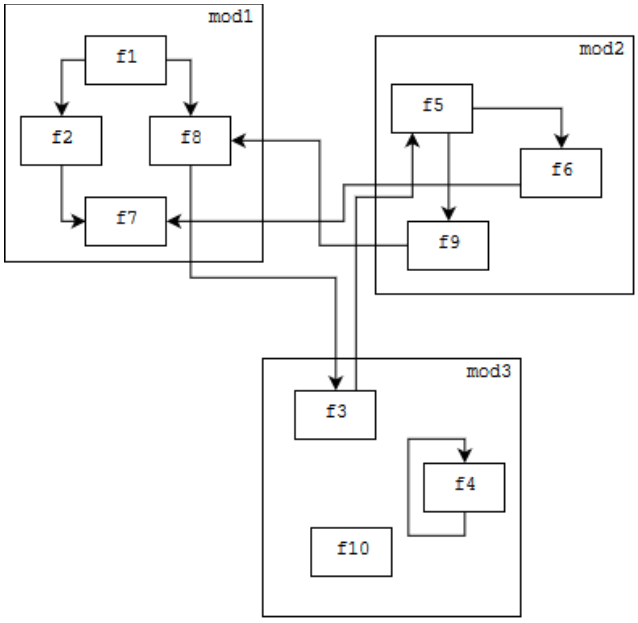
Forma de programar

- procedural: programação estruturada
- orientada a objetos:
  - programação orientada a objetos
  - programação modular (mesmo utilizando uma linguagem não orientada a objetos)

2) Estrutura de Funções



3) Estrutura de Chamadas



- `F4 -> F4`
  - `chamada recursiva direta`
- `F9 -> F8 -> F3 -> F5 -> F9`
  - `chamada recursiva indireta`
- `F10`
  - `função morta (em outra aplicação ela pode ter utilidade)`
- `F8 -> F3 -> F5 -> F6 -> F7`
  - `dependência circular entre módulos`
- `Supondo F6 -> F7`
  - `arco de chamada`

4) Função

É uma porção auto-contida de código. Possui:

- um nome
- uma assinatura
- um ou mais (ponteiro de função) corpos de código

5) Especificação da Função

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

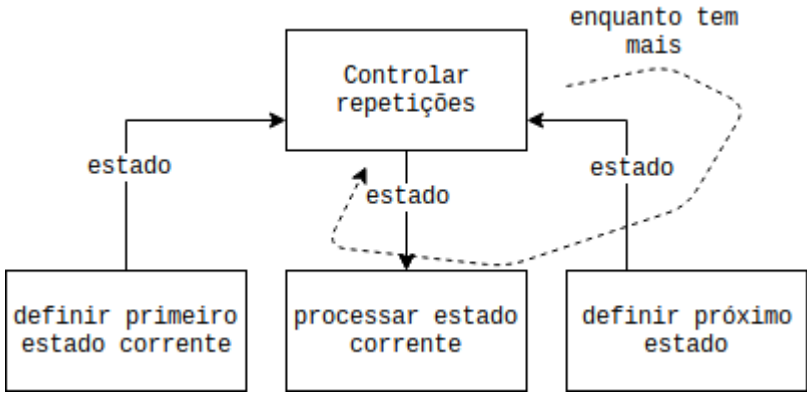
6) Interfaces

- conceitual: definição da interface da função sem preocupação com a implementação
  - inserirSimbolo(Tabela, Simbolo) -> Tabela, Id Símbolo, condRet
- físico: implementação do conceitual
  - `tpCondRet insSimb(tpSimb * simbolo)`
  - `tabela`: `global static` no módulo
- implícito
  - dados de interface diferntes de parâmetros e valores de retorno

