universidade de aveiro

Tópico B

REFATORAÇÃO

Padrões e Desenho de Software

Mário Silva (93430)

Índice

Refatoração

Introdução/Descrição

A refatoração é uma técnica disciplinada para reestruturar um corpo de código existente, alterando a sua estrutura interna sem alterar o comportamento externo. Torna o código mais simples e fácil de estender, tornando assim mais rápido de construir sistemas complicados.

Na sua essência é uma série de pequenas transformações a um dado código inicial com o objetivo de o tornar mais limpo e com um design mais simples. Cada transformação (chamada de "refatoração") faz pouco, mas uma sequência dessas transformações pode produzir uma reestruturação significativa. Como cada refatoração é pequena, é menos provável que cause erros no código. O sistema é mantido totalmente funcional após cada refatoração, reduzindo as chances de um sistema ser seriamente quebrado durante a reestruturação.

Nomeação de variáveis inadequada, classes e métodos inchados, números mágicos, isso torna o código desleixado e difícil de entender. No entanto, quando um sistema de software é bem-sucedido, há sempre a necessidade de aprimorá-lo, corrigir problemas e adicionar novos recursos. Mas a natureza de uma base de código faz uma grande diferença em como é fácil fazer essas alterações. Muitas vezes, as funcionalidades são aplicadas umas sobre as outras de uma maneira que torna cada vez mais difícil fazer alterações. Para combater isto, é importante refatorar o código para que aprimoramentos adicionais não levem a complexidade desnecessária. Um código limpo é mais fácil e barato de manter.

A refatoração faz parte da programação diária, é uma parte regular da atividade de programação. Quando necessário adicionar um novo recurso a uma base de código, analisa-se o código existente e considera-se se ele está estruturado de maneira a tornar a nova alteração direta. Caso contrário, altera-se o código existente para facilitar essa nova adição. Alterar imediatamente, é geralmente mais rápido do que realizar a refatoração só no fim (depende da situação e também pode variar tendo em conta da opinião de cada programador).

Code Smells

Quando se está a fazer a refatoração muitas vezes está-se a procurar aquilo que se chama de bad smells ou problemas de design comuns. Um mau design é normalmente quando o código é demasiado complicado, pouco claro ou duplicado.

Existem muitos bad smells que poderia explorar, mas para este trabalho não me vou focar muito a explicá-los, apenas vou enumerá-los:

- Nomes de variáveis impercetíveis.
- Código duplicado.
- Funções demasiado longas.
- Exposição de classes indecente.
- Classes longas.
- Combinação excessiva de métodos desnecessária.
- Demasiados parâmetros.
- Dados globais.
- Uso excessivo de primitivas.

Entre muitos outros…

- Declarações *if* demasiado complexas e impercetíveis.

Dívida técnica:

Consegue-se aumentar a velocidade de programação sem fazer testes e sem fazer refatoração ao código, contudo vai-se acumulando uma dívida técnica que gradualmente irá ter um impacto e diminuirá a velocidade de programação até que a dívida fique "paga".

Principais causas de acumulação de dívida técnica:

- Pressão para terminar o código.
- Falta de conhecimento sobre as consequências da dívida técnica.
- Falta de testes.
- Falta de documentação (por exemplo, comentários).
- Atraso da refatoração.
- Falta de comunicação entre os membros da equipa.
- Desenvolvimento simultâneo de longo prazo em vários ramos.

Razões para refatorar código:

- Refatorar para que quando o próprio ou outra pessoa retornar a esse código, não precise gastar tempo a tentar entendê-lo.
- Cada vez que seja precisar fazer uma alteração em um código duplicado, o programador tem de lembrar-se de fazer a mesma alteração em todas as instâncias. Isso aumenta a carga cognitiva e diminui o progresso.
- Remove código em excesso tornando mais fácil de perceber e modificar.
- Ao fazer refatoração conseguimos perceber melhor o código do sistema e torna-se mais fácil de encontrar bugs.
- Torna-se mais rápido de escrever um código bem estruturado e percetível devido a todas as razões apresentadas.

Quando fazer refatoração

- Ao adicionar funcionalidades.
- Ao corrigir bugs.
- Ao rever o código.

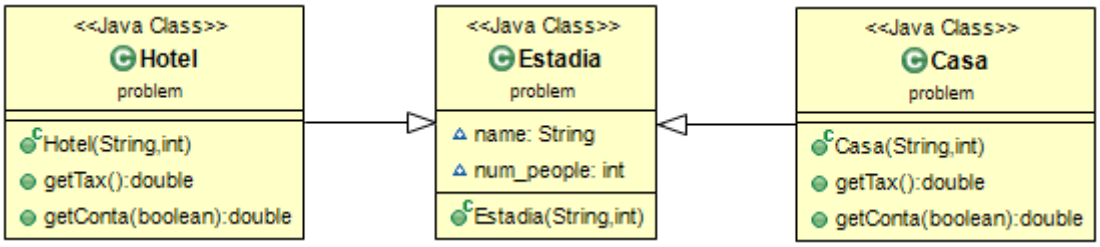
Verificações a fazer enquanto se faz refatoração

- O Código tem de facto ficar mais limpo.
- Não devem ser adicionadas novas funcionalidades durante a refatoração. Não convém misturar refatoração com desenvolvimento de novas funcionalidades de forma a não perder o raciocínio que é necessário fazer na refatoração e evitar possivelmente ter de refatorar o código logo a seguir outra vez. Às vezes pode ser necessário reescrever por completo partes de código se estiver escrito muito desleixadamente.
- É importante também verificar que os testes que existiam passam todos depois da refatoração (tendo a certeza que os testes estão de facto corretos).

