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

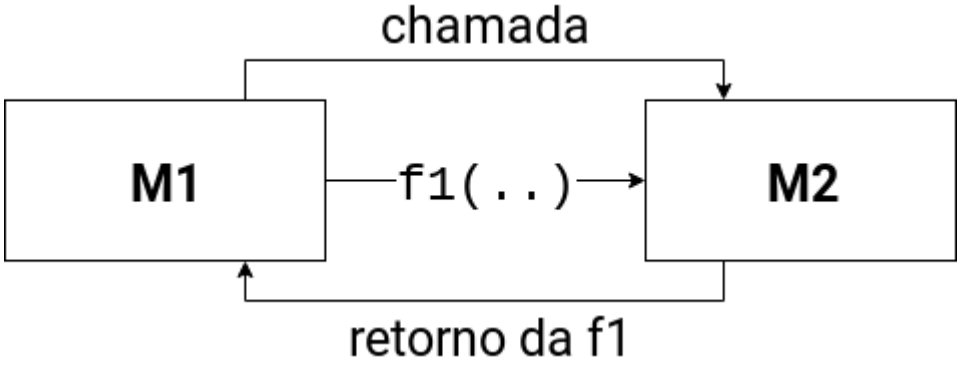
Mecanismo de troca de *dados, estados e eventos* entre elementos de um **mesmo nível de abstração**.

