AceleraDev Java

Módulo 8

Tópicos da Aula

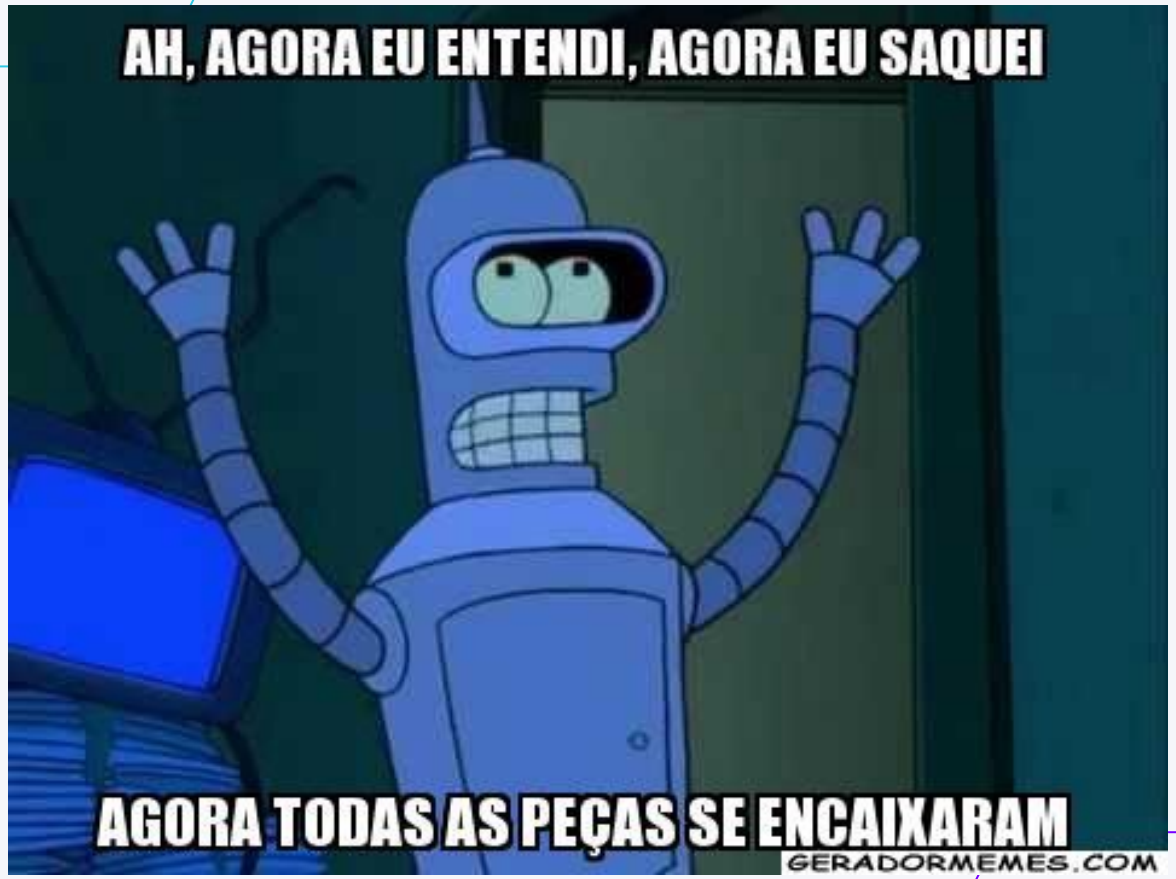
- SOLID;
- CLEAN CODE;
- OBJECT CALISTHENICS;
- TDD.

Boas práticas de desenvolvimento

- O que sao boas práticas de desenvolvimento?
- Por que procuramos desenvolver bons códigos?

O que é SOLID?

- Acrônimo que representa 5 princípios da programação orientada a objetos.
- Criado pelo Uncle Bob (Robert Martin) e nomeado por Michael Feathers
- [S] Single Responsibility Principle (Principio da Responsabilidade Única)
- [O] Open/Closed Principle (Principio do Aberto / Fechado)
- [L] Liskov Substitution Principle (Princípio da Substituição de Liskov).
- [I] Interface Segregation Principle (Princípio da Segregação de Interface)
- [D] Dependency Inversion Principle (Princípio da Inversão de Dependência).


