

---

# Evolução de Software

---

# Mudança de software

---

- Mudança de software é inevitável
  - Novos requisitos surgem quando o software é usado;
  - O ambiente de negócio muda;
  - Erros devem ser reparados;
  - Novos computadores e equipamentos são adicionados ao sistema;
  - O desempenho ou a confiabilidade do sistema deve ser melhorada.
- Um problema-chave para as organizações é a implementação e o gerenciamento de mudanças em seus sistemas.

# Importância da evolução

---

- As organizações fazem grandes investimentos em seus sistemas de software – eles são **ativos críticos** de negócios.
- Para manter o valor desses ativos de negócio, eles devem ser mudados e atualizados.
- A maior parte do orçamento de software nas grandes organizações é voltada para evolução do software existente ao invés do desenvolvimento de um novo software.

# Dinâmica da evolução de programas

---

- **Dinâmica de evolução de programas** é o estudo dos processos de mudança de sistema.
- Lehman e Belady propuseram que havia uma série de 'leis' que se aplicavam a **todos** os sistemas quando eles evoluíam.
- Na prática, são observáveis, de fato, mas com ressalvas
  - Aplicáveis principalmente a sistemas de grande porte

# Leis de Lehman

**Tabela 21.1** Leis de Lehman

Lei	Descrição
Mudança contínua	Um programa usado em um ambiente real deve mudar necessariamente ou tornar-se progressivamente menos útil.
Complexidade crescente	À medida que um programa muda, sua estrutura tende a se tornar mais complexa. Recursos extras devem ser dedicados para preservar e simplificar a estrutura.
Evolução de programa de grande porte	A evolução de programa é um processo auto-regulável. Atributos de sistemas como tamanho, tempo entre versões e número de erros reportados é quase invariável em cada versão de sistema.
Estabilidade organizacional	Durante o ciclo de vida de um programa, sua taxa de desenvolvimento é quase constante e independente de recursos dedicados ao desenvolvimento do sistema.
Conservação de familiaridade	Durante o ciclo de vida de um sistema, mudanças incrementais em cada versão são quase constantes.
Crescimento contínuo	A funcionalidade oferecida pelos sistemas deve aumentar continuamente para manter a satisfação do usuário.
Qualidade em declínio	A qualidade dos sistemas entrará em declínio a menos que eles sejam adaptados a mudanças em seus ambientes operacionais.
Sistema de feedback	Os processos de evolução incorporam sistemas de feedback com vários agentes e loops e você deve tratá-los como sistemas de feedback para conseguir aprimoramentos significativos de produto.

# Aplicabilidade das leis de Lehman

---

- As leis de Lehman parecem ser, geralmente, aplicáveis a sistemas customizados de grande porte desenvolvidos por grandes organizações.
  - Confirmado em mais recente trabalho de Lehman sobre o projeto FEAST (veja leitura adicional no Website do livro)
- Não está claro como elas devem ser modificadas para
  - Sistemas que incorporam um número significativo de componentes COTS;
  - Pequenas organizações;
  - Sistemas de tamanho médio.

# Manutenção de software

---

- É a modificação de um programa após ter sido colocado em uso.
- A manutenção normalmente **não envolve** mudanças consideráveis na arquitetura do sistema.
- As mudanças são implementadas pela modificação de componentes existentes e pela adição de novos componentes ao sistema.

# A Manutenção é Inevitável

---

- Os sistemas estão fortemente acoplados ao seu ambiente.
  - Quando um sistema é instalado em um ambiente, ele muda esse ambiente e, portanto, muda os requisitos de sistema.
- Portanto, os sistemas DEVEM ser mantidos se forem úteis em um ambiente.



