DES-SIS-II - A2 Texto de apoio

Site: EAD Mackenzie
Tema: DESENVOLVIMENTO DE SISTEMAS II {TURMA 03B} 2023/1
Livro: DES-SIS-II - A2 Texto de apoio

Impresso por: FELIPE BALDIM GUERRA .
Data: quarta, 1 mar 2023, 06:52

Descrição

Índice

1. REFATORAÇÃO

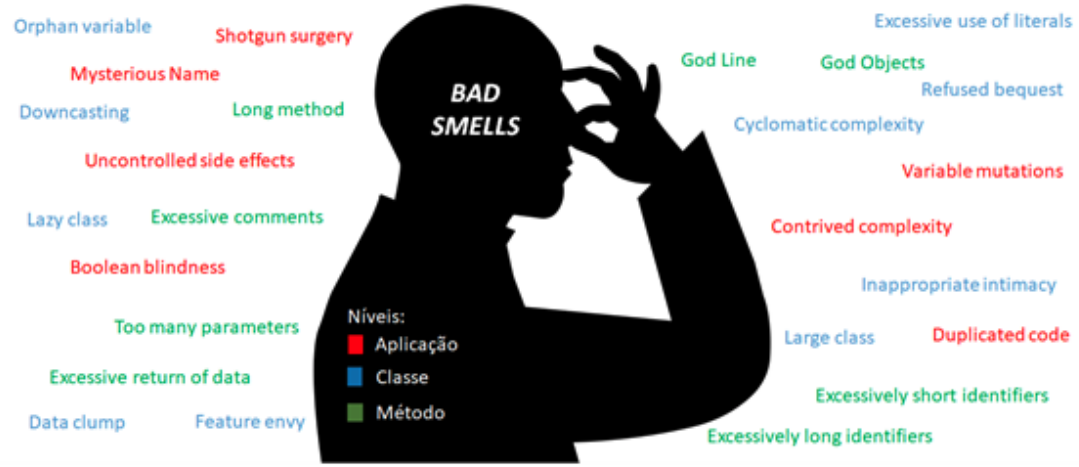
Refatoração – O que é?

A refatoração é um conjunto de técnicas, baseadas em boas práticas de desenvolvimento de software, que têm por objetivo reestruturar um código já existente, alterando sua estrutura interna (ou seja, a organização do código) ao mesmo tempo em que mantém seu comportamento externo (o resultado do código deve seguir sendo o mesmo).

Inicialmente, as técnicas de refatoração de código foram apresentadas por Martin Fowler em seu livro Refactoring – Improving the Design of Existing Code , apresentando um extenso catálogo de técnicas para melhoria de código. Cada uma dessas técnicas é bastante pontual e gera pequenas modificações no código. Uma refatoração sozinha não impacta (ou não deveria impactar) significativamente o desempenho, a manutenibilidade, o potencial de reuso ou a clareza do código. Porém, a aplicação de diversas técnicas em um código pode melhorar sobremaneira esses aspectos.

Em seu livro, Fowler popularizou a expressão bad smell (“mau cheiro”) – criada anteriormente por Kent Beck – que se refere a estruturas no código que necessitam de ser refatoradas.

> Mas, atenção! Bad smells não são bugs ou erros no código, que impeçam seu funcionamento correto e, sim, problemas no design do código que podem afetar seu desempenho, sua clareza, a capacidade de realizar a manutenção ou expandir o código – ou, também, aumentar a possibilidade de bugs no futuro.



Fonte: Elaborada pelo autor.