AH, AGORA EU ENTENDI, AGORA EU SAQUEI
AGORA TODAS AS PEÇAS SE ENCAIXARAM
GERADORMEMES.COM

Princípio da Responsabilidade Única (SRP)

A class should have one, and only one, reason to change.

- Esse princípio basicamente trata do nível de coesão de uma classe.
- O que é coesão?

```
public class Funcionario {

	private Integer id;
	private String nome;
	private Double salario;
	private Connection connection;

	public String getNome() {
		return nome;
	}

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

	public Double getSalario() {
		return salario;
	}

	public void setSalario(Double salario) {
		this.salario = salario;
	}

	public Double calculaSalario() {
		return this.salario - (this.salario * 0.225);
	}

	public void salva() throws SQLException{

		this.connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/empresa?useSSL=false", "root", "");
		Statement stmt = this.connection.createStatement();
		String sql = "insert into funcionario (id, nome, salario) values (" + this.id + "," +
				this.nome + "," + this.salario + ")";
		int rs = stmt.executeUpdate(sql);

		if (rs == 1){
			System.out.println("Funcionario inserido com sucesso.");
		}else if (rs == 0){
```

O que há de errado na classe anterior?

Nada? Tudo? O que deveríamos mudar? Ela possui as responsabilidades que deveria?

- O que podemos fazer para melhorar?
  - Divisão de responsabilidades em outras classes

```java
public class ConnectionDAO {

	private Properties connectionProps;
	private Connection conn;
	private String dbms;
	private String dbName;
	private String serverName;
	private String portNumber;

	private static final String JDBC = "jdbc:";

	private static final Logger logger = Logger.getLogger(ConnectionDAO.class);
```

```java
public void setPortNumber(String portNumber) {
	this.portNumber = portNumber;
}

public Connection createConnection() {
	Connection newConnection = null;
try {

	if (getDbms().equals("mysql")) {
			newConnection = DriverManager.getConnection(JDBC + getDbms() + "://" + getServerName() + ":" + getPortNumber()
			+ "/" + getDbName() + "?useSSL=false", getConnectionProps());
	}else if (getDbms().equals("postgreSQL")){
	newConnection = DriverManager.getConnection(JDBC + getDbms() + "://" + getServerName() + ":" + getPortNumber()
	+ "/" + getDbName() + "?useSSL=false", getConnectionProps());
   }else if (getDbms().equals("derby")) {
	newConnection = DriverManager.getConnection(JDBC + getDbms() + ":" + getDbName() + ";create=true", getConnectionProps());
   }
	setConnection(newConnection);
	logger.info("Connected to database");
	} catch (SQLException e) {
		logger.error(e);
	}

   return newConnection;
}
```

```java
public class FuncionarioDAO {

	private static final Logger logger = Logger.getLogger(FuncionarioDAO.class);

	public void salva(Funcionario funcionario) throws SQLException{

		ConnectionDAO connectionDAO = new ConnectionDAO("root", "");
		connectionDAO.setDbms("mysql");
		connectionDAO.setServerName("localhost");
		connectionDAO.setPortNumber("8080");
		connectionDAO.setDbName("mock");

		try (Connection connection = connectionDAO.createConnection();
			 Statement stmt = connection.createStatement();) {

			String sql = "insert into funcionario (id, nome, salario) values (" + funcionario.getId() + "," +
					funcionario.getNome() + "," + funcionario.getSalario() + ")";
			int rs = stmt.executeUpdate(sql);

			if (rs == 1){
				logger.info("Funcionario inserido com sucesso.");
			}
		} catch (SQLException e) {
			logger.error("Nenhum funcionario inserido." + e);
		}
	}
}
```

```java
public interface RegraDeCalculo {

	public double calcula (Funcionario funcionario);

}
```

```java
public enum Cargo {

	DESENVOLVEDOR_SENIOR(new RegraVinteDoisEMeioPorcento()),
	DESENVOLVEDOR_JUNIOR(new RegraOnzePorcento());

	private RegraDeCalculo regra;

	Cargo(RegraDeCalculo regra){
		this.regra = regra;
	}

	public RegraDeCalculo getRegra() {
		return regra;
	}

}
```

```java
public class RegraVinteDoisEMeioPorcento implements RegraDeCalculo{

        @Override
        public double calcula(Funcionario funcionario) {
                return funcionario.getSalario() - (funcionario.getSalario() * 0.225);
        }

}
```

```java
public class RegraOnzePorcento implements RegraDeCalculo{

        @Override
        public double calcula(Funcionario funcionario) {
                return funcionario.getSalario() - (funcionario.getSalario() * 0.11);
        }

}
```

```java
public class Funcionario {

	private Integer id;
	private String nome;
	private double salario;
	private Cargo cargo;

	public Funcionario() {}

	public Funcionario(Integer id, String nome, double salario, Cargo cargo) {
		this.id = id;
		this.nome = nome;
		this.salario = salario;
		this.cargo = cargo;
	}
```

```java
	public void setCargo(Cargo cargo) {
		this.cargo = cargo;
	}

	public double calculaSalario() {
		return cargo.getRegra().calcula(this);
	}
```