# Tipos de manutenção

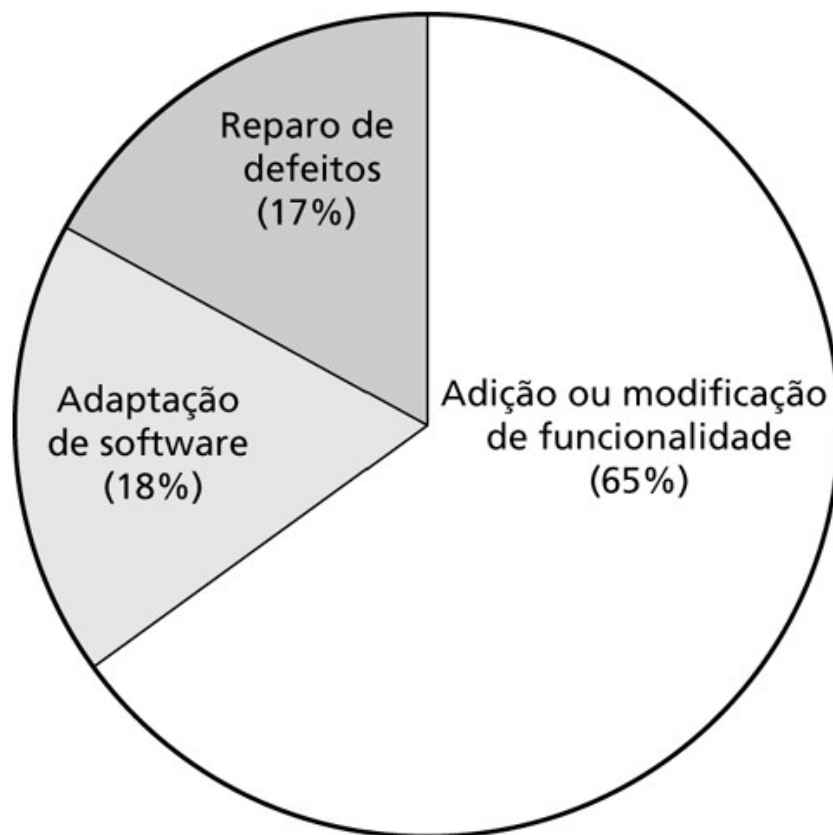
---

- Manutenção para **reparar defeitos** de software
  - Mudança em um sistema para corrigir deficiências de maneira a atender aos seus requisitos.
- Manutenção para **adaptar** o software a um ambiente operacional diferente
  - Mudança de um sistema de tal maneira que ele opere em um ambiente diferente (computador, OS, etc.) à partir de sua implementação inicial.
- Manutenção para **adicionar funcionalidade** ao sistema ou modificá-lo
  - Modificação do sistema para satisfazer a novos requisitos.

# Distribuição de esforços de manutenção

**Figura 21.2**

Distribuição de esforços de manutenção.



# Custos de manutenção

---

- Geralmente, são maiores que os custos de desenvolvimento (de 2 a 100 vezes, dependendo da aplicação).
- São afetados por fatores técnicos e não técnicos.
- A manutenção corrompe a estrutura do software, tornando a manutenção posterior mais difícil.
  - *Design Erosion*
- Software em envelhecimento pode ter altos custos de suporte (por exemplo, linguagens antigas, compiladores, etc.).

# Fatores de custo de manutenção

---

- Estabilidade da equipe
  - Os custos de manutenção são reduzidos se o mesmo pessoal estiver envolvido por algum tempo.
- Responsabilidade contratual
  - Os desenvolvedores de um sistema podem não ter responsabilidade contratual pela manutenção, portanto, não há incentivo para projetar para mudanças futuras.
- Habilidade do pessoal
  - O pessoal da manutenção geralmente é inexperiente e tem conhecimento limitado de domínio.
- Idade e estrutura do programa
  - À medida que os programas envelhecem, sua estrutura é degradada e se torna mais difícil de ser compreendida e modificada.

# Previsão de manutenção

---

- A previsão de manutenção está relacionada à avaliação de quais partes do sistema podem causar problemas e ter altos custos de manutenção
  - A aceitação de mudança depende da facilidade de manutenção dos componentes afetados por ela;
  - A implementação de mudanças degrada o sistema e reduz a sua facilidade de manutenção;
  - Os custos de manutenção dependem do número de mudanças, e os custos de mudança dependem da facilidade de manutenção.
- XP vai na contramão da previsão de manutenção

# Métricas de complexidade

---

- Ajudam na realização das previsões de facilidade de manutenção
- A maior parte do esforço de manutenção é despendida sobre um número relativamente pequeno de componentes de sistema.
- A **complexidade** depende de vários fatores:
  - Acoplamento
  - Coesão
  - Tamanho dos elementos
  - Separação de interesses em termos de elementos de código
  - As métricas acima podem ser coletadas *off-line*

# Métricas de processo

---

- Podem ser usadas para avaliar a facilidade de manutenção
  - Número de solicitações para manutenção corretiva;
  - Tempo médio necessário para análise de impacto;
  - Tempo médio para implementar uma solicitação de mudança;
  - Número de solicitações de mudança pendentes.
- Essas métrica precisam ser colhidas *on-line*
- Se qualquer uma ou todas essas estão aumentando, isso pode indicar um declínio na facilidade de manutenção.

