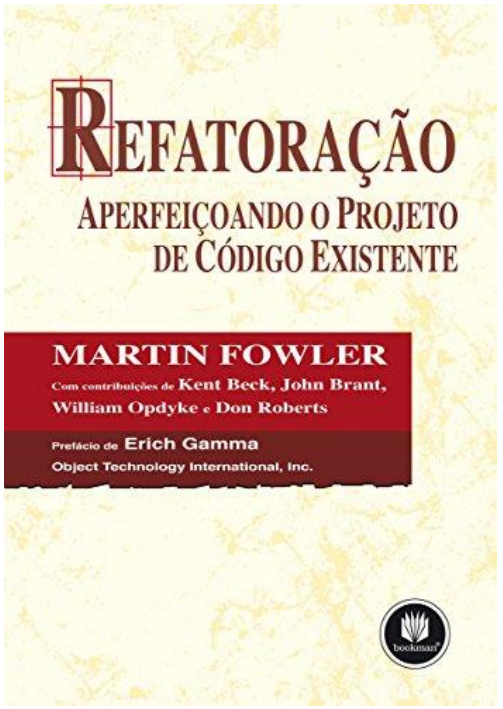
MÓDULO 4

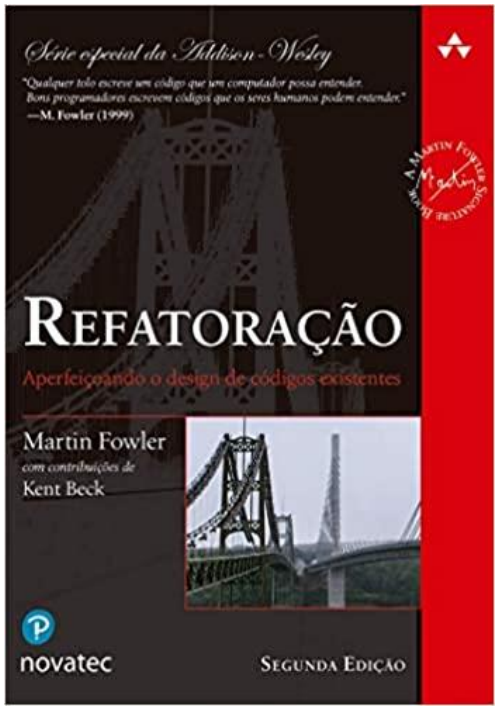
Refatoração

Referência:

- FOWLER, Martin. Refatoração: aperfeiçoando o projeto de código existente, 1ed. Porto Alegre: Bookman, 2004.

- FOWLER, Martin. Refatoração: aperfeiçoando o projeto de código existente, 2ed. Novatec. 2020.





Mover Atributo

Grupo: Movendo Recursos entre Objetos

Resumo: Ocorre quando um atributo pertencente a uma classe é mais utilizado por outra classe do que aquela que o possui.

Move-se tal atributo para a classe que mais o utiliza.

Motivação: Algumas vezes, pode-se encontrar atributos que (i) estão sendo utilizados por outras classes além da classe que o possui ou (ii) atributos que deviam pertencer a uma classe, mas, por uma falha de projeto, estão definidos em outra.

Mover Atributo

Mecânica:

1) Caso o atributo em questão seja publico, utiliza-se a refatoração Encapsular Atributo.

2) Define-se um atributo igual ao analisado na classe destino.

3) Modifica-se os pontos que fazem referência ao atributo que se quer mover e faz-se com que referenciem o atributo recém criado na classe destino.

4) Apaga-se o atributo na classe de origem.

5) Busca-se no código os pontos que referenciem o atributo apagado e faz-se com que referenciem o atributo criado na classe destino.

6) Execute testes.

Mover Atributo

```java
public class Aluno {
  private String nome;
  public String curso;

  public Aluno(String nome) {
    this.nome = nome;
  }

  public void setNome(String nome) {
    this.nome = nome;
  }

  public String getNome() {
    return this.nome;
  }
}
```

```java
public class Curso {
  private String descricao;

  public Curso(String descricao) {
    this.descricao = descricao;
  }

  public String getDescricao () {
    return descricao;
  }

  public void setDescricao (String descricao) {
    this.descricao = descricao;
  }
}
```