Problema

As subclasses são desenvolvidas em paralelo, às vezes por pessoas diferentes, o que leva à duplicação de código, erros e dificuldades na manutenção do código, pois cada alteração deve ser feita em todas as subclasses.

As subclasses **Hotel** e **Casa** são estendidas da classe **Estadia** e implementam algoritmos que contêm operações semelhantes.



Para a classe Casa temos o método *getTax()*:

```
public double getTax() {
    return num_people*0.875;
}
```

E para a classe Hotel:

```
public double getTax() {
    return num_people*0.925;
}
```

E em ambas temos o método *getConta()*:

```
public double getConta(boolean discount){
    if (num_people>=5 && discount == true) {
        return num_people*getTax()*0.8;
    } else if (num_people>=5 && discount == false) {
        return num_people*getTax();
    } else if (num_people<5 && discount == true) {
        return num_people*getTax()*0.9*0.8;
    } else if (num_people<5 && discount == false) {
        return num_people*getTax()*0.9;
    }
    return 0;
}
```

Solução

Primeiro de tudo, vamos aplicar uma boa prática para o método *getConta()* que como podemos observar é uma declaração *if* demasiado grande e não muito percetível, daí a necessidade de refatorá-lo.

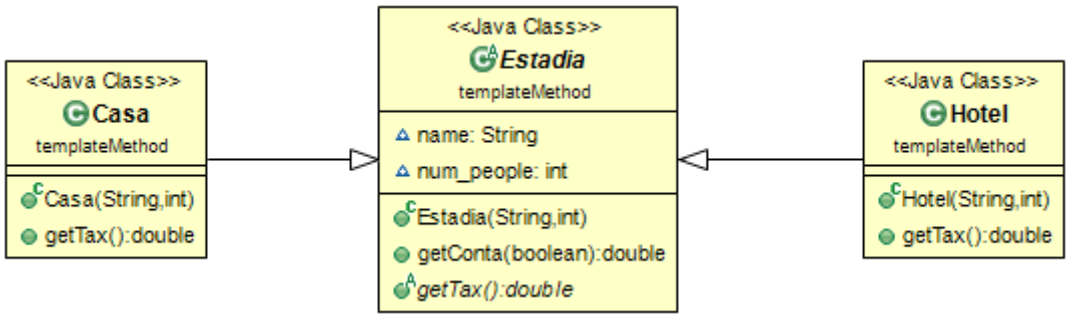
Para isso vou colocar as condições mais complexas numa variável temporal com um nome muito mais explícito e fácil de perceber, e também visto que a conta efetuada tem em todas as opções *num_people*getTax()*, coloquei esse valor numa variável temporária inicial.

```
public double getConta(boolean discount){
    final boolean moreThanFourWithDiscount = ( num_people>=5 && discount == true );
    final boolean moreThanFourWithoutDiscount = ( num_people>=5 && discount ==false);
    final boolean lessThanFiveWithDiscount = ( num_people<5 && discount == true );
    final boolean lessThanFiveWithoutDiscount = ( num_people<5 && discount == false);
    double conta = num_people*getTax();

    if (moreThanFourWithDiscount) {
        conta *= 0.8;
    } else if (moreThanFourWithoutDiscount) {
        return conta;
    } else if (lessThanFiveWithDiscount) {
        conta *= 0.9*0.8;
    } else if (lessThanFiveWithoutDiscount) {
        conta *= 0.9;
    }
    return conta;
}
```

Agora para tratar da solução, temos de mover a estrutura do algoritmo e métodos idênticos para uma superclasse e deixar as implementações diferentes nas subclasses respetivas, ou seja, mover o método *getConta()* para a superclasse **Estadia**.

Como também existem métodos diferentes como o *getTax()*, decidi tornar a classe **Estadia** abstrata para poder mover o método para a superclasse, colocando-o como abstrato, e deixar a implementação nas subclasses respetivas.



Benefícios da Refatoração:

- Ao criar um método de Template consegue-se eliminar a duplicação de código ao agrupar o conjunto de operações partilhadas numa classe superior e deixando apenas as diferenças nas subclasses.
- A utilização de um padrão de comportamento adequado, neste caso o Template Method, é bastante útil, quando uma nova versão do algoritmo for desenvolvida, apenas é preciso criar uma nova subclasse; nenhuma alteração no código existente é necessária, é, portanto, mais facilmente extensível.

Referências

https://refactoring.guru/refactoring

https://refactoring.com/

https://pt.wikipedia.org/wiki/Refatora%C3%A7%C3%A3o

https://sourcemaking.com/refactoring/decompose-conditional

https://www.altexsoft.com/blog/engineering/code-refactoring-best-practices-when-and-when-not-to-do-it/