# Processos de evolução

---

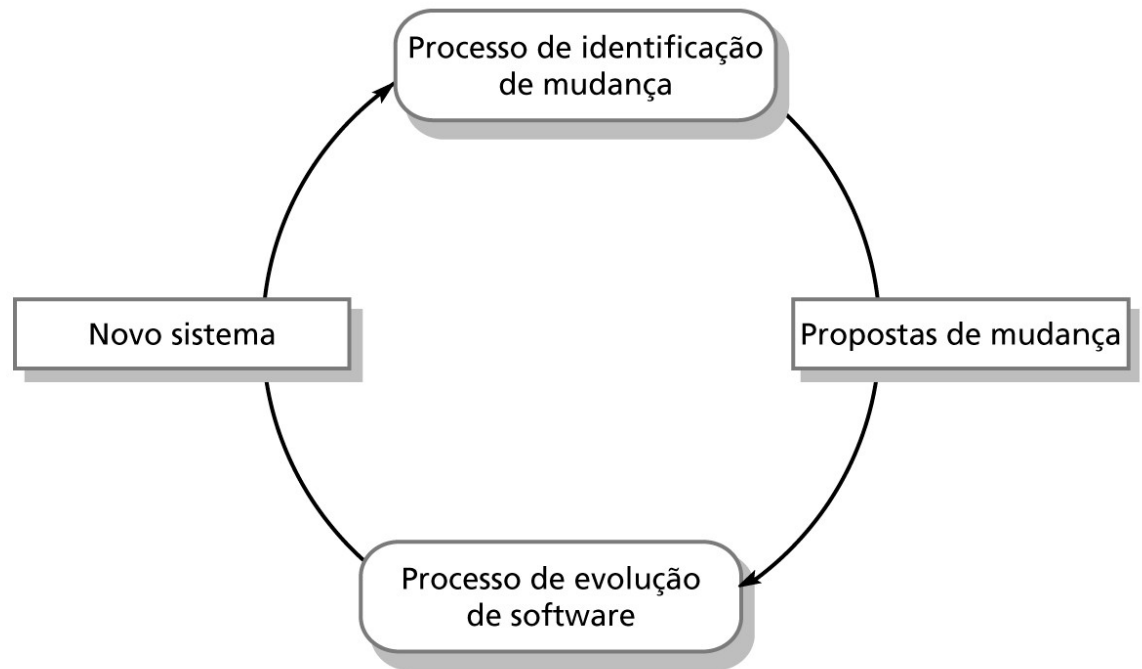
- Os processos de evolução dependem
  - Do tipo de software que está sendo mantido;
  - Dos processos de desenvolvimento usados;
  - Das habilidades e das experiências do pessoal envolvido;
- Propostas para mudança são os direcionadores para a evolução do sistema.
- Metodologias mais novas não costumam ter um **processo separado**



