Refatoração do Jeito Certo!

"Qualquer tolo consegue escrever código que um computador entenda. Bons programadores escrevem código que humanos possam entender" - **Martin Fowler**

Refatoração: uma modificação feita na estrutura interna do software para deixá-lo mais fácil de compreender e menos custoso para alterar, sem que seu comportamento observável seja alterado

Refatoração VS Reestruturação

Refatoração: Está totalmente ligada à aplicação de pequenos passos que preservam o comportamento, efetuando uma grande mudança por meio do encadeamento de uma sequência desses passos

Reestruturação: Termo genérico para referir a qualquer tipo de reorganização ou de limpeza de uma base de código, refatoração é um tipo particular de reestruturação

POR QUE DEVEMOS REFATORAR?

Refatoração melhora o design do software

- Impede que o design entre em decadência devido a alterações para atingir objetivos de curto prazo
- Reduz código duplicados

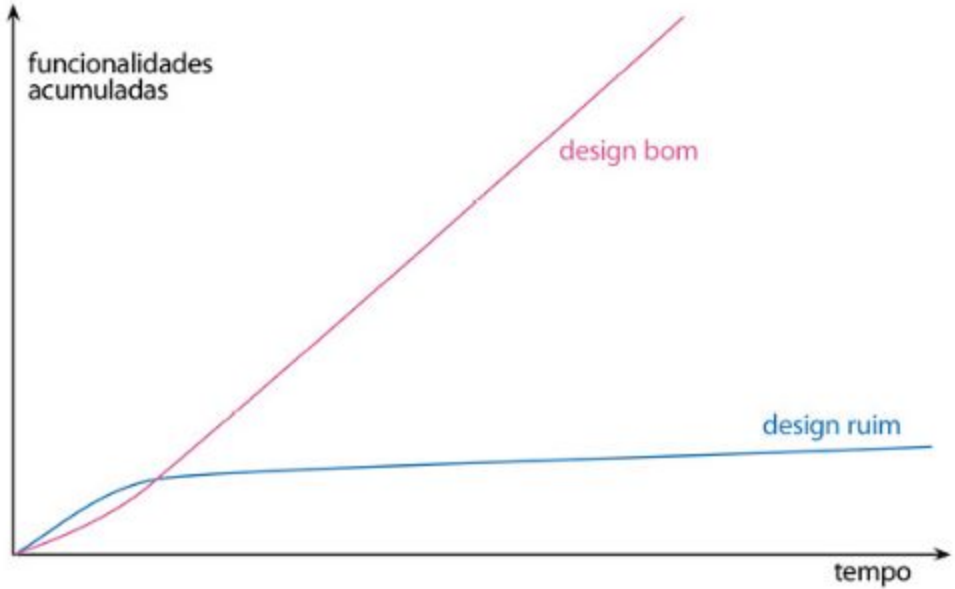
Refatoração deixa o software mais fácil de entender

- Toda informação para entender um código deve estar contida no próprio código

Refatoração ajuda a encontrar bugs

- Ao refatorar entendemos em maiores detalhes o nosso domínio e problemas relacionados
- Quebra pressuposições

Refatoração ajuda a programar mais rapidamente



Refatoração 2nd - Martin Fowler - pg. 74

QUANDO DEVEMOS REFATORAR?

Refatoração preparatória

- Para facilitar o acréscimo de novas funcionalidades

É como se eu quisesse me deslocar 100 km para leste, mas, em vez de ir devagar pela floresta, eu dirigisse 20 km para o norte até a rodovia, e então seguisse 100 km para leste a uma velocidade três vezes maior do que seria possível caso eu tivesse avançado em linha reta. Quando as pessoas estiverem pressionando você para simplesmente seguir em linha reta, às vezes é necessário dizer: ‘Espere, tenho que consultar o mapa e encontrar o caminho mais rápido’. A refatoração preparatória faz isso para mim.
– Jessica Kerr

Refatoração para compreensão

- Devolve o entendimento obtido de volta para o código
- Quando usar algo do sistema que não está muito claro

Refatoração para coleta de lixo

- Quando uma funcionalidade está claro em seu propósito, mas sabemos que poderia ser implementada de uma maneira melhor
- Essa refatoração pode ser demorada, então devemos usar a regra do acampamento:

sempre deixe o local mais limpo do que quando o encontrou

Refatorações planejadas e oportunistas

- As refatorações anteriores são oportunistas, porém não planejadas
- Refatoração não deve ser uma atividade separada/distinta da programação
- Refatoração planejadas devem ser raras
- Devem ser usada em 2 casos:
  - Quando a refatoração vem sendo negligenciada
  - Quando uma área de problema cresce com o tempo apesar de usar a refatoração

Refatoração de longo prazo

- Pessoas dedicadas a refatoração deve ser evitado
- Definir uma área que precisa ser refatorada e sempre que alguém passar por ali deve refatorar

Refatorando em uma revisão de código

- Disseminar o conhecimento entre o time (domínio, programação, refatoração, etc.)
- Ao invés de sugerir melhorias, refatorar
- Utilizar técnicas de pair programing

QUANDO NÃO REFATORAR?

- Código que não precisa ser modificado
- Quando é mais fácil reescrever**

São casos raríssimos e apenas a experiência e conhecimento do problema que poderá determinar qual é o caminho correto.