Mover Atributo

```java
public class Aluno {
  private String nome;

  public Aluno(String nome) {
    this.nome = nome;
  }

  public void setNome(String nome) {
  this.nome = nome;
  }

  public String getNome() {
    return this.nome;
  }
}
```

```java
public class Curso {
  private String nome;
  private String descricao;

  public Curso(String nome, String descricao) {
    this.nome = nome;
    this.descricao = descricao;
  }

  public String getNome () {
    return nome;
  }

  public void setNome (String nome) {
    this.nome = nome;
  }

  public String getDescricao () {
    return descricao;
  }

  public void setDescricao (String descricao) {
    this.descricao = descricao;
  }
}
```

Acrescentar Parâmetro

Grupo: Tornando as Chamadas de Métodos Mais Simples
Resumo: Alguns métodos precisam de mais informações para executar determinada tarefa. Acrescenta-se um parâmetro a ele e alteram-se as chamadas a ele para que possam fornecer este dado.

Motivação: Usada quando um método recém modificado precisa de mais informações para realizar suas operações.
Como as informações antes disponíveis são poucas, acrescenta-se um parâmetro ao método.

Acrescentar Parâmetro

Mecânica:

1) Certifique-se de que o método não tem sua assinatura implementada em uma super ou subclasse. Caso tenha, cada passo da mecânica deve ser repetido em todas as implementações existentes.

2) Acrescenta-se o novo parâmetro à lista de parâmetros do método.

3) Modifique os pontos que chamam o método, para que passem a referenciar o método novo.

4) Execute testes.

Acrescentar Parâmetro

```
public class Relatorio {
  public void imprimirCabecalhoProduto() {
    System.out.println("--------------------");
    System.out.println("Preço dos Produtos:");
    System.out.println("--------------------");
  }

  public void imprimirCabecalhoServico() {
    System.out.println("--------------------");
    System.out.println("Preço dos Serviços:");
    System.out.println("--------------------");
  }
}

public class Main {
  public static void main (String[] args) {
    Relatorio relatorio = new Relatorio();
    relatorio.imprimirCabecalhoProduto();
    relatorio.imprimirCabecalhoServico();
  }
}
```

Acrescentar Parâmetro

```
public class Relatorio {
  public void imprimirCabecalho(String item) {
    System.out.println("--------------------");
    System.out.format("Preço dos %s:\n", item);
    System.out.println("--------------------");
  }
}

public class Main {
  public static void main (String[] args) {
    Relatorio relatorio = new Relatorio();
    relatorio.imprimirCabecalho("Produtos");
    relatorio.imprimirCabecalho("Serviços");
  }
}
```

Introduzir Objeto Parâmetro

Grupo: Tornando as Chamadas de Métodos Mais Simples

Resumo: Algumas classes possuem vários métodos que recebem vários parâmetros em comum.

Cria-se um objeto que possua os atributos mais usados pelos métodos e modifique-os para receberem um objeto, no lugar dos vários dados que recebiam como parâmetros.

Motivação: Introduzindo objetos no lugar de vários dados como parâmetro, pode-se reduzir e muito o código duplicado, além de juntar os dados em um só lugar.

Pode-se reduzir significativamente o número de parâmetros passados ao método.

Introduzir Objeto Parâmetro

Mecânica:

1) Cria-se a classe que terá os atributos que substituirão os parâmetros passados aos métodos.

2) Declare nela os atributos que deseja substituir pelos parâmetros.

3) Modifique os métodos para receberem como parâmetro um objeto da classe criada, apagando os antigos parâmetros.

4) Execute os testes.

Introduzir Objeto Parâmetro

```
public class Relatorio {
  public void imprimir(String nome, double quantidade, double preco) {
    System.out.println("-----------------------------------------------");
    System.out.format("Nome: %s Quantidade (kg): %.2f Preço: R$ %.2f\n", nome,
quantidade, preco);
    System.out.println("-----------------------------------------------");
  }
}
```

```
public class Main {
  public static void main (String[] args) {
    Relatorio relatorio = new Relatorio();
    relatorio.imprimir("Arroz", 5, 18.75);
  }
}
```

Introduzir Objeto Parâmetro

```
public class Produto {
  private String nome;
  private double quantidade;
  private double preco;
  public Produto(String nome, double quantidade, double preco) {
    this.nome = nome;
    this.quantidade = quantidade;
    this.preco = preco;
  }
  public String getNome () {
    return nome;
  }
  public void setNome (String nome) {
    this.nome = nome;
  }
  public double getQuantidade () {
    return quantidade;
  }
  public void setQuantidade (double quantidade) {
    this.quantidade = quantidade;
  }
  public double getPreco () {
    return preco;
  }
  public void setPreco (double preco) {
    this.preco = preco;
  }
}
```