# Identificação de mudança e evolução

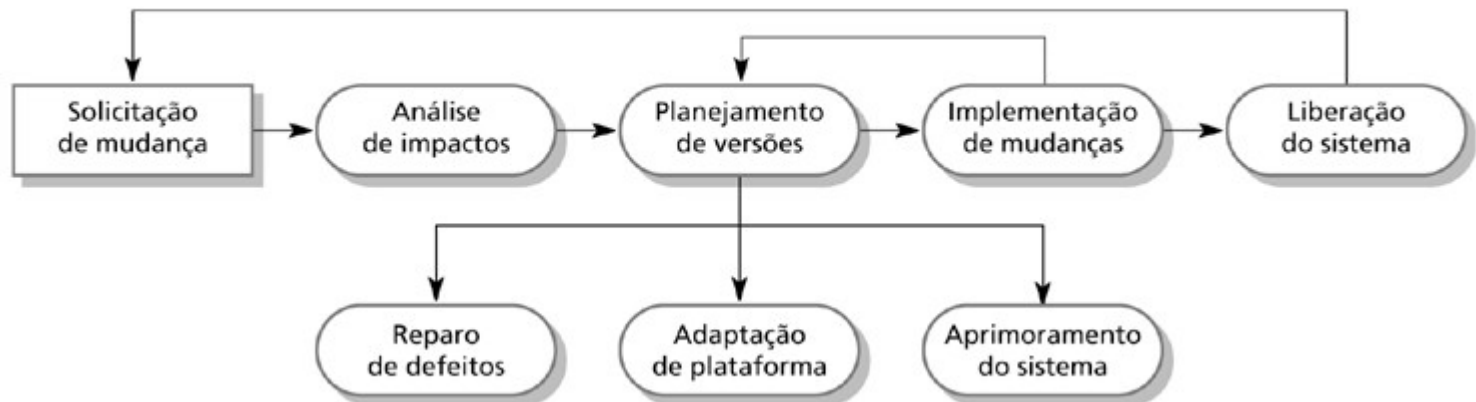
**Figura 21.5**

Processos de identificação de mudança e evolução.



# O processo de evolução de sistema

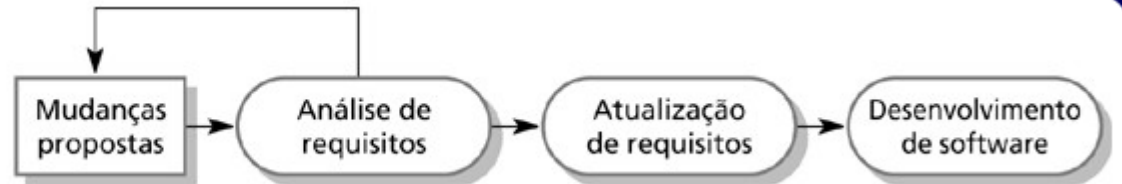
**Figura 21.6** Processo de evolução de sistema.



# Implementação de mudanças

**Figura 21.7**

Implementação de mudança.



Na prática, pode envolver todas as etapas do processo

# Solicitações de mudança urgentes

---

- Mudanças **urgentes** podem ter de ser implementadas sem passar por todos os estágios do processo de desenvolvimento de software
  - Se um defeito sério de sistema tem de ser reparado;
  - Se mudanças no ambiente do sistema (por exemplo, atualização do OS) têm efeitos inesperados;
  - Se existem mudanças de negócio que necessitam de uma resposta muito rápida (e.g.mudança de lei)
- POP – *Patch-Oriented Programming*
  - Podem resultar em problemas ainda piores

# Reengenharia de sistema

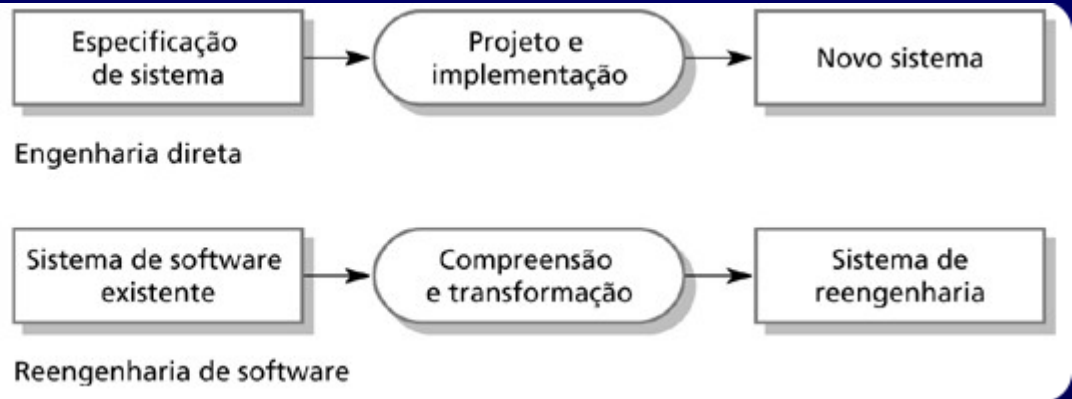
---

- É a **reestruturação** ou **reescrita** de parte ou de todo um sistema sem mudar sua funcionalidade.
  - Importante ressaltar: **reestruturação de grande porte!**
- Aplicável onde partes de um sistema de grande porte necessitam de manutenção freqüente.
- Envolve a adição de esforço para tornar o sistema mais fácil de manter.
  - **Simplicidade** é um objetivo **complexo**