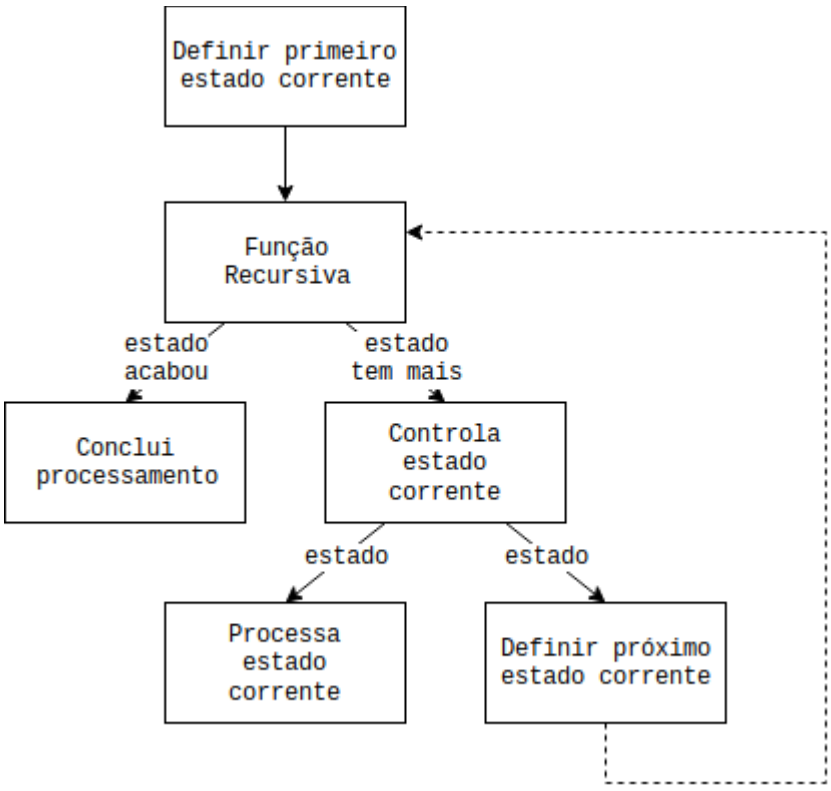
7) Housekeeping

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 define um estado
  - ex: índice em pesquisa de vetor
- estado: valoração do descritor
  - ex: `i = 0;`

Obs:

- não é necessáriamente observável
  - ex: cursor de posicionamento de arquivo
- não precisa ser único
  - ex: limite inferior e limite superior de uma pesquisa binária.
    - O descritor de estado de uma busca binária tem duas variáveis

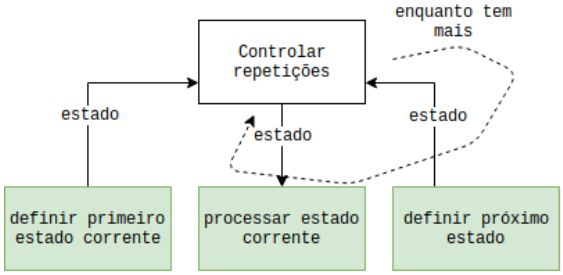
11) Esquema de Algorítmo

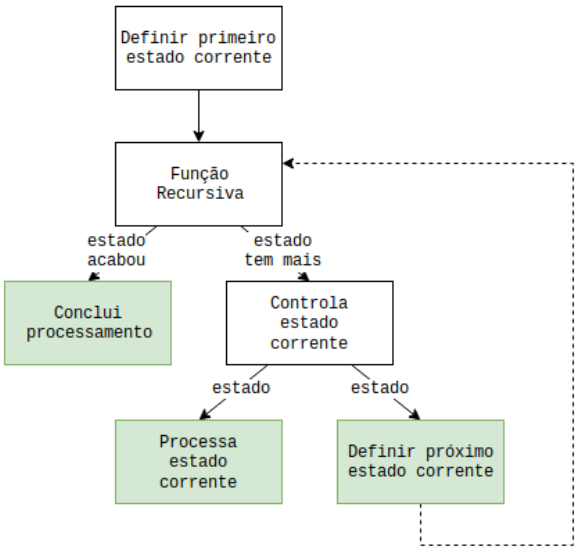
```
inf = obterLimInf();
sup = obterLimSup();
while (inf <= sup) {
     meio = (inf + sup) / 2;
     comp = comparar(valorProc, obterValor(meio));
     if (comp == igual) { break; };
     if (comp == menor) { sup = meio – 1; }
     else { inf = meio + 1; }
}
```

`obterLimInf, obterLimSup e obterValor` são hotspots.