Introduzir Objeto Parâmetro

```java
public class Relatorio {
  public void imprimir(Produto produto) {
    System.out.println("-----------------------------------------");
    System.out.format("Nome: %s Quantidade (kg): %.2f Preço: R$ %.2f\n",
produto.getNome(), produto.getQuantidade(), produto.getPreco());
    System.out.println("-----------------------------------------");
  }
}
```

```java
public class Main {
  public static void main (String[] args) {
    Relatorio relatorio = new Relatorio();
    relatorio.imprimir(new Produto ("Arroz", 5, 18.75));
  }
}
```

Extrair Subclasse

Grupo: Lidando com Generalização

Resumo: Algumas classes possuem um conjunto de recursos que somente são utilizados em determinadas situações.

Cria-se uma classe filha (subclasse) e move-se este conjunto de recursos para ela.

Motivação: Usa-se esta refatoração para dividir os recursos de uma classe que carrega consigo recursos que poderiam pertencer a uma hierarquia de classes.

Usada quando é clara a necessidade de mover conhecimento de uma classe para outra, neste caso, uma subclasse da classe com muitos recursos.

Extrair Subclasse

Mecânica:

1) Crie uma classe e modifique-a para que se torne uma classe filha da classe que se deseja extrair recursos.

2) Mova os recursos da classe pai para a subclasse que acabara de ser criada.

3) Ajuste as chamadas antes feitas à classe pai, para, a partir desse momento, a subclasse.

4) Certifique-se que tudo está correto testando a aplicação.

Extrair Subclasse

```
public class Anuncio {
  private String titulo;
  private String texto;
  private double preco;

  public String getTitulo () {
    return titulo;
  }
  public void setTitulo (String titulo) {
    this.titulo = titulo;
  }
  public String getTexto () {
    return texto;
  }
  public void setTexto (String texto) {
    this.texto = texto;
  }
  public double getPreco () {
    return preco;
  }
  public void setPreco (double preco) {
    this.preco = preco;
  }
}
```

Extrair Subclasse

```
public class Anuncio {
  private String titulo;
  private String texto;

  public String getTitulo () {
    return titulo;
  }
  public void setTitulo (String titulo) {
    this.titulo = titulo;
  }
  public String getTexto () {
    return texto;
  }
  public void setTexto (String texto) {
    this.texto = texto;
  }
}
```

```
public class AnuncioPago extends Anuncio{
  private double preco;

  public double getPreco () {
    return preco;
  }
  public void setPreco (double preco) {
    this.preco = preco;
  }
}
```

Subir Método/Atributo na Hierarquia

Grupo: Lidando com Generalização

Resumo: Alguns métodos nas subclasses de uma hierarquia apresentam características parecidas uns com os outros.

Neste contexto movem-se os métodos da subclasse para a superclasse.

Motivação: Usada quando há métodos em uma hierarquia de classes, onde eles apresentam características que permitem concluir que há duplicação de código. Alguns métodos que possuem códigos parcialmente idênticos podem vir a se tornar um problema no futuro, considerando que mudanças feitas em um também devem ser feitas no outro.

> **OBS**: quando a refatoração move um método de subclasses para uma superclasse, ele é chamado de **Pull Up Method.**

Subir Método/Atributo na Hierarquia

Mecânica:

1) Analise os métodos para se certificar que eles têm comportamentos iguais ou são totalmente iguais. Se tiverem código parecido, mas com o mesmo objetivo, transforme-os para que fiquem iguais usando a refatoração Substituir Algoritmo.

2) Verifique os nomes dos métodos que se deseja subir na hierarquia. Caso sejam diferentes, modifique para que se tornem iguais.

3) Na superclasse, declare um método com o mesmo nome de um dos métodos que se deseja subir na hierarquia. Copie o código do método que está subindo na hierarquia para o método recém criado na superclasse.

4) Apaga-se o método da subclasse.

5) Repita o processo com todos os métodos que devem subir na hierarquia, removendo assim os métodos duplicados.

6) Realize testes.