# Engenharia direta e reengenharia

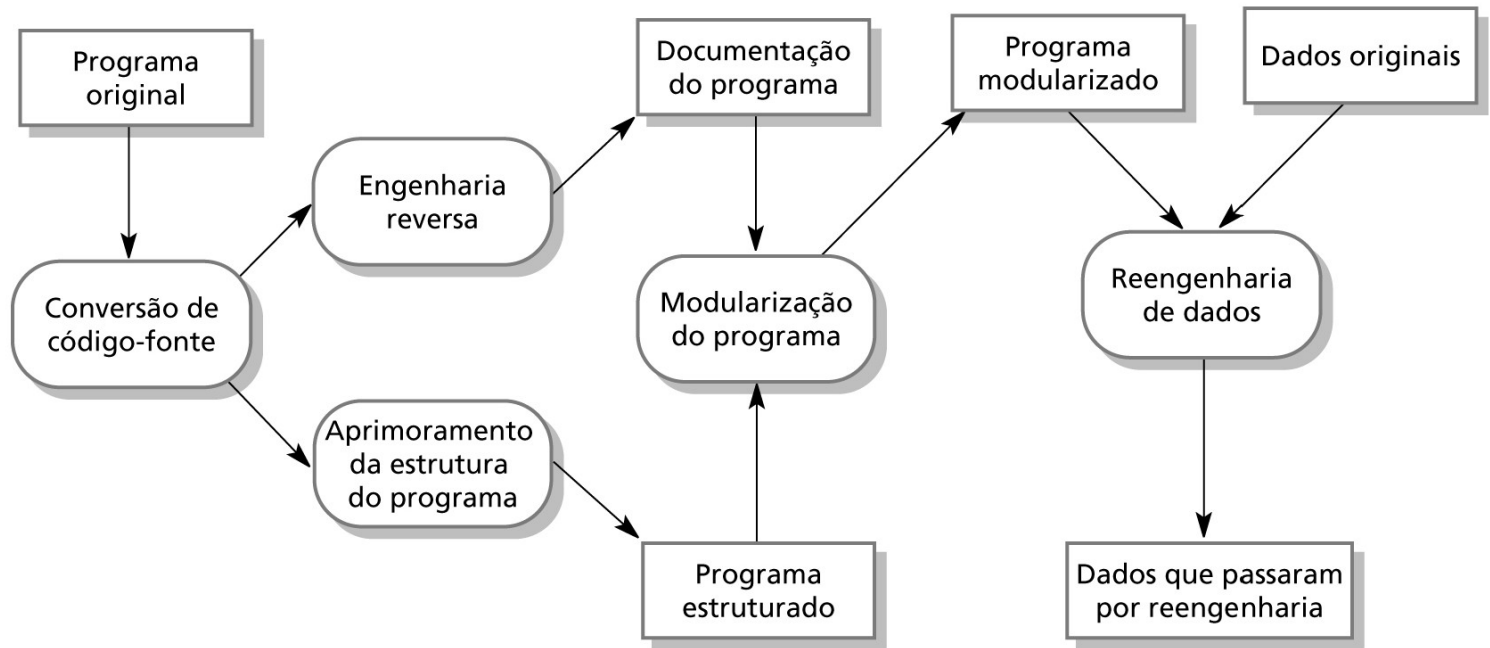
**Figura 21.9**

Engenharia direta e reengenharia.