O esquema de algorítmo é a parte genérica e os hotspots as partes específicas. Juntos formam o conceito de Framework.

Por exemplo, nos exemplos de repetição e recursão, os verdes são os hotspots:





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

Normalmente ocorrem em:

- ptrogramação orientada a objetos
- frameworks

Se esquema correto e hotspots com assertivas válidas então programa correto.

12) Parâmetros do tipo ponteiro para função

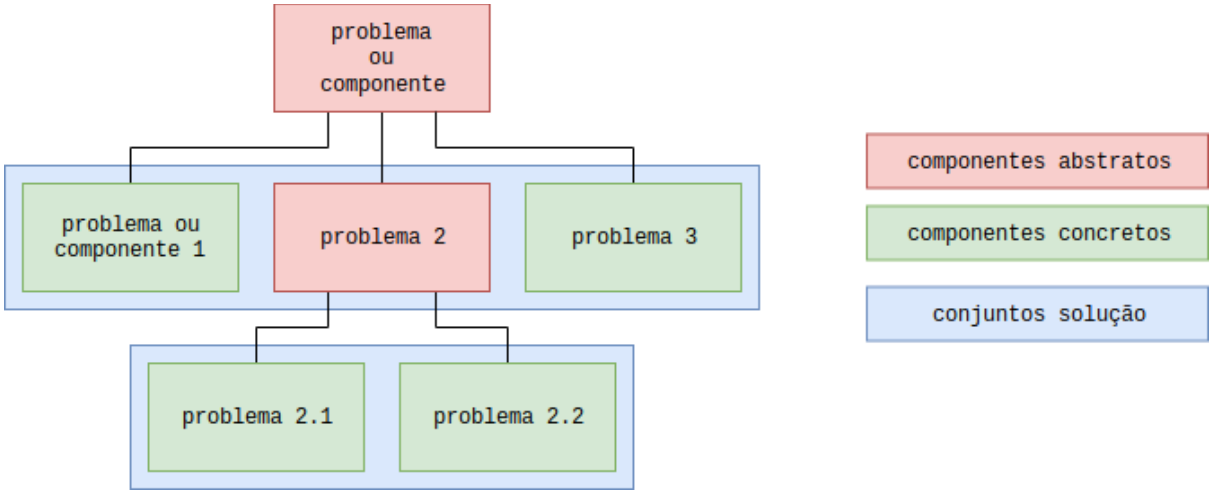
```
float areaQuad(float base, float altura) {
     return base * altura
}

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

int  ProcessaArea(float  valor1,  float  valor2,  float(*Func)
(float, float)) {
     ...
     printf("%f\n", Func(valor1, valor2));
     ...
}

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

Decomposição Sucessiva

1) Conceito

- divisão e conquista
  - dividir um problema em subproblemas menores de forma que seja possível resolvê-los
- a decomposição sucessiva é um método de divisão e conquista

2) Estruturas de Decomposição



Um componente abstrato se subdivide em subproblemas.