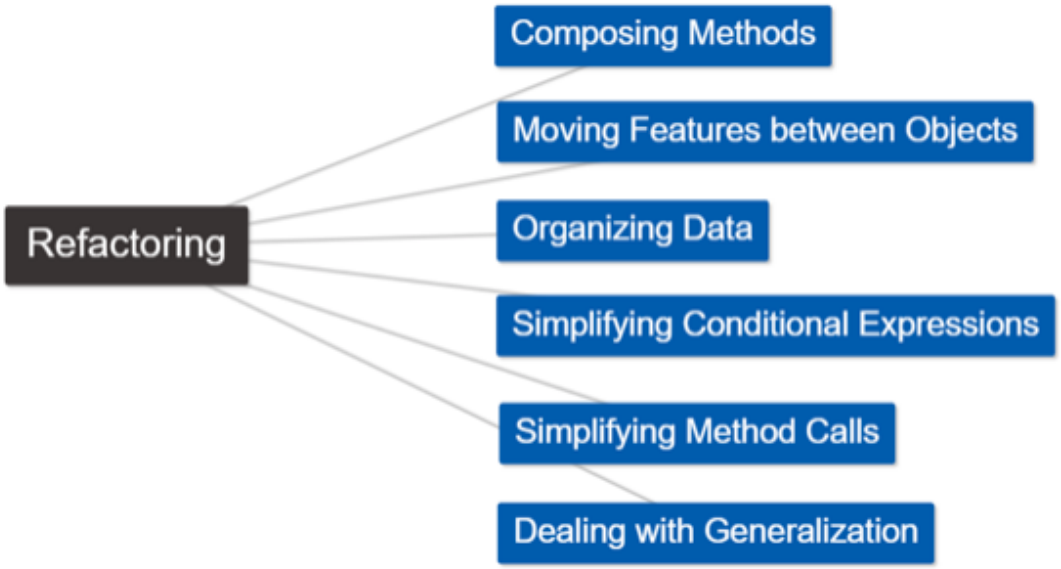
> Não se preocupe em decorar os nomes dos bad smells e tampouco das técnicas de refatoração. Com um pouco de prática, você aprenderá a identificá-las.

Como a quantidade de técnicas de refatoração (e bad smells correspondentes) apresentadas por Fowler é bastante grande, neste material, iremos dividi-las em "tipos" de refatoração – de acordo com seu propósito (esta divisão é proposta pelo site Refactoring Guru, indicado como material complementar).

> Serão mantidos os nomes originais em inglês de todas as técnicas e seus tipos (assim como das bad smells), para evitar mal-entendidos devido às diferentes traduções possíveis.

Técnicas de Refatoração

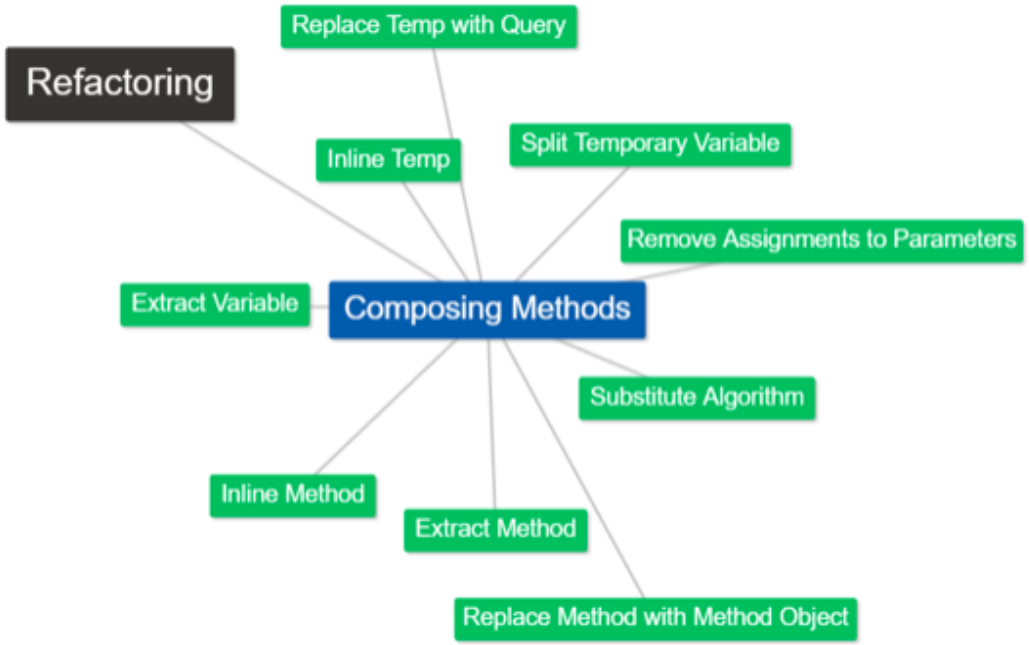
As técnicas de refatoração descrevem os passos necessários para a modificação de código. Mas atenção: toda técnica de refatoração tem seus prós e contras! Dessa maneira, é papel do desenvolvedor de software avaliar a aplicação de cada técnica e saber avaliar seu impacto. Vamos dividi-las aqui em seis categorias e, para cada categoria, apresentaremos ao menos um exemplo:



Fonte: Elaborada pelo autor.

Técnicas de Refatoração – Composing Methods

Esta categoria de técnicas de refatoração diz respeito a como escrever adequadamente um método. É bastante frequente encontrar métodos excessivamente longos ou com alguns detalhes de implementação que necessitam de passar por refatoração. A seguir, estão as técnicas de refatoração previstas no catálogo de Martin Fowler que entram nesta categoria:



Fonte: Elaborada pelo autor.