# O processo de reengenharia

**Figura 21.10** Processo de reengenharia.



# Atividades do processo de reengenharia

---

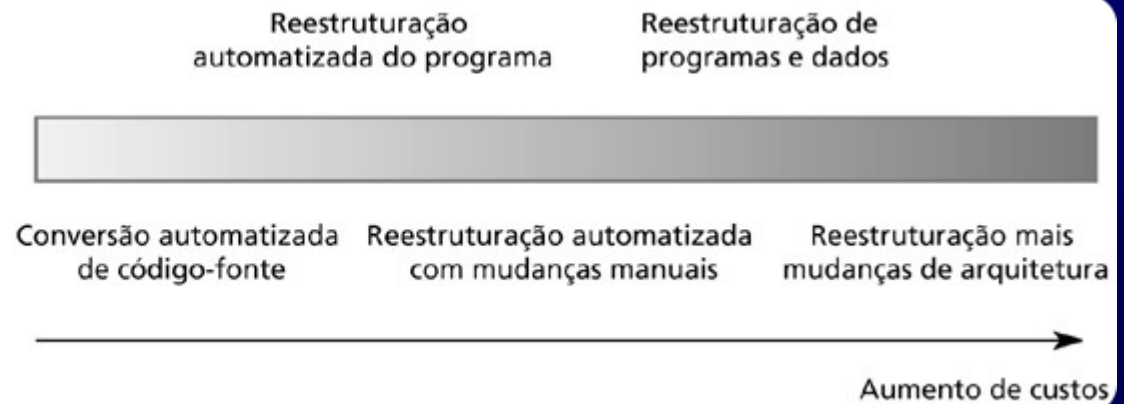
- Conversão de código-fonte
  - Converter o código para uma nova linguagem.
- Engenharia reversa
  - Analisar o programa para compreendê-lo.
- Aprimoramento da estrutura de programa
  - Analisar e modificar a estrutura para facilidade de entendimento.
- Modularização de programa
  - Reorganizar a estrutura do programa.
- Reengenharia de dados
  - Limpar e reestruturar os dados do sistema.



# Abordagens de reengenharia

**Figura 21.11**

Abordagens de reengenharia.




# Fatores do custo de reengenharia

---

- A qualidade do software que deve passar pela reengenharia.
- O apoio de ferramentas disponíveis para reengenharia.
- Extensão da conversão de dados.
- A disponibilidade do pessoal especializado
  - Pode ser um problema com sistemas antigos

# Refatoração: Melhorando a Qualidade de Código Pré- Existente



Prof. Dr. Fabio Kon

Prof. Dr. Alfredo Goldman

Departamento de Ciência da Computação  
IME / USP

Laboratório de Programação eXtrema

# Refatoração (*Refactoring*)

---

- Uma [pequena] modificação no sistema que não altera o seu comportamento funcional,
- mas que melhora alguma qualidade não-funcional:
  - simplicidade
  - flexibilidade
  - clareza
  - desempenho
- Refatoração (Substantivo) vs. Refatoração (Verbo)

# Exemplos de Refatoração

---

- Mudança do nome de variáveis
- Mudanças nas interfaces dos objetos
- Pequenas mudanças arquiteturais
- Encapsular código repetido em um novo método
- Generalização de métodos
  - `raizQuadrada(float x) ⇒ raiz(float x, int n)`

# Aplicações

---

1. Melhorar código antigo e/ou feito por outros programadores.
2. Desenvolvimento incremental *à la* XP.
  - Em geral, um *passo de refatoração* é tão simples que parece pouco útil.
  - Mas quando se juntam 50 passos, bem escolhidos, em seqüência, o código melhora radicalmente.

# Passos de Refatoração

---

- Cada passo é trivial.
- Demora pouco tempo para ser realizado.
- É uma operação sistemática e óbvia (ovo de Colombo).
- O segredo está em ter um bom vocabulário de *refatorações* e saber aplicá-las criteriosamente e sistematicamente.
- Atualmente, muitas refatorações são **automatizadas** por ferramentas

# Quando Usar Refatoração

---

- Sempre há duas possibilidades:
  1. Melhorar o código existente.
  2. Jogar fora e começar do 0.
- É sua responsabilidade avaliar a situação e decidir quando é a hora de optar por um ou por outro.
- Refatoração é importante para **desenvolvimento e evolução!**



# Catálogo de Refatorações

---

- [Fowler, 2000] contém 72 refatorações.
- Análogo aos padrões de desenho orientado a objetos [Gamma et al. 1995] (GoF).
- Vale a pena gastar algumas horas com [Fowler, 2000].
- (GoF é obrigatório, não tem opção).