Um componente concreto não se subdivide. É possível a resolução dele como está.
Um conjunto solução é **um nível** de subproblemas. A união dos dois conjuntos solução acima **não são** um conjunto solução.
O problema/componente principal também pode ser chamado de componente raíz.

1 estrutura: 1 solução
1 solução: n estruturas

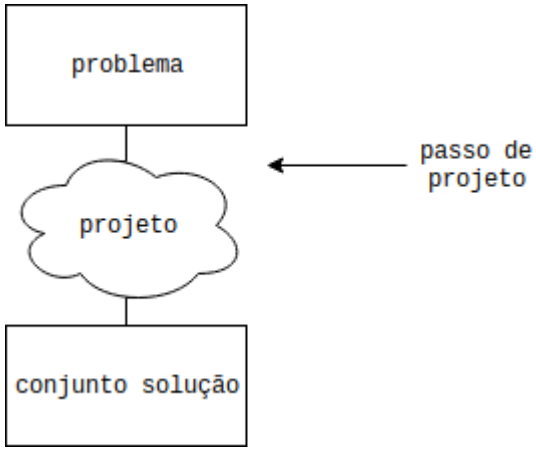
Como uma solução pode admitir várias estruturas, existem estruturas boas e ruins.

3) Critérios de Qualidade

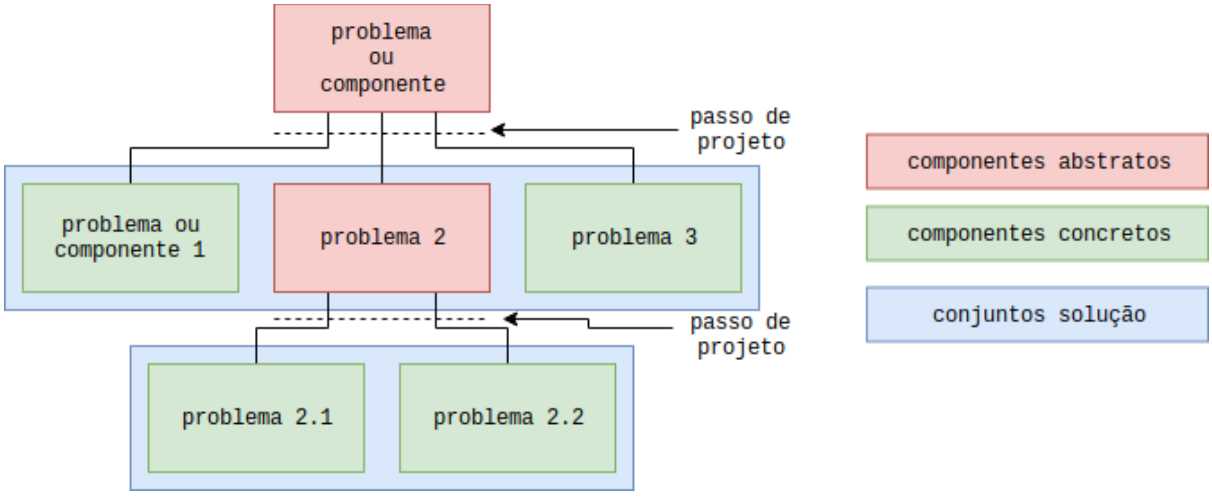
- complexidade
- necessidade
  - todos os componentes de um conjunto solução são necessários?
- suficiência
  - os componentes que fazem parde do conjunto solução são suficientes para resolver o componente abstrato?

- ortogonalidade
  - dois componentes não realizam a mesma tarefa
  - o que um componente faz, nenhum outro faz (dentro de um conjunto solução).

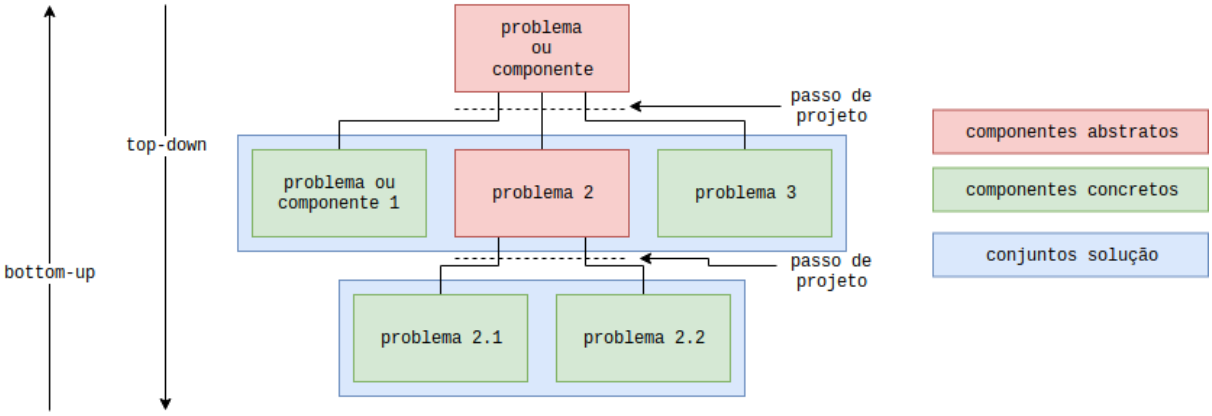
4) Passo de Projeto



Por exemplo, no exemplo dado acima:



5) Direção de Projeto



Bottom-Up: começa em baixo, testando e vai implementando.
Top-Down: faz o principal primeiro com “fakes” e dummy data, pra depois ir implementando.

Argumentação de Corretude