Subir Método/Atributo na Hierarquia

```
public class Animal {
  public void comida (String alimento) {
    System.out.println(alimento);
 }
}

public class Cachorro extends Animal {
   public void latir () {
      System.out.println("latido");
   }
   public void pelagem () {
   System.out.println("pelagem de cachorro");
   }
}

public class Galinha extends Animal {
   public void cacarejar () {
      System.out.println("cacarejo");
   }
   public void plumagem () {
   System.out.println("plumagem de galinha");
   }
}
```

Subir Método/Atributo na Hierarquia

```
public class Animal {
  public void comida (String alimento) {
    System.out.println(alimento);
  }
  public void som (String voz) {
    System.out.println(voz);
  }
}

public class Cachorro extends Animal {
   public void pelagem () {
     System.out.println("pelagem de cachorro");
   }
}

public class Galinha extends Animal {
   public void plumagem () {
     System.out.println("plumagem de galinha");
   }
}
```

Descer Método/Atributo na Hierarquia

Grupo: Lidando com Generalização

Resumo: o método ou atributo de uma superclasse é usado por apenas uma (ou algumas) subclasse(s).

Motivação: melhorar a coerência da classe. O método/atributo está localizado onde você espera encontrá-lo. o método ou atributo deveria ser universal para todas as subclasses, mas é usado em apenas uma subclasse. Caso um método ou atributo seja necessário em mais de uma subclasse, mas não para todas, pode ser útil criar uma subclasse intermediária e mover o método para ela. Isso permite evitar a duplicação de código que resultaria do envio de um método para todas as subclasses.

OBS: quando a refatoração move um método/atributo de superclasse para uma subclasse, ele é chamado de **Push Down Method/Field.**

Descer Método/Atributo na Hierarquia

Mecânica:

1)Declare o método em uma subclasse e copie seu código da superclasse.

2)Remova o método da superclasse.

3)Encontre todos os locais onde o método é usado e verifique se ele é chamado a partir da subclasse.

4) Realize testes.

Descer Método/Atributo na Hierarquia

```java
package principal;
public class Funcionario {
  private String nome;
  private double comissao;

  public String getNome () {
    return nome;
  }

  public void setNome (String nome) {
    this.nome = nome;
  }

  public double getComissao () {
    return comissao;
  }

  public void setComissao (double comissao) {
    this.comissao = comissao;
  }
}
```

```java
package principal;
public class Engenheiro extends Funcionario{
  private String crea;

  public String getCrea () {
    return crea;
  }

  public void setCrea (String crea) {
    this.crea = crea;
  }
}
```

```java
package principal;
import java.util.ArrayList;
public class Vendedor extends Funcionario {
  private ArrayList<String> idiomas;

  public ArrayList<String> getIdiomas () {
    return idiomas;
  }

  public void setIdiomas (ArrayList<String> idiomas) {
    this.idiomas = idiomas;
  }
}
```

Descer Método/Atributo na Hierarquia

```
package principal;
public class Funcionario {
  private String nome;

  public String getNome () {
    return nome;
  }

  public void setNome (String nome) {
    this.nome = nome;
  }
}
```

```
package principal;
public class Engenheiro extends Funcionario {
  private String crea;

  public String getCrea () {
    return crea;
  }

  public void setCrea (String crea) {
    this.crea = crea;
  }
}
```

Descer Método/Atributo na Hierarquia

```
package principal;
import java.util.ArrayList;
public class Vendedor extends Funcionario {
  private ArrayList<String> idiomas;
  private double comissao;

  public ArrayList<String> getIdiomas () {
    return idiomas;
  }

  public void setIdiomas (ArrayList<String> idiomas) {
    this.idiomas = idiomas;
  }

  public double getComissao () {
    return comissao;
  }

  public void setComissao (double comissao) {
    this.comissao = comissao;
  }
}
```

Substituir Método por Objeto Método

Grupo: Compondo Métodos

Resumo: Métodos longos demais precisam ser divididos com a refatoração Extrair Método, mas se ele tiver muitas variáveis locais o grau de dificuldade aumenta. Neste contexto, cria-se um objeto cujos atributos equivalem a essas variáveis.

Motivação: Prepara os métodos longos para serem divididos com a refatoração Extrair Método ao diminuir a complexidade deles.

Substituir Método por Objeto Método

Mecânica:

1) Cria-se uma classe cujo nome deve ser o mesmo do método longo.

2) Acrescentam-se atributos a classe criada equivalentes às variáveis locais do método longo.

3) Substitua as referencias às variáveis locais no método longo pelos atributos da classe criada.

4) Realize testes.