- Interfaces de sistema com sistema, entre módulos…

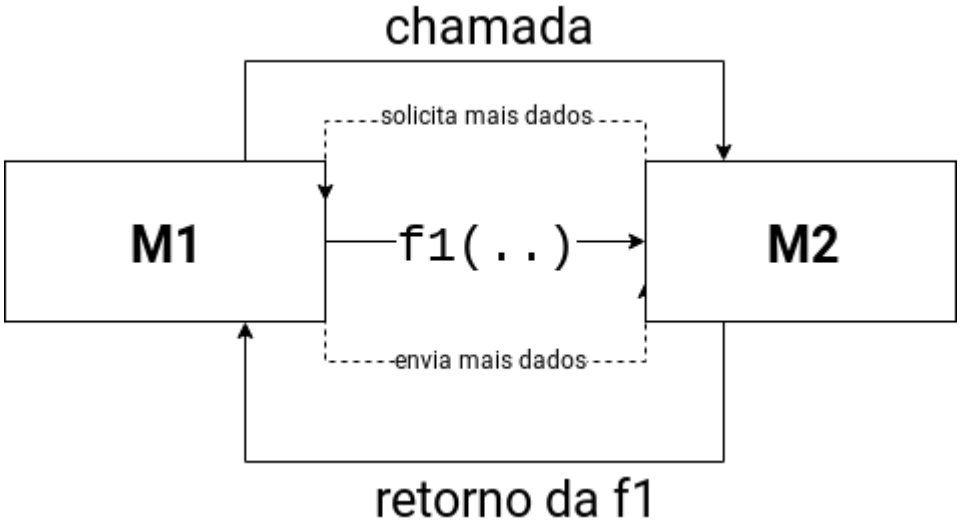
a) Exemplos de Interface

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

b) Relacionamento Cliente - Servidor

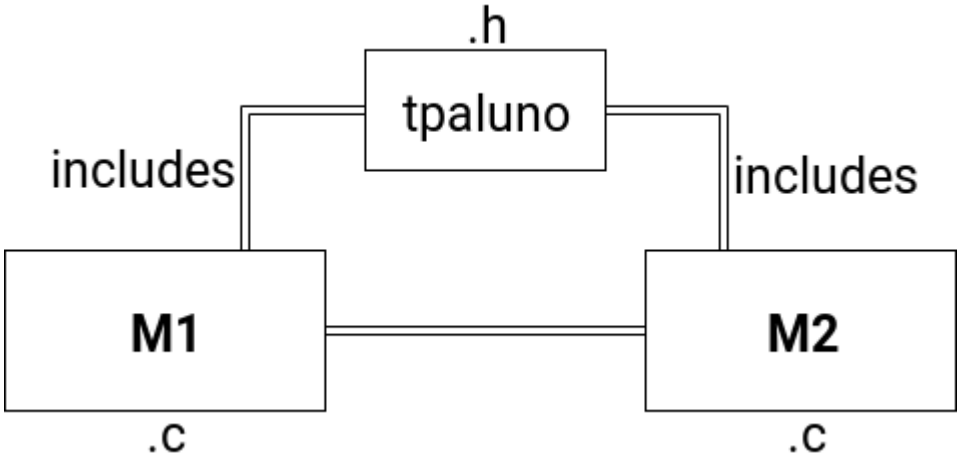


Caso especial: callback



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)

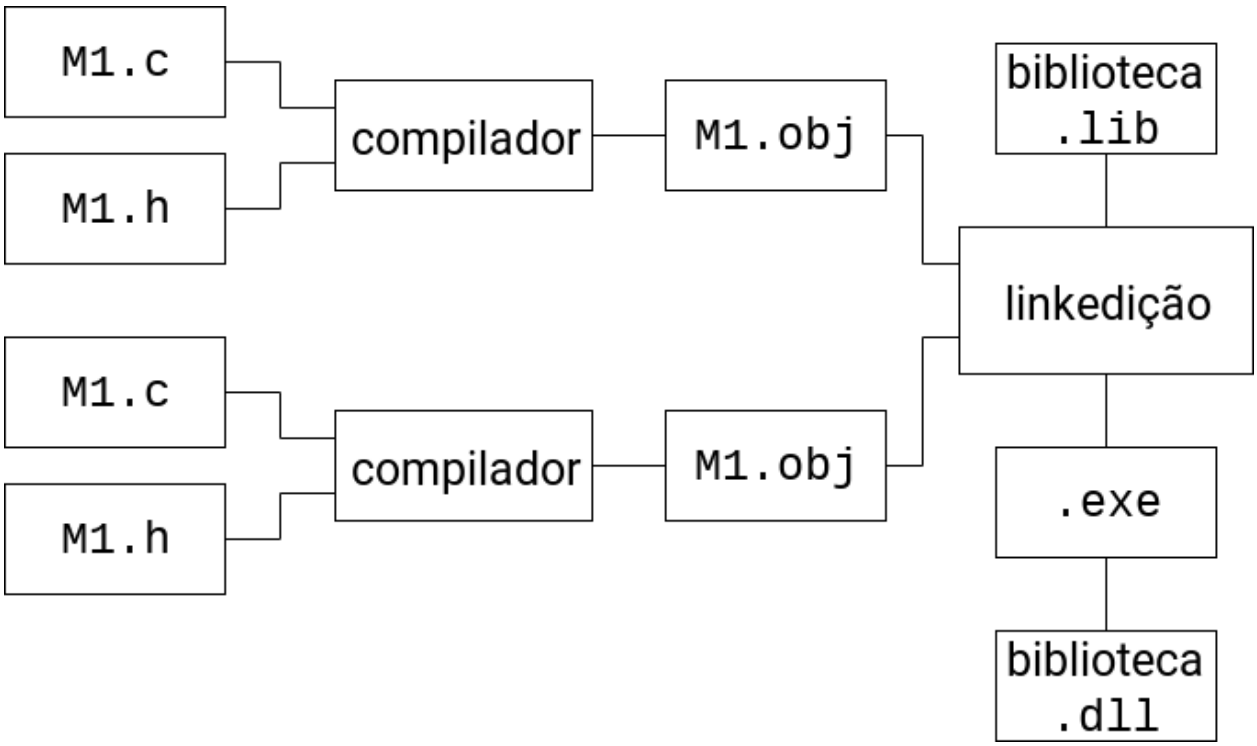
- **Semântica**: significado
  `MOD 1 (int) <----> MOD2 (int)`
  Depende, se um `int` for *idade* e o outro *quantidade de filhos*, há algum problema.