```
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
```

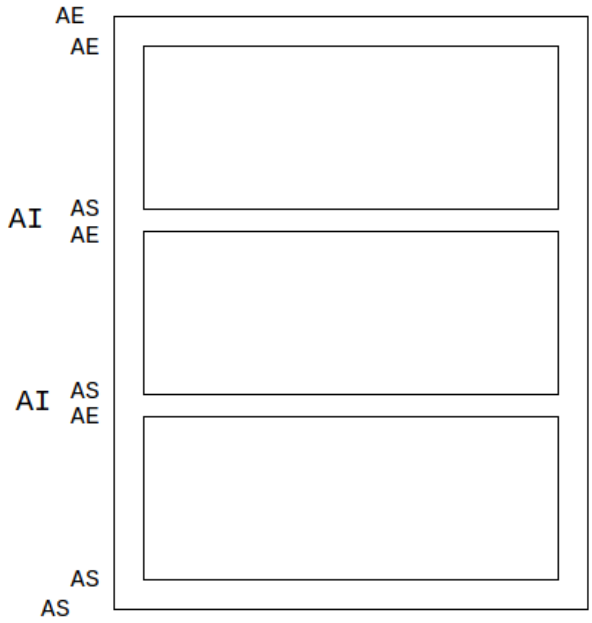
1) Definição

é um método utilizado para argumentar que um bloco de código está correto

2) Tipos de Argumentação

- sequência
  - argumentar um bloco sequencial de código
- seleção
  - pega um bloco inteiro de ifs, elses...
- repetição
  - loop

3) Argumentação de Sequência



AE - Assertiva de Entrada
AS - Assertiva de Saída
AI - Assertiva Intermediária

Sequência no código exemplo acima

`AE Principal:`

- existe um vetor válido e um elemento a ser pesquisado

`AS Principal:`

- mensagem “achou” se o elemento foi encontrado (ind aponta elemento encontrado) e “não achou” se não (ind > LL).

`AI1:`

- IND aponta para a primeira posição do vetor

`AI2:`

- se o elemento foi encontrado, ind aponta para o mesmo. Se não, ind > LL

4) Argumentação de Seleção

1

```
SE
  BLOCO
FIM-SE
```

2

```
SE
  BLOCO 1
SENÃO
  BLOCO 2
FIM-SE
```

1

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

2

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

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

O símbolo + significa executando. Exemplo: a primeira linha do primeiro por extenso:

Assertiva de Entrada verdadeira e condição verdadeira, executando B -> assertiva de saída válida

Seleção no código exemplo

```
AE: AI2
AS: AS Principal
```

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

Pela assertiva de entrada, se o elemento for encontrado ind aponta para ele e é menor do que LL. Como a condição é verdadeira, IND <= LL. B1 apresenta mensagem “achou”, valendo a assertiva de saída.

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

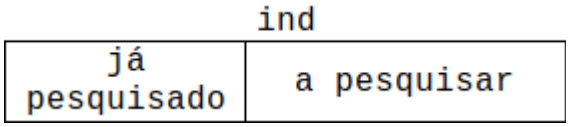
Pela assertiva de entrada, IND > LL se o elemento não for encontrado. Como a condição é falsa IND > LL, neste caso B2 é executado apresentando mensagem “não achou”, valendo assertiva de saída.

5) Argumentação de Repetição

Repetição no código exemplo
AE: AI1
AS: AI2

AINV (assertiva invariante)

- envolve os dados descritores de estado
- válida a cada ciclo da repetição
- existem 2 conjuntos: a pesquisar e já pesquisados



- ind aponta para elemento do conjunto a pesquisar

1
AE => AINV (a invariante vale antes da repetição iniciar)

- pela AE, IND aponta par ao primeiro elemento do vetor e todos estão em a pesquisar. O conjunto já pesquisado está vazio, valendo AINV.

2
AE && (c == F) => AS
- não entra ou não completa o ciclo

- não entra: pela AE, ind = 1. Como (c == F), LL < 1, ou seja, LL = 0 (vetor vazio). Neste caso vale a AS pois o elemento pesquisado não foi encontrado.
- Não completa o 1º ciclo: pela AE, IND aponta para 1º elemento do vetor. Se este for igual ao pesquisado, o break é executado e IND aponta para o elemento encontrado, vale AS.

3
AE && (c == T) + B => AINV
Pela AE, IND aponta para o primeiro elemento do vetor. Como (c == T), este 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 AINV continue valendo, B deve garantir que um elemento passe de a pesquisar para 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 a pesquisar que é igual a pesquisado. Neste caso, vale AS pois ele[ind] = pesquisado.

6
Término
Como a cada ciclo, B garante que um elemento de a pesquisar para já pesquisado e o conjunto a pesquisar possui um número finito de elementos, a repetição termina em um número finito de passos.

... continuação da argumentação do exemplo