Substituir Método por Objeto Método

```
public class Funcionario {
  private String nome;
  private String cpf;

  public String getNome () {
    return nome;
  }
  public void setNome (String nome) {
    this.nome = nome;
  }
  public String getCpf () {
    return cpf;
  }
  public void setCpf (String cpf) {
    this.cpf = cpf;
  }
  public double salario (int horas, double valorHora,
                         int horasExtras, double valorHoraExtra) {
    double valor = horas * valorHora + horasExtras * valorHoraExtra;
    double adicional = valor * 0.1;
    return valor + adicional;
  }
}
```

Substituir Método por Objeto Método

```
public class Salario {
  private int horas, horasExtras;
  private double valorHora, valorHoraExtra;

  public Salario (int horas, int horasExtras,
                  double valorHora, double valorHoraExtra) {
    this.horas = horas;
    this.horasExtras = horasExtras;
    this.valorHora = valorHora;
    this.valorHoraExtra = valorHoraExtra;
  }
  public double total () {
    return this.valor() + this.adicional();
  }
  public double valor () {
   return this.horas * this.valorHora + this.horasExtras * this.valorHoraExtra;
  }
  public double adicional () {
    return this.valor() * 0.1;
  }
}
```

Substituir Método por Objeto Método

```
public class Funcionario {
  private String nome;
  private String cpf;
  private Salario salario;

  public void setNome (String nome) {
    this.nome = nome;
  }
  public String getNome () {
    return nome;
  }
  public String getCpf () {
    return cpf;
  }
  public void setCpf (String cpf) {
    this.cpf = cpf;
  }
  public void setSalario (Salario salario) {
    this.salario = salario;
  }
  public double getSalario () {
    return salario.total();
  }
}
```

Criar um Método Padrão

Grupo: Lidando com a Generalização

Resumo: Alguns métodos em uma hierarquia de classe parecem ser iguais, salvo algumas pequenas diferenças. Cria-se um método que contem os passos comuns aos métodos e sobe-o na hierarquia.

Motivação: Eliminar muito código duplicado de métodos, pois os trechos de código iguais são movidos para um método que servirá como um *template*.

Criar um Método Padrão

Mecânica:

1) Busque por métodos nas subclasses que contenham trechos de código com a mesma funcionalidade.

2) Crie um novo método na superclasse que receberá o trecho de código comum aos métodos em questão.

3) Adapte o software para as novas mudanças.

4) Realize testes.

Criar um Método Padrão

```
public class Salario {
  protected int horas, horasExtras, numeroFilhos;
  protected double valorHora, valorHoraExtra;

  public Salario (int horas, int horasExtras, int numeroFilhos,
                  double valorHora, double valorHoraExtra) {
    this.horas = horas;
    this.horasExtras = horasExtras;
    this.numeroFilhos = numeroFilhos;
    this.valorHora = valorHora;
    this.valorHoraExtra = valorHoraExtra;
  }
}
```

Criar um Método Padrão

```
public class SalarioFuncionario extends Salario {

  public SalarioFuncionario(int horas, int horasExtras, int numeroFilhos,
                            double valorHora, double valorHoraExtra) {
    super(horas, horasExtras, numeroFilhos, valorHora, valorHoraExtra);
  }

  public double total () {
    double valorHora = this.horas * this.valorHora;
    double valorHoraExtra = this.horasExtras * this.valorHoraExtra;
    double valor = valorHora + valorHoraExtra;
    double salarioFamilia = this.numeroFilhos * 1500;
    double adicional = valor * 0.1;
    double soma = valor + salarioFamilia + adicional;
    return soma;
  }
}
```

Criar um Método Padrão

```
public class SalarioGerente extends Salario {

  public SalarioGerente(int horas, int horasExtras, int numeroFilhos,
                        double valorHora, double valorHoraExtra) {
    super(horas, horasExtras, numeroFilhos, valorHora, valorHoraExtra);
  }

  public double total () {
    double valorHora = this.horas * this.valorHora;
    double valorHoraExtra = this.horasExtras * this.valorHoraExtra;
    double valor = valorHora + valorHoraExtra;
    double salarioFamilia = this.numeroFilhos * 1500;
    double adicional = valor * 0.15;
    double soma = valor + salarioFamilia + adicional;
    return soma;
  }
}
```