Extract Method
Problema Existe um fragmento de código que pode ser agrupado. *O bad smell associado é o Long Method – ou seja, se você encontrar algum método muito longo, pode ser que haja fragmentos de código que podem ser agrupados em um método.*

Solução
Mova este código para um novo método separado e substitua o código antigo por uma chamada para o método.

Considere um sistema de compra e venda de uma farmácia que tem o seguinte método para impressão de nota fiscal:

```
//CÓDIGO SEM REFATORAR
public void imprimeNF(){
        System.out.println("FARMÁCIA REFATORADA");
        System.out.println("Nota Fiscal");
        System.out.println("CNPF: 998.998.998-98");
        //... imprime dados da compra
        System.out.println("Obrigado! Volte Sempre!");
        System.out.println("Data da compra: "+(new
Date()).toString());
    }
```

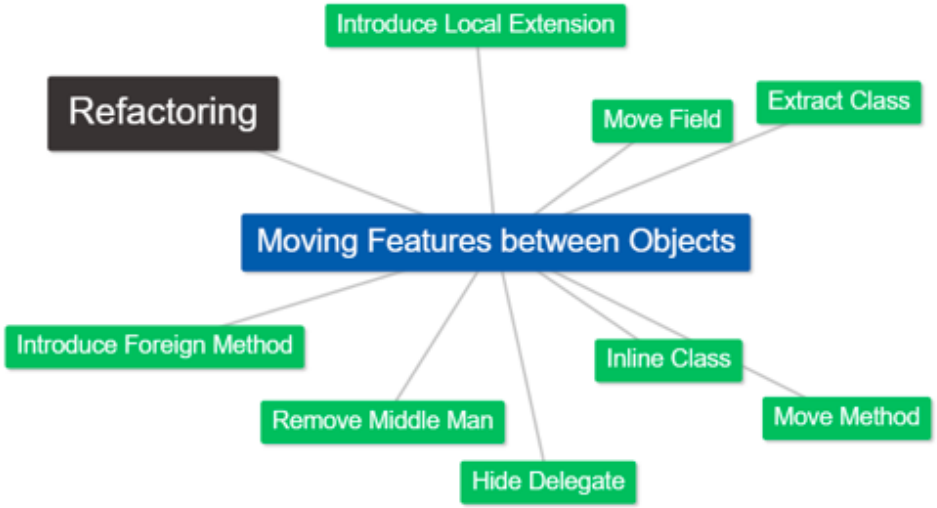
Note que as primeiras linhas do método, assim como as últimas linhas, não têm muita relação com a compra em si: elas representam linhas de cabeçalho ou rodapé que, inclusive, podem ser úteis em outras situações. Neste caso, o ideal seria que essas funcionalidades fossem isoladas em métodos próprios:

```
//CÓDIGO COM REFATORAÇÃO DO TIPO "EXTRACT METHOD"
public void imprimeCabeçalho(){
        System.out.println("FARMÁCIA REFATORADA");
        System.out.println("Nota Fiscal");
        System.out.println("CNPF: 998.998.998-98");
    }

public void imprimeRodapé(){
        System.out.println("Obrigado! Volte Sempre!");
        System.out.println("Data da compra: "+ (new
Date()).toString());
    }

public void imprimeNF(){
        imprimeCabeçalho();
        //... imprime dados da compra
        imprimeRodapé();
    }
```

Leia mais em: <https://refactoring.guru/refactoring/techniques/composing-methods>.

Técnicas de Refatoração – Movimentando Funcionalidades Entre Objetos

Essas técnicas dizem respeito a como mover com segurança uma funcionalidade entre classes, criar novas classes e ocultar detalhes de implementação do acesso público. As técnicas desta categoria são:



Fonte: Elaborada pelo autor.

Vejamos um exemplo da técnica Extract Class:

Extract Class
Problema Quando uma classe faz o trabalho de duas, isso impacta sua coesão. **Solução** Crie uma classe e coloque nela os campos e métodos responsáveis pela funcionalidade a ser separada.