```
          ...
          enquanto ind <= LL faça
               se ele[ind] == pesquisado
               break
               fim-se
AI3->
               ind <- ind + 1
          fim-enquanto
          ...
```

Argumentação do bloco de dentro do enquanto:

Sequência
AE: AINV
AS: AINV
AI3: o elemento pesquisado não é igual a ele[ind] ou o elemento foi encontrado em IND.

Seleção
Seleção do tipo 1.
AE: AINV
AS: AI3

AE && (c == T) + B => AS
Pela AE (AINV), IND aponta para um elemento do conjunto a pesquisar. Como (c == T), o ELE[IND] é igual ao pesquisado e assim o elemento encontrado em IND, valendo a AS.

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

Instrumentação

1) Problemas ao Realizar Testes

Esforço de Diagnose

- não saber resolver e ter que buscar a origem do erro
- 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.
  - Acontece só às vezes. Ora funciona, ora não. Não tem padrão.
- Causa externa ao código que mostra a falha
- Ponteiros loucos (*wild pointers*)
- Comportamento inesperado do hardware.

2) O que é instrumentação

- fragmentos inseridos nos módulos
  - código
  - dados
- não contribuem para o objetivo do programa
- monitora o serviço enquanto o mesmo é executado
- consome recursos de execução
- custa para ser desenvolvida

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

Programa tolerante a falhas

- É robusto
- possui mecanismos de recuperação

Deterioração controlada

- Habilidade de continuar operando corretamente mesmo com uma perda de funcionalidade

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