Criar um Método Padrão

```
public class Salario {
  protected int horas, horasExtras, numeroFilhos;
  protected double valorHora, valorHoraExtra;

  public Salario (int horas, int horasExtras, int numeroFilhos,
                  double valorHora, double valorHoraExtra) {
    this.horas = horas;
    this.horasExtras = horasExtras;
    this.numeroFilhos = numeroFilhos;
    this.valorHora = valorHora;
    this.valorHoraExtra = valorHoraExtra;
  }

  public double subTotal () {
    double valorHora = this.horas * this.valorHora;
    double valorHoraExtra = this.horasExtras * this.valorHoraExtra;
    double valor = valorHora + valorHoraExtra;
    double salarioFamilia = this.numeroFilhos * 1500;
    return valor + salarioFamilia;
  }
}
```

Criar um Método Padrão

```
public class SalarioFuncionario extends Salario {

  public SalarioFuncionario(int horas, int horasExtras, int numeroFilhos,
                            double valorHora, double valorHoraExtra) {
    super(horas, horasExtras, numeroFilhos, valorHora, valorHoraExtra);
  }

  public double total () {
    double adicional = this.subTotal() * 0.1;
    double soma = this.subTotal() + adicional;
    return soma;
  }
}
```

Criar um Método Padrão

```
public class SalarioGerente extends Salario {

  public SalarioGerente(int horas, int horasExtras, int numeroFilhos,
                        double valorHora, double valorHoraExtra) {
    super(horas, horasExtras, numeroFilhos, valorHora, valorHoraExtra);
  }

  public double total () {
    double adicional = this.subTotal() * 0.15;
    double soma = this.subTotal() + adicional;
    return soma;
 }
}
```

Encapsular Coleção

Grupo: Organizando Dados

Resumo: Métodos que retornam uma coleção de objetos podem ter seus dados expostos. Criam-se métodos de gravação, remoção e leitura que serão responsáveis por manipular a coleção de objetos, evitando operações ilegais por parte dos que a manipulam.

Motivação: Esta refatoração deve ser aplicada para evitar o uso indevido de dados contidos em coleções, como por exemplo, objetos.

A falta de controle sobre as coleções de objetos permite que os dados presentes nelas fiquem expostos sujeitos a alterações por quem não tem esse privilégio.

Encapsular Coleção

Mecânica:

1) Busca-se por coleções de dados em que há risco de modificações descontroladas.

2) Substitua os pontos no código onde dados são adicionados ou removidos da coleção por meio de seus métodos (coleções como ArrayLists e Lists possuem o método .*add() e .remove()* que permite a adição e remoção de dados, respectivamente) por métodos que façam a adição e remoção de dados.

3) Altere os métodos que fazem adição ou remoção de dados da coleção para que não permitam a modificação da coleção.

4) Execute testes para se certificar que as modificações não alteraram as funcionalidades do sistema.

Encapsular Coleção

```
import java.util.List;
import java.util.ArrayList;

public class Empresa {
  private List<String> funcionarios = new ArrayList<String>();

  public Empresa() {}

  public List<String> getFuncionarios() {
    return this.funcionarios;
  }

  public void inserirFuncionario(String funcionario){
    funcionarios.add(funcionario);
  }

  public void removerFuncionario(String funcionario){
    funcionarios.remove(funcionario);
  }
}
```

Encapsular Coleção

```
import java.util.List;
public class Main {
  public static void main (String[] args) {
    Empresa empresa = new Empresa();

    empresa.inserirFuncionario("Ana");
    empresa.inserirFuncionario("Bruna");
    empresa.getFuncionarios().add("Carlos");

    System.out.println("\nLista de funcionários:");
    List<String> funcionarios = empresa.getFuncionarios();
    for (String funcionario : funcionarios)
      System.out.println(funcionario);

    empresa.getFuncionarios().remove("Ana");
    empresa.removerFuncionario("Bruna");

    System.out.println("\nLista de funcionários:");
    funcionarios = empresa.getFuncionarios();
    for (String funcionario : funcionarios)
      System.out.println(funcionario);
  }
}
```

Encapsular Coleção

```
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class Empresa {
  private List<String> funcionarios = new ArrayList<String>();

  public Empresa() {}

  public List<String> getFuncionarios() {
    return Collections.unmodifiableList(funcionarios);
  }

  public void inserirFuncionario(String funcionario){
    funcionarios.add(funcionario);
  }

  public void removerFuncionario(String funcionario){
    funcionarios.remove(funcionario);
  }
}
```