Por exemplo, em um sistema no qual há o cadastro de clientes, observe a seguinte classe:

```java
//CÓDIGO SEM REFATORAÇÃO

class Cliente

{

    private String nome, cpf, logradouro, compl, bairro, cidade,
estado, cep;

    private int num;


    public void printDados() {

        System.out.println("Nome: "+nome);

        System.out.println("CPF: "+cpf);

        System.out.println("Endereço: ");

        System.out.println(logradouro + num + compl);

        System.out.println("Bairro:" + bairro);

        System.out.println(cidade + estado + cep);

    }

}
```

Note que a classe Cliente acumula funcionalidades de armazenar e exibir os dados do cliente e de seu endereço. Os dados de endereço, especificamente, poderiam ser isolados em outra classe (o que aumenta seu potencial de reuso, já que esta classe Endereco pode ser reutilizada para outros fins – endereço de fornecedores, de empresas etc.). Veja o código refatorado:

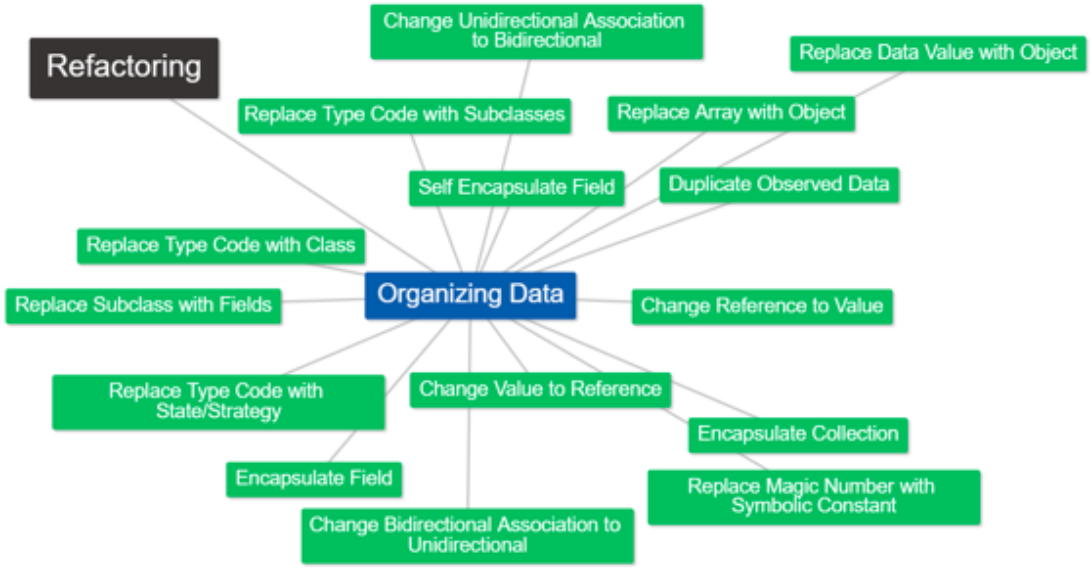
```java
//CÓDIGO COM REFATORAÇÃO DO TIPO "EXTRACT CLASS"
class Endereco
{
    private String logradouro, compl, bairro, cidade, estado, cep;
    private int num;

    public void printDados() {
        System.out.println("Endereço: ");
        System.out.println(logradouro + num + compl);
        System.out.println("Bairro:" + bairro);
        System.out.println(cidade + estado + cep);
    }
}


class Cliente
{
    private String nome, cpf;
    private Endereco end;

    public void printDados() {
        System.out.println("Nome: "+nome);
        System.out.println("CPF: "+cpf);
        end.printDados();
    }
}
```

Leia mais em: <https://refactoring.guru/refactoring/techniques/moving-features-between-objects >.

Faça, agora, a primeira atividade Praticando, disponível no ambiente virtual.

Técnicas de Refatoração – Organizando Dados

As técnicas nesta categoria estão relacionadas com manipulação adequada de dados nas classes, de maneira a torná-las mais reutilizáveis. As técnicas desta categoria são mostradas a seguir



Fonte: Elaborada pelo autor.

Vejamos um exemplo com a técnica Replace Array with Object:

Replace Array with Object
Problema O código tem uma matriz que contém vários tipos de dados. **Solução** Substitua a matriz por um objeto que terá atributos separados para cada elemento.