Princípio do Aberto/Fechado (OCP)

You should be able to extend a classes behavior, without modifying it.

- Abertas para ampliaçao, mas fechadas para modificaçao.
- Utilizar herança, interface e composição quando for necessário, mas não podemos permitir a abertura dessa classe para fazer pequenas modificações.
- Observe a seguinte classe de um e-commerce fictício:

```java
public class CalculadoraDePrecos {

    public double calcula(Produto produto) {

        Frete frete = new Frete();
        double desconto = 0d;

        int regra = produto.getMeioPagamento();

        switch(regra) {
                case 1:
                        System.out.println("Venda à vista");
                        TabelaDePrecoAVista tabela1 = new TabelaDePrecoAVista();
                        desconto = tabela1.calculaDesconto(produto.getValor());
                        break;
                case 2:
                        System.out.println("Venda à prazo");
                        TabelaDePrecoAPrazo tabela2 = new TabelaDePrecoAPrazo();
                        desconto = tabela2.calculaDesconto(produto.getValor());
                        break;
        }

        double valorFrete = frete.calculaFrete(produto.getEstado());
        return produto.getValor() * (1 - desconto) + valorFrete;
    }
}
```

```java
public class TabelaDePrecoAVista {

	public double calculaDesconto(double valor) {
		if(valor > 100.0) {
			return 0.05;
		}else if(valor > 500.0) {
			return 0.07;
		}else if(valor > 1000.0) {
			return 0.10;
		}else {
			return 0d;
		}
	}

}
```

```java
public class TabelaDePrecoAPrazo {

	public double calculaDesconto(double valor) {
		if(valor > 100.0) {
			return 0.01;
		}else if(valor > 500.0) {
			return 0.02;
		}else if(valor > 1000.0) {
			return 0.05;
		}else {
			return 0d;
		}
	}

}
```

```java
public class Frete {

        public double calculaFrete(String estado) {

                if("SAO PAULO".equals(estado.toUpperCase())) {
                        return 7.5;
            }else if("MINAS GERAIS".equals(estado.toUpperCase())){
                return 12.5;
            }else if("RIO DE JANEIRO".equals(estado.toUpperCase())) {
                return 10.5;
            }else {
                return 10.0;
            }
        }

}
```

Qual o problema da implementação anterior?

- Complexidade?
- Acoplamento?
- Como resolver?

```java
public interface TabelaDePreco {

        public double calculaDesconto(double valor);

}
```

```java
public interface ServicoDeFrete {

        public double calculaFrete(String estado);

}
```

```java
public class TabelaDePrecoAPrazo implements TabelaDePreco{

	@Override
	public double calculaDesconto(double valor) {
		if(valor > 100.0) {
			return 0.01;
		}else if(valor > 500.0) {
			return 0.02;
		}else if(valor > 1000.0) {
			return 0.05;
		}else {
			return 0d;
		}
	}

}
```

```java
public class TabelaDePrecoAVista implements TabelaDePreco{

	@Override
	public double calculaDesconto(double valor) {
		if(valor > 100.0) {
			return 0.05;
		}else if(valor > 500.0) {
			return 0.07;
		}else if(valor > 1000.0) {
			return 0.10;
		}else {
			return 0d;
		}
	}

}
```

```java
public class Frete implements ServicoDeFrete{

        @Override
        public double calculaFrete(String estado) {
                if("SAO PAULO".equals(estado.toUpperCase())) {
                        return 7.5;
            }else if("MINAS GERAIS".equals(estado.toUpperCase())){
                return 12.5;
            }else if("RIO DE JANEIRO".equals(estado.toUpperCase())) {
                return 10.5;
            }else {
                return 10.0;
            }
        }

}
```