e) Análise de Interface

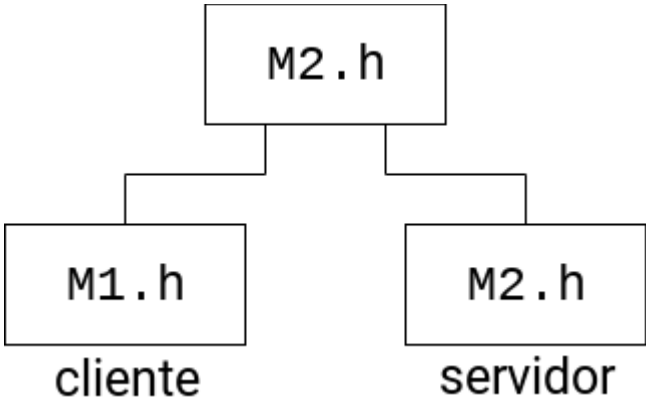
`tpDadosAluno * opterDadosAluno(int mat); // protótipo ou assinatura de função de acesso`

- 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.

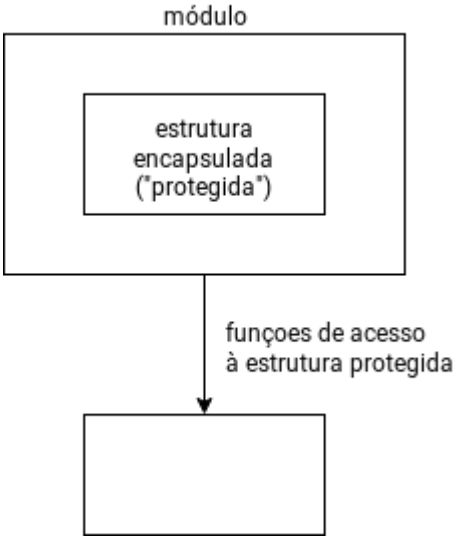
6) Módulo de Definição (`.h`)

- 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`)

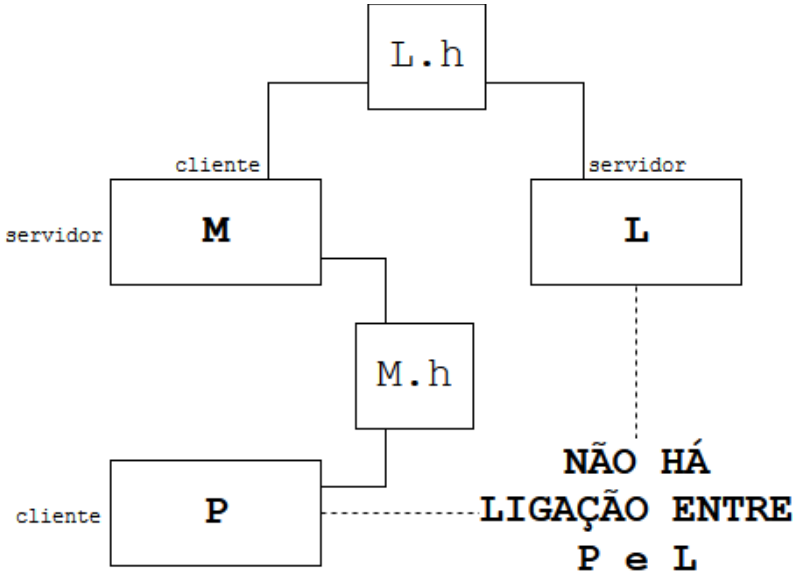
- Código das funções de acesso
- Códigos e protótipos das funções internas
- Variáveis internas ao módulo
- Documentação voltada para o programador do módulo servidor