Observe a seguinte implementação de um catálogo de filmes em um serviço de streaming:

```
class CatalogoFilmes
{
    private String [][] catalogo = {
        {"Fale com ela","Pedro Almodóvar","Espanha","2002"},
        {"Asas do Desejo","Wim Venders","Alemanha", "1987"},
        {"Central do Brasil","Walter Salles","Brasil","1998"}
        /*mais filmes*/
    };
}
```

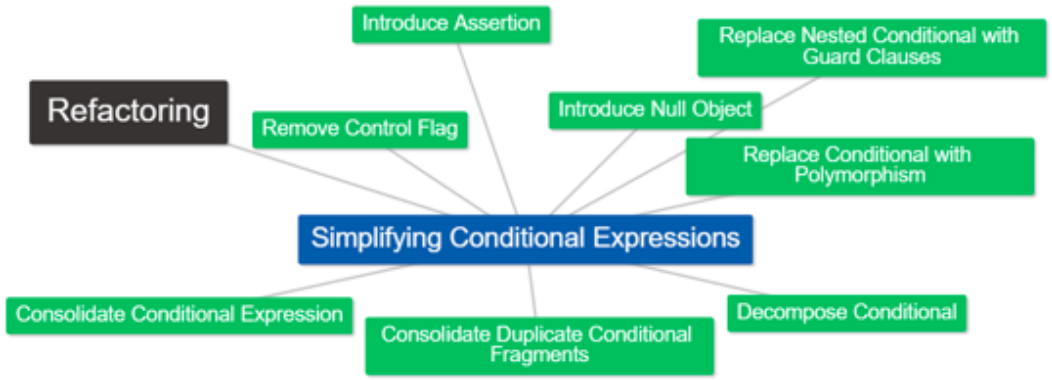
Note que é uma matriz bidimensional, com n elementos em que cada um deles tem quatro outros elementos que representam os dados de um filme. Observe o código refatorado:

```
//CÓDIGO COM REFATORAÇÃO DO TIPO "REPLACE ARRAY WITH OBJECT"
class Filme
{
    String nome, diretor, país;
    int ano;
    public Filme (String nome, String diretor, String país, int ano){
        this.nome = nome;
        this.diretor = diretor;
        this.país = país;
        this.ano = ano;
    }
}



class CatalogoFilmes
{
    private Filme [] catalogo = {
        new Filme("Fale com ela","Pedro Almodóvar","Espanha",2002),
        new Filme("Asas do Desejo","Wim Venders","Alemanha", 1987),
        new Filme("Central do Brasil","Walter Salles","Brasil",1998)
        /*mais filmes*/
    };
}
```

Veja que, com a aplicação desta técnica, os dados ficam bem mais organizados e permitem, inclusive, tipos diferentes (note que o atributo ano pôde ser representado como int e não mais como String).

Técnicas de Refatoração – Simplificando Expressões Condicionais

As técnicas deste grupo tentam eliminar longas expressões condicionais (if – then – else ou switch, por exemplo), que são difíceis de manter e expandir.



Fonte: Elaborada pelo autor.

> Este grupo de técnicas de refatoração tem uma técnica chamada Replace Conditional with Polymorphism que será aprofundada na Aula 4, a respeito de GRASP.

Vejamos um exemplo com a técnica Introduce Null Object :

Introduce Null Object
Problema O código tem métodos que retornam null (ou valores similares) em vez de objetos reais, o que leva à necessidade de verificações de null frequentes ao longo do código.

Solução

Em vez de null, retorne um Null Object que encapsule o comportamento padrão.

Observe o seguinte código desta situação-exemplo de uma venda de ingressos para teatro em que os clientes podem levar um acompanhante, cujo desconto pode ser de 25%, para menores de 18 anos, ou de 10%:

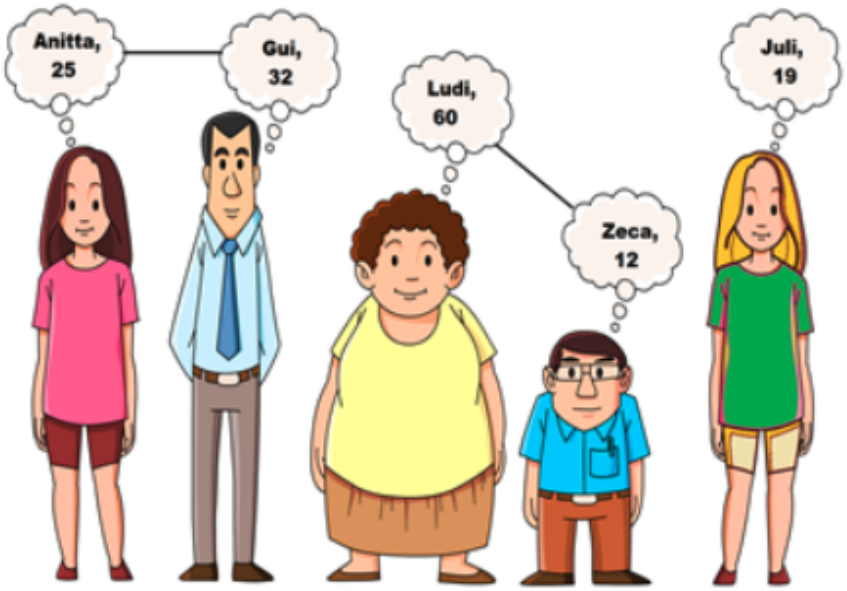
```java
//CÓDIGO ANTES DA REFATORAÇÃO
abstract class Pessoa
{
    protected String nome, cpf;
    public Pessoa (String nome, String cpf){
        this.nome = nome;
        this.cpf = cpf;
    }
}


class Acompanhante extends Pessoa
{
    private int idade;
    public Acompanhante (String nome, String cpf, int idade){
        super(nome, cpf);
        this.idade = idade;
    }
    public double getDesconto(){
        return (idade < 18) ? 0.25 : 0.1;
    }
}


class Cliente extends Pessoa
{
    Acompanhante a;
    public Cliente (String nome, String cpf, Acompanhante a){
        super(nome, cpf);
        if (a!=null) this.a = a;
    }

    public double calculaValorIngresso(double valorUnit){
        if (a!=null) return valorUnit;
        return valorUnit + (1-a.getDesconto())*valorUnit;
    }
}
```