Encapsular Coleção

```
import java.util.List;
public class Main {
  public static void main (String[] args) {
    Empresa empresa = new Empresa();

    empresa.inserirFuncionario("Ana");
    empresa.inserirFuncionario("Bruna");
    empresa.inserirFuncionario("Carlos");

    System.out.println("\nLista de funcionários:");
    List<String> funcionarios = empresa.getFuncionarios();
    for (String funcionario : funcionarios)
      System.out.println(funcionario);

    empresa.removerFuncionario("Bruna");

    System.out.println("\nLista de funcionários:");
    funcionarios = empresa.getFuncionarios();
    for (String funcionario : funcionarios)
      System.out.println(funcionario);
  }
}
```

Substituir Comando Condicional por Polimorfismo

Grupo: Simplificando Expressões Condicionais

Resumo: Substitui comandos condicionais que selecionam ações a serem executadas por métodos polimórficos.

Motivação: Usada para remover expressões condicionais para um determinado objetivo que se repetem em vários trechos de código.

Simplifica-se o projeto de código e removem-se duplicações.

Substituir Comando Condicional por Polimorfismo

Mecânica:

1) Verifique o método que contenha uma sentença *switch* ou *else ifs*.
2) Aplica-se a refatoração Extrair Método para que a expressão condicional fique sozinha no método, caso necessário.
3) Usa-se Mover Método para levar o método que contem a expressão condicional para a superclasse.
4) Cria-se um método em cada uma das subclasses e cola-se nele o trecho de código da lógica condicional que se refere à subclasse em questão. Caso a subclasse já possua um método que possa ser usado para substituir o trecho da lógica condicional, desconsidera-se este passo.
5) Deleta-se o código da lógica condicional que acabou de copiar .
6) Executa-se todos os passos até remover toda lógica condicional.
7) Execute testes.

Substituir Comando Condicional por Polimorfismo

```
public class Produto {
  private String tipo;
  private double valor;

  Produto(String tipo, double valor){
    this.tipo = tipo;
    this.valor = valor;
  }

  public double getDesconto() {
    switch (this.tipo) {
      case "roupa":
        return 0.05;
      case "calçado":
        return 0.04;
      case "casa":
        return 0.03;
      default:
        return 0;
    }
  }
}
```

```
import java.util.ArrayList;

public class Main {
  public static void main (String[] args) {
    ArrayList<Produto> vendas = new ArrayList<Produto>();

    vendas.add(new Produto("casa", 99));
    vendas.add(new Produto("calçado", 86));
    vendas.add(new Produto("roupa", 57));

    for (Produto produto : vendas) {
      System.out.println(produto.getDesconto());
    }
  }
}
```

Substituir Comando Condicional por Polimorfismo

```
public abstract class Produto {
  private double valor;

  Produto(double valor){
    this.valor = valor;
  }
  public abstract double getDesconto();
}
```

```
public class Roupa extends Produto {
  public Roupa(double valor) {
    super(valor);
  }
  public double getDesconto () {
    return 0.05;
  }
}
```

```
public class Casa extends Produto {
  public Casa(double valor) {
    super(valor);
  }
  public double getDesconto () {
    return 0.03;
  }
}
```

```
public class Calcado extends Produto {
  public Calcado(double valor) {
    super(valor);
  }
  public double getDesconto () {
    return 0.04;
  }
}
```

```
import java.util.ArrayList;

public class Main {
  public static void main (String[] args) {
    ArrayList<Produto> vendas = new
ArrayList<Produto>();

    vendas.add(new Casa(99));
    vendas.add(new Calcado(86));
    vendas.add(new Roupa(57));

    for (Produto produto : vendas) {
 System.out.println(produto.getDesconto());
    }
  }
}
```

Extrair Classe

Grupo: Movendo Recursos entre Objetos

Resumo: Classes com dupla responsabilidade devem ser divididas em duas ou mais dependendo da natureza das responsabilidades.

Motivação: Alguns desenvolvedores porem acabam implementando funcionalidades em uma classe que não condizem com as suas responsabilidades, o que aumenta a complexidade da classe, reduz a coesão e diminui a manutenibilidade.

Extrair Classe

Mecânica:

1) Analisando uma classe com excesso de responsabilidades, vê-se qual é a melhor divisão possível, pensando em quais classes precisam ser criadas.
2) Cria-se a classe ou as classes necessárias.
3) Movem-se as responsabilidades de uma classe para outra, usando a refatoração Mover Método.
4) Estabeleça as associações corretas com a classe criada.
5) Execute testes.