PROBLEMAS COM A REFATORAÇÃO

Demorando mais para ter novas funcionalidades

- O principal propósito da refatoração é fazer com que programemos mais rápido, agregando mais valor com menos esforço
- Não devemos refatorar apenas para deixar o código mais bonito
- Certas refatorações podem exigir um maior esforço e tempo, por isso precisamos ponderar entre:
  - Refatorar
  - Refatorar parcialmente
  - Não Refatorar
  - Adiar

Dono do código

- Interfaces públicas podem limitar as refatorações, pois precisamos manter a compatibilidade.
- Ao refatorar, pode ser necessário alterar módulos mantidos por outros times

Branches

- Features branches que possuem uma vida longa podem gerar conflitos ao realizar o merge devido a refatorações de outras branches
- O ideal é que seja usado técnicas de CI (Continuous Integration) e toggles de funcionalidades para que as branches sejam atualizadas com maior frequência.

Testes

- A refatoração deve ser feitas em pequenos passos para evitar a introdução de bugs e falhas, porém apenas com testes robustos conseguimos refatorar com segurança e tranquilidade
- Os testes precisam ser executados com frequência, então devem ser extremamente rápidos

Código legado

- Normalmente são complexos, confusos, escritos por outra pessoa e sem testes ou difíceis de testar
- Não existe “*easy way*” aqui, devemos ir com calma e por partes pequenas
- Sempre refatorar onde estiver mexendo
- Se não tiver testes, escrever o teste primeiro

ARQUITETURA E YAGNI

YAGNI - You Aren’t Going to Need It

Arquitetura e Design é algo vivo e mutável. Evolui junto com o software durante o tempo, então não devemos pensar em todas a possibilidades logo de ínicio.

REFATORAÇÃO E DESEMPENHO

- Não especule, faça medições!
- Um código bem fatorado é mais simples para entender onde precisa realmente de otimizações

“MAUS CHEIROS” NO CÓDIGO

Se está cheirando mal, troque-o.
– Vovó Beck

Nome misterioso

- Se encontrar um nome melhor, troque-o
- Se estiver muito complicado de encontrar o nome certo, tente refatorar e simplificar esse trecho de código

Código duplicado

- Tente unificar estruturas de código duplicado
- Evita ter que procurar duplicações ao precisar mudar algo naquele trecho

Funções longas

- Dificultam o entendimento
- Fazem mais de uma coisa
- Extrair funções é a refatoração mais comum neste caso
- Funções pequenas e bem nomeadas retiram a necessidade de olhar sua implementação, assim encontramos facilmente o que procuramos

Lista longa de parâmetros

- Dificulta o uso, costumamos confundir a ordem dos parâmetros
- Usar até 3 parâmetros
- Quando precisar de mais dados, utilizar uma classe/objeto

Dados globais

- Sem garantia de quando e por quem será alterado
- Propenso a gerar bugs

Dados mutáveis

- Cria efeitos colaterais e bugs difíceis de localizar
- Prefira por conceitos de programação funcional

Alteração divergente

- É quando um módulo é frequentemente alterado de formas diferentes e por motivos diferentes
- Princípio do Aberto/Fechado - Aberto para extensão, fechado para modificação

Cirurgia com rifle

- Oposto a alteração divergente
- Sempre que fizer uma mudança em um lugar, precisa fazer várias outras alterações em outros lugares

Inveja de recursos

- É quando uma função ou um módulo gasta mais tempo se comunicando com funções ou dados de outro módulo do que consigo mesmo
- Neste caso devemos unir as funções ou módulo

Agrupamentos de dados

- Conjuntos de dados que andam juntos realmente deveriam ter um lar juntos

Obsessão por primitivos

- Crie tipos que fazem sentido para o seu domínio. Ex; CPF, Telefone

Switches repetidos

- São difíceis de manter
- Sempre que alterar um precisa encontrar os demais
- Fácil para introduzir bugs

Laços

- Não utilizar de forma genérica
- Prefira o uso de pipelines. Ex: map, filter, reduce

Elemento ocioso

- Em algum momento este elemento foi importante
- A partir do momento que deixar de ser relevante, deve ser removido

Generalidade especulativa

- Adição de flexibilidade na expectativa de que será necessário em algum momento
- Se apenas os testes fazem uso, remova

Campo temporário

- Uma classe deve precisar de todos os seus atributos
- Se algum atributo é são usado de forma condicional, ele provavelmente deveria estar em outro lugar

Cadeia de mensagem

- É quando um cliente pede um objeto para outro objeto, cujo cliente então pede para outro objeto, cujo cliente então pede para outro objeto, e assim por diante
- Gera acoplamento

Intermediário

- Classes anêmicas
- Não tem nenhuma real função a não ser chamar alguma outra ponta

Ex: Marcar a reunião com um diretor de uma empresa

Classe grande

- Conhecidas como God Class, possuem muitas responsabilidades
- Tendem a possuir duplicações e dados desnecessários

Classes alternativas com interfaces diferentes

- Impedem o uso da substituição
- Padronize as interfaces

Comentários

- Comentários geralmentes são formas de expressar o que o programador não foi capaz de comunicar através do código

LET'S PRACTICE