Perceba que houve a necessidade de fazer dois testes quanto à nulidade do objeto Acompanhante. Veja como fica com a aplicação da técnica de refatoração Null Object:

```
//CÓDIGO COM REFATORAÇÃO DO TIPO "NULL OBJECT"
//MANTÉM AS CLASSES PESSOA E ACOMPANHANTE
class AcompanhanteNulo extends Acompanhante
{
    public AcompanhanteNulo(){
        super("","",-1);
    }
    public double getDesconto(){
        return 0;
    }
}


class Cliente extends Pessoa
{
    Acompanhante a;
    public Cliente (String nome, String cpf, Acompanhante a){
        super(nome, cpf);
        this.a = a;
    }

    public double calculaValorIngresso(double valorUnit){
        return valorUnit + (1-a.getDesconto())*valorUnit;
    }
}
```

Veja que não é mais necessário fazer nenhuma verificação – as condicionais foram retiradas da classe Cliente !

Como usar essas classes e métodos? Imagine uma peça de teatro cujo ingresso custe R$ 50,00 e temos cinco pessoas organizadas da seguinte maneira: Anitta, que é cliente e tem o Gui como acompanhante; Ludi, que é cliente e tem o Zeca como acompanhante; e Juli, que é uma cliente que vai assistir à peça sozinha.



Fonte: Elaborada pelo autor.

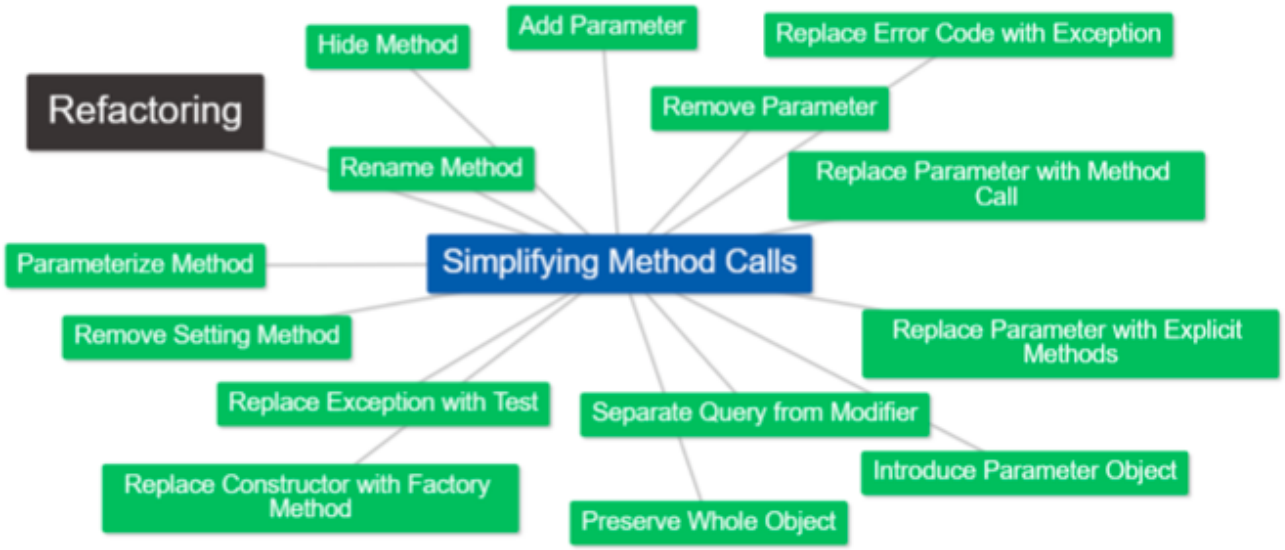
O fragmento de código que pode ser criado aplicando a refatoração Null Object é:

```
Cliente c1 = new Cliente("Anitta","12345678",
                 new Acompanhante("Gui","23456789",32));
Cliente c2 = new Cliente("Ludi","34567891",
                 new Acompanhante("Zeca","45678912",12));
Cliente c3 = new Cliente("Juli","56789123", new AcompanhanteNulo());
c1.calculaValorIngresso(50.0); //Valor total: R$ 90,00
c2.calculaValorIngresso(50.0); //Valor total: R$ 78,50
c3.calculaValorIngresso(50.0); //Valor total: R$ 50,00
```