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

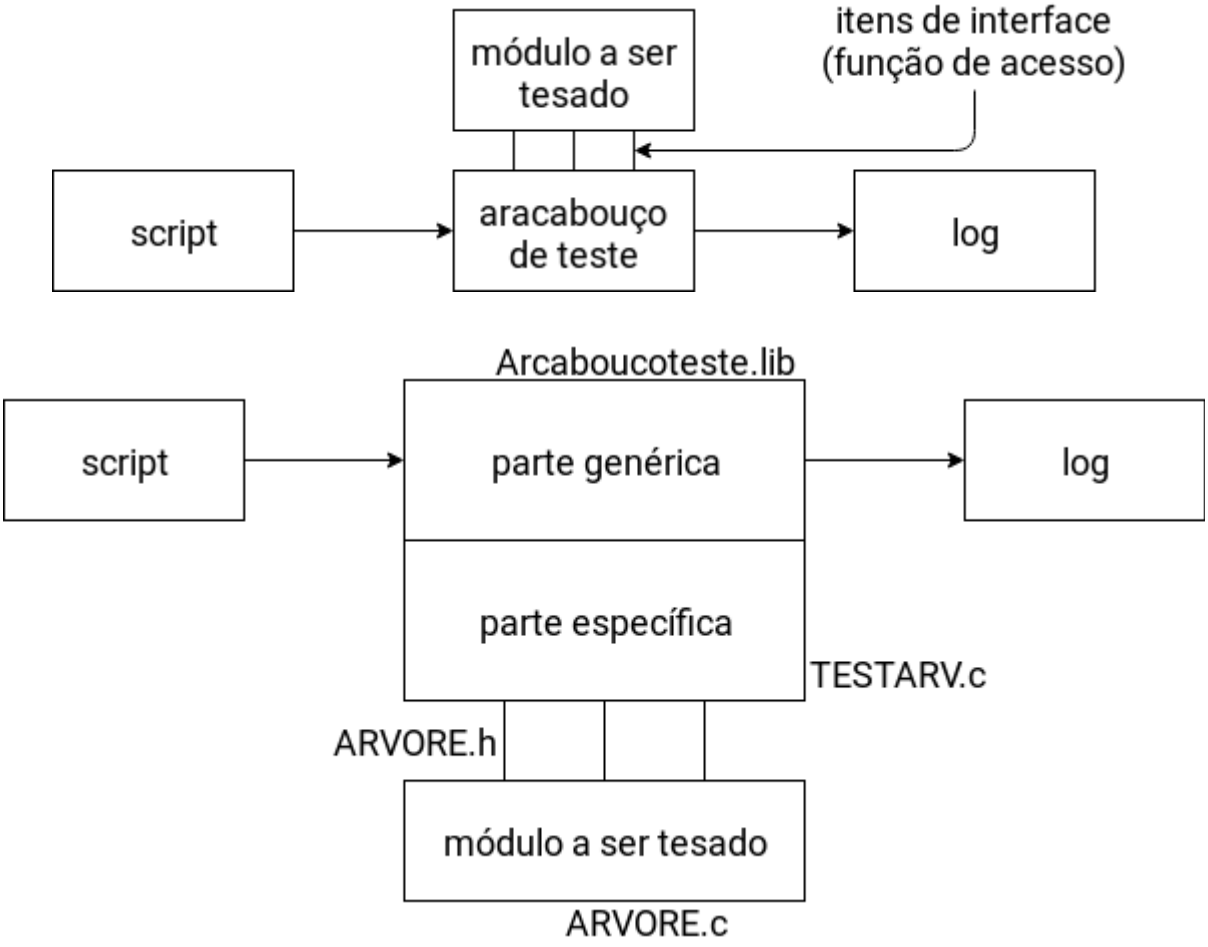
1) Objetivo

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

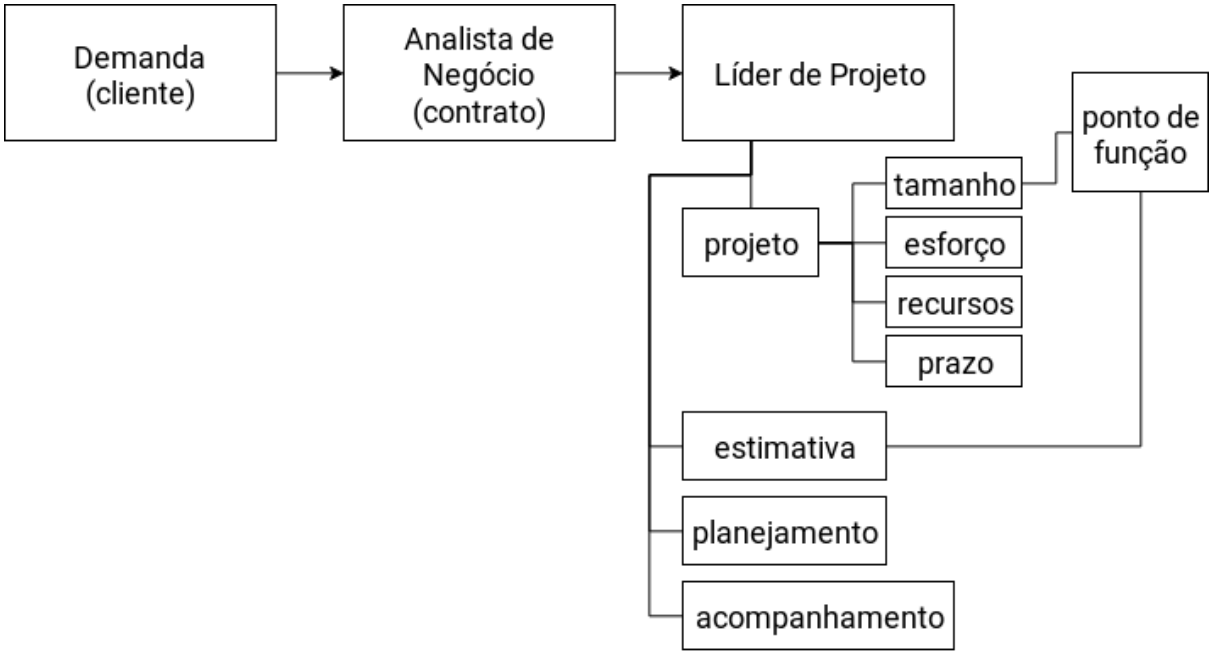
```
== caso 1
== caso 2
== caso 3
1 >> Função erperava 0 e retornou 1
0 << <- =recuperar
```