# Dica

---

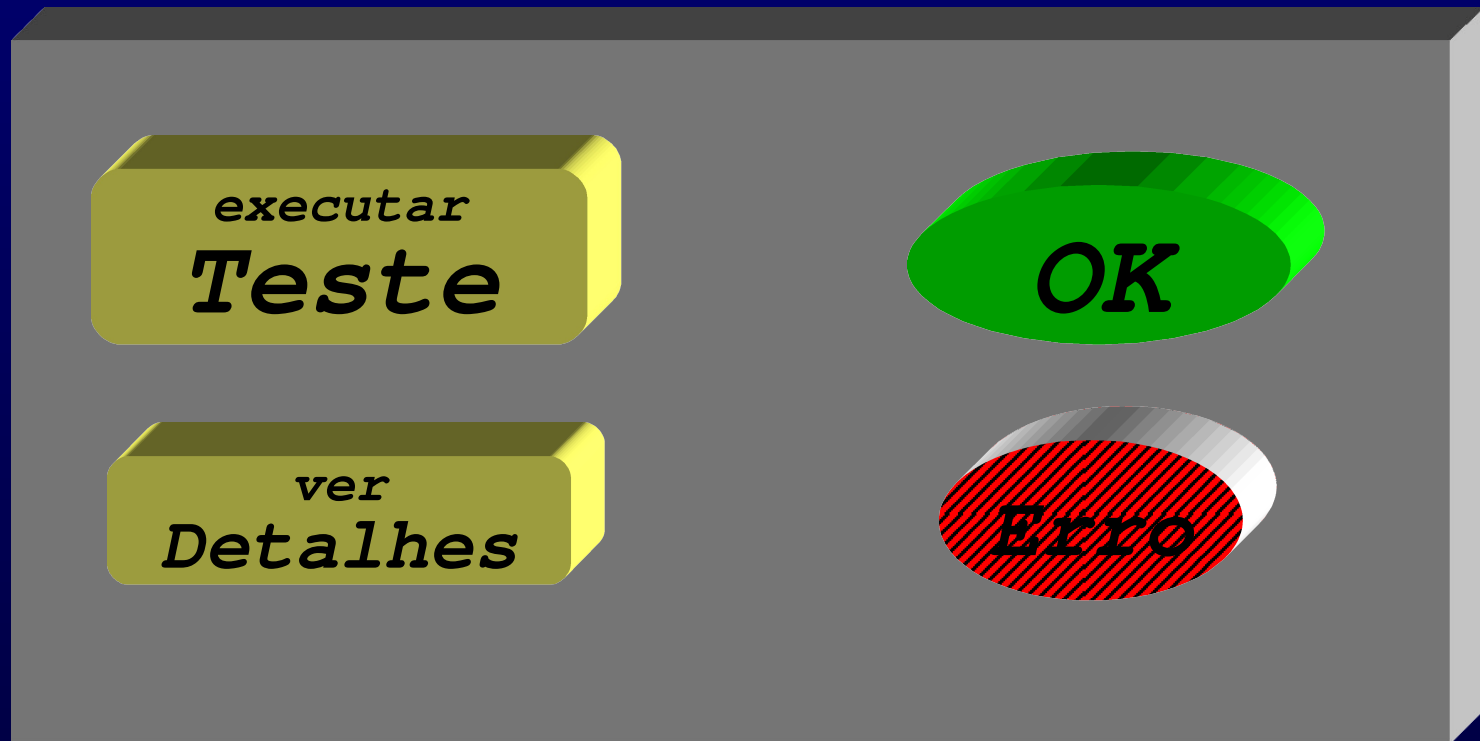
*Quando você tem que adicionar uma funcionalidade a um programa e o código do programa não está estruturado de uma forma que torne a implementação desta funcionalidade conveniente, primeiro refatore de modo a facilitar a implementação da funcionalidade e, só depois, implemente-a.*

# O Primeiro Passo em Qualquer Refatoração

---

- Antes de começar a refatoração, verifique se você tem um conjunto sólido de testes para verificar a funcionalidade do código a ser refatorado.
- Refatorações podem adicionar erros.
- Os testes vão ajudá-lo a detectar erros se eles forem criados.

# O Testador Ideal



# Formato de Cada Entrada no Catálogo

---

- **Nome** da refatoração.
- **Resumo** da situação na qual ela é necessária e o que ela faz.
- **Motivação** para usá-la (e quando não usá-la).
- **Mecânica**, i.e., descrição passo a passo.
- **Exemplos** para ilustrar o uso.