Perceba a consistência na chamada dos métodos, com ou sem acompanhante. Note também que a idade do Cliente é irrelevante para as regras de cálculo do sistema, diferente da idade do *Acompanhante*. Faça, agora, a segunda atividade Praticando disponível no ambiente virtual.

Técnicas de Refatoração – Simplificando Chamadas de Métodos

Neste grupo de técnicas, encontram-se aquelas cujo propósito é simplificar os métodos de maneira a se tornarem mais coesos e com maior potencial de reuso. A seguir, temos a lista de técnicas desta categoria, presentes no catálogo de Fowler:



Fonte: Elaborada pelo autor.

Vejamos um exemplo com a técnica de refatoração Separate Query from Modifier :

Separate Query from Modifier
Problema O código possui um método que retorna um valor, mas também altera algum atributo do objeto. **Solução** Divida o método em dois métodos separados: um que retorna o valor e o outro que modifica o atributo do objeto.

Observe este código. Imagine que, em uma aplicação bancária, você tem métodos em uma classe *ContaCorrente* que executam saque e depósito de dinheiro. Veja o código antes da refatoração:

```
//CÓDIGO ANTES DA REFATORAÇÃO

class ContaCorrente
{
    private double saldo;

    public double saque (double valor){
        saldo -= valor;
        return saldo;
    }

    public double depósito (double valor){
        saldo += valor;
        return saldo;
    }
}
```

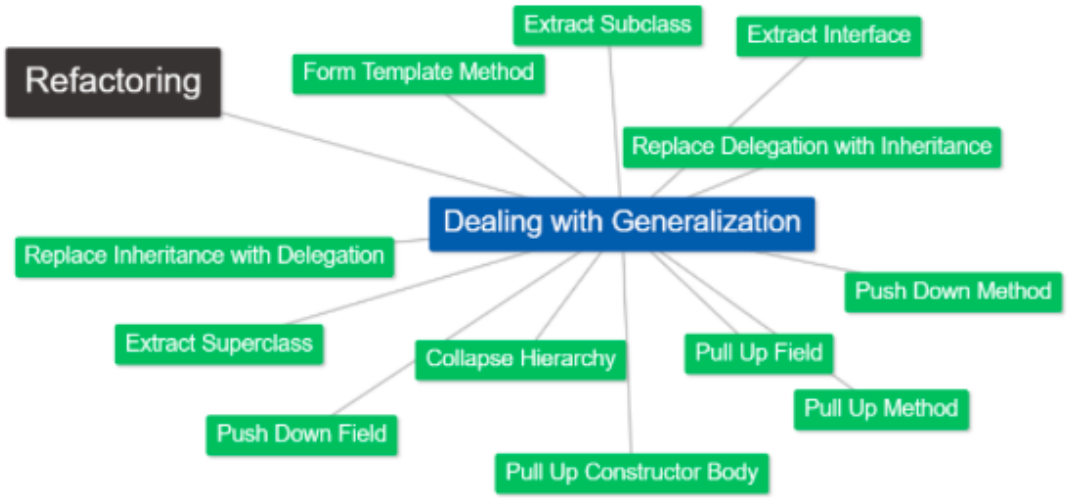
Veja que são dois métodos (modifiers) que realizam modificações no atributo saldo e, ao mesmo tempo, funcionam como métodos de consulta (query) ao saldo. De acordo com esta técnica de refatoração, o código refatorado ficaria assim:

```
//CÓDIGO COM REFATORAÇÃO DO TIPO "SEPARATE QUERY FROM MODIFIER"

class ContaCorrente
{
    private double saldo;

    public void saque (double valor){
        saldo -= valor;
    }

    public void depósito (double valor){
        saldo += valor;
    }

    public double consulta (){
        return saldo;
    }
}
```

Agora, as funcionalidades de modificação estão separadas da consulta, aumentando as possibilidades de reutilização de todas elas.