Extrair Classe

```
public class Funcionario {
  private String nome;
  private double salario;
  private String marca;
  private String modelo;

  Funcionario(String nome, double salario, String marca, String modelo){
    this.nome = nome;
    this.salario = salario;
    this.marca = marca;
    this.modelo = modelo;
  }
}
```

Extrair Classe

```
public class Funcionario {
  private String nome;
  private double salario;
  private Veiculo veiculo;

  Funcionario(String nome, double salario, Veiculo veiculo){
    this.nome = nome;
    this.salario = salario;
    this.veiculo = veiculo;
  }
}
```

```
public class Veiculo {
  private String marca;
  private String modelo;

  Veiculo (String marca, String modelo){
    this.marca = marca;
    this.modelo = modelo;
  }
}
```

Extrair Superclasse

Grupo: Lidando com Generalização

Resumo: Algumas classes possuem características em comum, sejam atributos ou métodos. Cria-se uma superclasse que receberá todas características em comum das classes.

Motivação: Reduz o código duplicado ao centralizar informações que poderão ser consultadas em um único lugar ao invés de vários, o que aumenta a manutenibilidade do código fonte.

Extrair Superclasse

Mecânica:

1) Cria-se uma classe e modificam-se as classes que contem características em comum para se tornarem subclasses da classe criada.
2) Aplica-se nas subclasses a refatoração Subir Método na Hierarquia.
3) Analisando os métodos que restaram nas subclasses, caso haja código em comum entre eles, aplica-se a refatoração Extrair Método e a refatoração Subir Método na Hierarquia novamente. Ainda há a o recurso de usar a refatoração Criar Método Padrão para mover o método padrão para a superclasse.
4) Fazem-se os ajustes necessários nas chamadas às subclasses.
5) Executam-se os testes necessários.

Extrair Superclasse

```java
public class PessoaFisica {
  private String nome;
  private String cpf;

  PessoaFisica (String nome, String cpf){
    this.nome = nome;
    this.cpf = cpf;
  }
}
```

```java
public abstract class PessoaJuridica {
  private String nome;
  private String cnpj;

  PessoaJuridica (String nome, String cnpj){
    this.nome = nome;
    this.cnpj = cnpj;
  }
}
```

Extrair Superclasse

```
public class Pessoa  {
  private String nome;

  Pessoa (String nome){
    this.nome = nome;
  }
}
public class PessoaFisica extends Pessoa {
  private String cpf;

  PessoaFisica (String nome, String cpf){
    super(nome);
    this.cpf = cpf;
  }
}
public class PessoaJuridica extends Pessoa {
  private String cnpj;

  PessoaJuridica (String nome, String cnpj){
    super(nome);
    this.cnpj = cnpj;
  }
}
```

Substituir Herança por Delegação

Grupo: Lidando com Generalização

Resumo: Subclasses que fazem uso de uma pequena parte da superclasse devem ter seus métodos ajustados a fim de que deleguem para a superclasse através de um campo.

Motivação: Usada para simplificar o projeto de código existente, ao substituir uma herança por uma delegação.

A subclasse que faz uso apenas de uma parte do código de sua superclasse pode ter sua complexidade diminuída através da delegação que deixa claro que apenas uma parte da classe pai está sendo utilizada.

Substituir Herança por Delegação

Mecânica:

1) Na subclasse, cria-se um atributo privado do tipo da superclasse.

2) Modificam-se os métodos da subclasse para fazerem uso do atributo privado criado.

3) Modifica-se a subclasse retirando sua classificação de classe filha.

Substituir Herança por Delegação

```
public class Pessoa  {
  private String nome;
  private String lazer;

  Pessoa (String nome, String lazer){
    this.nome = nome;
    this.lazer = lazer;
  }

  public String getLazer() {
    return this.lazer;
  }
}
```

```
public class PessoaJuridica extends Pessoa {
  private String cnpj;

  PessoaJuridica (String nome, String cnpj){
    super(nome, "");
    this.cnpj = cnpj;
  }
}
```

Substituir Herança por Delegação

```
public class Pessoa  {
  private String nome;
  private String lazer;

  Pessoa (String nome, String lazer){
    this.nome = nome;
    this.lazer = lazer;
  }

  public String getLazer() {
    return this.lazer;
  }
}
```

```
public class PessoaJuridica {
  private String cnpj;
  private Pessoa pessoa;

  PessoaJuridica (String cnpj, Pessoa pessoa){
    this.cnpj = cnpj;
    this.pessoa = pessoa;
  }
}
```