# *Extract Method (110)*

---

- **Nome:** *Extract Method*
- **Resumo:** *Você tem um fragmento de código que poderia ser agrupado. Mude o fragmento para um novo método e escolha um nome que explique o que ele faz.*
- **Motivação:** *é uma das refatorações mais comuns. Se um método é longo demais ou difícil de entender e exige muitos comentários, extraia trechos do método e crie novos métodos para eles. Isso vai melhorar as chances de reutilização do código e vai fazer com que os métodos que o chamam fiquem mais fáceis de entender. O código fica parecendo comentário.*

# *Extract Method (110)*

---

## **Mecânica:**

- Crie um novo método e escolha um nome que explicita a sua intenção (o nome deve dizer o que ele faz, não como ele faz).
- Copie o código do método original para o novo.
- Procure por variáveis locais e parâmetros utilizados pelo código extraído.
  - Se variáveis locais forem usados apenas pelo código extraído, passe-as para o novo método.
  - Caso contrário, veja se o seu valor é apenas atualizado pelo código. Neste caso substitua o código por uma atribuição.
  - Se é tanto lido quando atualizado, passe-a como parâmetro.

## **Compile e teste**

# *Extract Method (110)*

## Exemplo Sem Variáveis Locais

---

```
void imprimeDivida () {
    Enumerate e = _pedidos.elementos ();
    double divida = 0.0;
    // imprime cabeçalho
    System.out.println ("*****");
    System.out.println ("*** Dívidas do Cliente ***");
    System.out.println ("*****");
    // calcula dívidas
    while (e.temMaisElementos ()) {
        Pedido cada = (Pedido) e.proximoElemento ();
        divida += cada.valor ();
    }
    // imprime detalhes
    System.out.println ("nome: " + _nome);
    System.out.println ("divida total: " + divida);
}
```



# *Extract Method (110)*

## Exemplo Com Variáveis Locais

---

```
void imprimeDivida () {
    Enumerate e = _pedidos.elementos ();
    double divida = 0.0;
    imprimeCabecalho ();
    // calcula dívidas
    while (e.temMaisElementos ()) {
        Pedido cada = (Pedido) e.proximoElemento ();
        divida += cada.valor ();
    }
    //imprime detalhes
    System.out.println("nome: " + _nome);
    System.out.println("divida total: " + divida);
}

void imprimeCabecalho () {
    System.out.println ("*****");
    System.out.println ("*** Dívidas do Cliente ***");
    System.out.println ("*****");
}
```

# *Extract Method (110)*

## Exemplo **COM** Variáveis Locais

---

```
void imprimeDivida () {
    Enumerate e = _pedidos.elementos ();
    double divida = 0.0;
    imprimeCabecalho ();
    // calcula dívidas
    while (e.temMaisElementos ()) {
        Pedido cada = (Pedido) e.proximoElemento ();
        divida += cada.valor ();
    }
    imprimeDetalhes (divida);
}

void imprimeDetalhes (double divida)
{
    System.out.println("nome: " + _nome);
    System.out.println("divida total: " + divida);
}
```

# *Extract Method (110)*

## com atribuição

---

```
void imprimeDivida () {  
    imprimeCabecalho ();  
    double divida = calculaDivida ();  
    imprimeDetalhes (divida);  
}  
  
double calculaDivida ()  
{  
    Enumerate e = _pedidos.elementos ();  
    double divida = 0.0;  
    while (e.temMaisElementos ()) {  
        Pedido cada = (Pedido) e.proximoElemento ();  
        divida += cada.valor ();  
    }  
    return divida;  
}
```

# *Extract Method (110)*

## depois de compilar e testar

---

```
void imprimeDivida () {
    imprimeCabecalho ();
    double divida = calculaDivida ();
    imprimeDetalhes (divida);
}

double calculaDivida ()
{
    Enumerate e = _pedidos.elementos ();
    double resultado = 0.0;
    while (e.temMaisElementos ()) {
        Pedido cada = (Pedido) e.proximoElemento ();
        resultado += cada.valor ();
    }
    return resultado;
}
```

# *Extract Method (110)*

## depois de compilar e testar

---

- dá para ficar mais curto ainda:

```
void imprimeDivida () {  
    imprimeCabecalho ();  
    imprimeDetalhes (calculaDivida ());  
}
```

- mas não é necessariamente melhor pois é um pouco menos claro.