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

5) Parte Específica

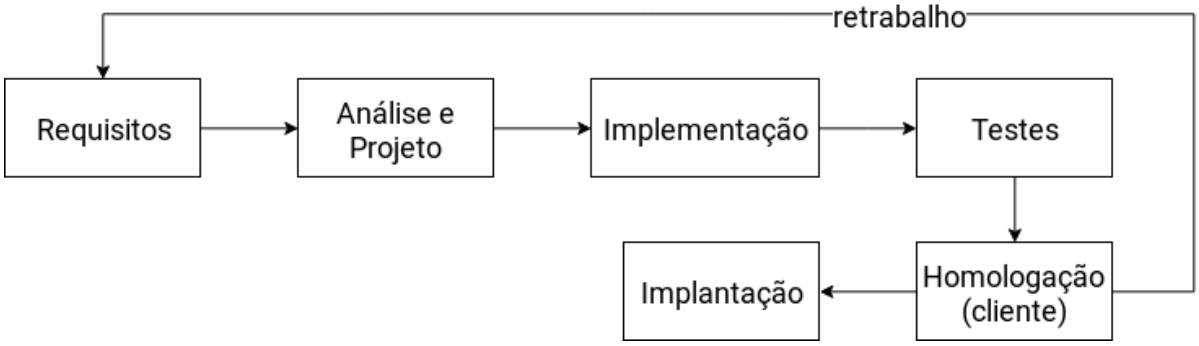
A parte específica que necessita ser implementada para que o framework (arcabouço) possa acoplar na aplicação chama-se hotspot.
Ex: TESTARV.c

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