Técnicas de Refatoração – Lidando com Generalização

Neste último (mas não menos importante) grupo de técnicas, estão aquelas relacionadas com a abstração de classes. Segue a lista de técnicas de refatoração que entram nesta categoria:



Fonte: Elaborada pelo autor.

Vejamos um exemplo com a técnica Extract Superclass:

Extract Superclass
Problema O código tem duas ou mais classes com atributos e métodos comuns. **Solução** Crie uma superclasse para essas classes e mova para ela todos os atributos e métodos idênticos.

Analisemos um código como exemplo. Imagine uma padaria que venda produtos frescos produzidos ali (pães, bolos etc.) e também alguns produtos industrializados (manteiga, leite etc.). Veja um primeiro código antes de ser refatorado:

```
//CÓDIGO ANTES DA REFATORAÇÃO
class ProdutoFresco
{
    private String descrição;
    private double peso, preço_kg;
    private Calendar data_fabricação;
    public double getPreço(){
        return peso*preço_kg;
    }
    public String getDescrição(){
        return descrição;
    }

     public String getDados(){
         String temp = "Preço por kg: "+ preço_kg +"\n";
         temp += "Peso: "+ peso + "\n";
         temp += "Fabricação: " + data_fabricação.toString() + "\n";
         temp += "Validade: 1 dia";
         return temp;
     }
 }
```

```
class ProdutoIndustrializado
{
    private String descrição;
    private double preço;
    private Calendar data_validade;
    public double getPreço(){
        return preço;
    }
    public String getDescrição(){
        return descrição;
    }
    public String getDados(){
        String temp = "Preço: "+ preço +"\n";
        temp += "Validade: " + data_validade.toString();
        return temp;
    }
}
```

Veja, no código, vários atributos e métodos em comum – ou o método inteiro, como *getDescrição* ( ), ou uma responsabilidade que precisa ser implementada, ainda que cada classe implemente de uma forma, como *getPreço*( ) e *getDados*( ).

Observe o código após a refatoração usando a técnica Extract Superclass:

```
//CÓDIGO APÓS A REFATORAÇÃO COM "EXTRACT SUPERCLASS"

abstract class Produto
{
    protected String descrição;
    protected double preço;
    protected Calendar data;

    public String getDescrição(){
        return descrição;
    }
    public abstract double getPreço();
    public abstract String getDados();
}

class ProdutoFresco extends Produto
{

    private double peso;
    public double getPreço(){
        return peso*preço;
    }

    public String getDados(){
        String temp = "Preço por kg: "+ preço +"\n";
        temp += "Peso: "+ peso + "\n";
        temp += "Fabricação: " + data.toString() + "\n";
        temp += "Validade: 1 dia";
        return temp;
    }
}
```

```
class ProdutoIndustrializado extends Produto
{
    public double getPreço(){
        return preço;
    }

    public String getDados(){
        String temp = "Preço: "+ preço +"\n";
        temp += "Validade: " + data.toString();
        return temp;
    }
}
```

Note que foi criada uma classe Produto abstrata (ou seja, que não pode ser instanciada) para reunir os elementos em comum das outras duas (ProdutoFresco e ProdutoIndustrializado). Os atributos similares de ambas as classes foram para esta nova classe, bem como o método com comportamento idêntico em ambas – getDescrição ( ). Os demais métodos, que são necessários, mas que se comportam de forma diferente em cada uma das subclasses, são declarados como abstratos na classe-mãe e implementados nas classes-filhas.

Em resumo, há muitas técnicas de refatoração e todas devem ser aplicadas com bastante atenção em relação a seus impactos – afinal, o princípio básico é manter a funcionalidade do código, alterando sua estrutura visando a uma melhor manutenibilidade, possibilidade de reuso e de extensão.

Além do catálogo original de Martin Fowler, outros catálogos foram surgindo. Hoje em dia, podemos aplicar o termo refatoração a outras etapas do desenvolvimento de software, como refatoração de projetos (baseando-se em Padrões de Projeto, a serem estudados nas Aulas 5 a 7, por exemplo) e mesmo de arquitetura.

> Navegue pelos sites <https://refactoring.com/catalog/> e <https://refactoring.guru/refactoring> para conhecer as demais técnicas de refatoração.