```
#ifdef _DEBUG
     codigo da instrumentação
     dados
#endif
```

6) Assertivas Executáveis

Assertivas -> Código

Vantagens

- Informam o problema quase imediatamente após ter sido gerado
- Controle de integridade feito pela máquina
  - integridade de memória, banco de dados, estrutura...
- Reduz o risco de falha humana

Precisam ser

- completas
- corretas

7) Depuradores

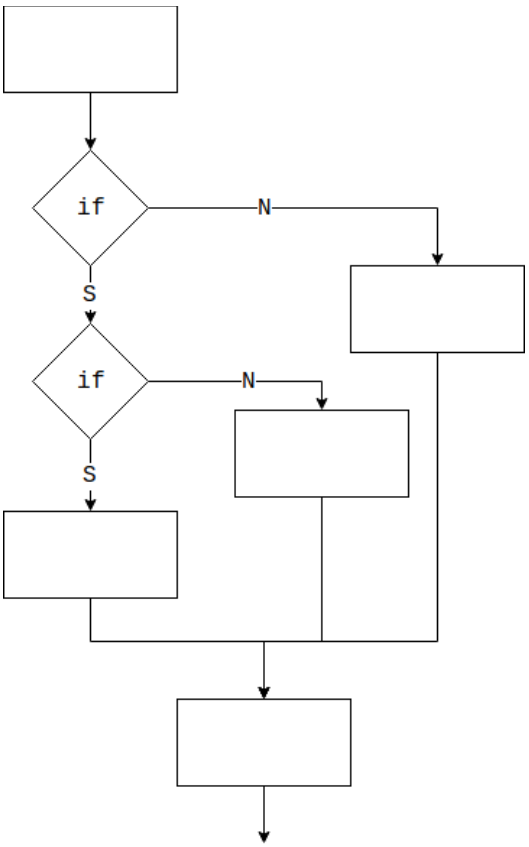
Ferramenta utilizada para executar o código passo a passo permitindo que se distribua *breakpoints* com objetivo de confinar os erros a serem pesquisados.
Obs: devem ser utilizados apenas como último recurso

8) Trace

Instrumento utilizado para apresentar uma mensagem no momento em que é executado.
Existem 2 tipos de *trace*:

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 aberta

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

Vetor de contadores:

LABEL	QTD
if1nao	0
ifsim	6
if2nao	8

Teste superficial, um caminho não foi passado.

Modulo CONTA do arcabouço:
Estrutura encapsulada vetor de contadores.

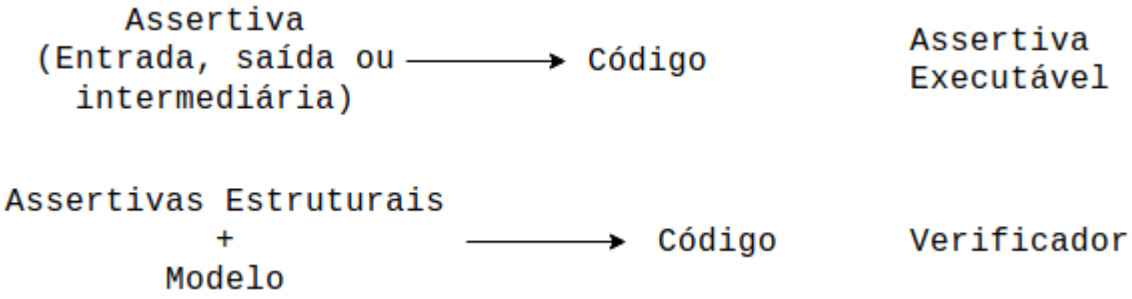
```
CNT_Contar(label);
```

No exemplo acima

```
INICIO
     COM1
     IF1 cond
          IF2 cond
               CNT_Contar(if2sim);
               COM2
          ELSE
               CNT_Contar(if2nao);
               COM5
          FIM-IF2
     ELSE
          CNT_Contar(if1nao);
          COM4
     FIM-IF1
     COM3
FIM
```

Não é necessário criar o if1sim, pois será possível ver se entrou no if2sim e no if2nao.
Se algum contador estiver zero, o teste é superficial.

10) Verificador Estrutural



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

11) Deturpador Estrutural

Quebra a estrutura correta.
Instrumento responsável por inserir erros na estrutura com o objetivo de testar o verificador.

```
deturpa(tipoDet);
```

Obs: a deturpação é sempre realizada no nó corrente.

```
=deturpa 1
=verifica
=deturpa 2
=verifica
=deturpa 3
...
=estatcont
```

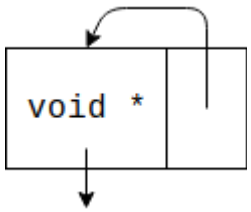
12) Recuperador Estrutural

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

13) Estrutura Auto Verificável

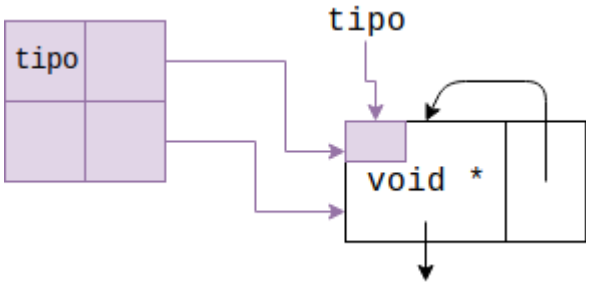
É a estrutura que contém todos os dados necessários para que seja totalmente verificada.

Exemplo: lista simplesmente encadeada genérica



É possível verificar todos os tipos apontados pela estrutura?

- incluir um campo tipo
- no modelo tem que aparecer diferente
- inclusão de um campo tipo no cabeça




```
inserir(pt, valor
          #ifdef _DEBUG
               , tipo
```

```
#endif
)
```

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

- inclusão de um campo pAntes e um pCabeça