Resultado:

```
public class CalculadoraDePrecos {

        private TabelaDePreco tabela;
        private ServicoDeFrete frete;

        public CalculadoraDePrecos(TabelaDePreco tabela, ServicoDeFrete frete) {
                this.tabela = tabela;
                this.frete = frete;
        }

    public double calcula(Produto produto) {
        double desconto = tabela.calculaDesconto(produto.getValor());
        double valorFrete = frete.calculaFrete(produto.getEstado());
        return produto.getValor() * (1 - desconto) + valorFrete;
    }
}
```

Princípio da Substituição de Liskov (LSP)

Derived classes must be substitutable for their base classes.

- "Os subtipos devem ser substituíveis pelos seus tipos de base";
- Herança deve ser utilizada de forma contextualizada e moderada, evitando os casos de classes serem estendidas apenas por possuírem algo em comum
- Descrita pela pesquisadora Barbara Liskov.

```java
public class ContaCorrenteComum {

    protected double saldo;

    public ContaCorrenteComum() {
        this.saldo = 0;
    }

    public void deposita(double valor) {
        this.saldo += valor;
    }

    public void saca(double valor) {
        if(valor <= this.saldo) {
             this.saldo -= valor;
        }else{
             throw new IllegalArgumentException("Saldo insuficiente.");
        }
    }

    public double getSaldo() {
        return saldo;
    }

    public void rende()      {
        this.saldo*= 0.02;
    }
}
```

```java
public class ContaSalario extends ContaCorrenteComum {

        public void rende() {
        throw new Exception("Essa conta não possui rendimento");
  }

}
```

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

public class Banco {

    public static void main(String[] args) {

        List<ContaCorrenteComum> listaDeContas = new ArrayList<>();
        listaDeContas.add(new ContaCorrenteComum());
        listaDeContas.add(new ContaSalario());

        for (ContaCorrenteComum conta : listaDeContas) {
            conta.rende();

            System.out.println("Novo Saldo:");
            System.out.println(conta.getSaldo());
        }
    }

}
```

Qual o problema da implementação anterior?

- É correto subirmos uma exceção para a classe que não possui o método rende()?
- Como resolver?

Composiçao:

```java
public class GerenciadorDeContas {

    private double saldo;

    public void deposita(double valor) {
        this.saldo += valor;
    }

    public void saca(double valor) {
        if(valor <= this.saldo) {
                this.saldo -= valor;
        }else{
                throw new IllegalArgumentException("Saldo insuficiente.");
        }
    }

    public double getSaldo() {
        return saldo;
    }

    public void rende(double taxa){
        this.saldo = this.saldo + (this.saldo*taxa);
    }
}
```

```java
public class ContaCorrenteComum {

    private GerenciadorDeContas gerenciador;

    public ContaCorrenteComum() {
        this.gerenciador = new GerenciadorDeContas();
    }

    public void deposita(double valor) {
        this.gerenciador.deposita(valor);
    }

    public void saca(double valor) {
        this.gerenciador.saca(valor);
    }

    public double getSaldo() {
        return this.gerenciador.getSaldo();
    }

    public void rende() {
        this.gerenciador.rende(0.02);
    }

    @Override
    public String toString() {
        return "Saldo conta corrente-> " + this.getSaldo();
    }
}
```

```java
public class ContaSalario {

    private GerenciadorDeContas gerenciador;

    public ContaSalario() {
        this.gerenciador = new GerenciadorDeContas();
    }

    public void deposita(double valor) {
        this.gerenciador.deposita(valor);
    }

    public void saca(double valor) {
        this.gerenciador.saca(valor);
    }

    public double getSaldo() {
        return this.gerenciador.getSaldo();
    }

    @Override
    public String toString() {
        return "Saldo conta salario-> " + this.getSaldo();
    }
}
```

Princípio da Segregação de Interfaces (ISP)

Make fine grained interfaces that are client specific.

- "Muitas interfaces específicas são melhores do que uma interface geral".;
- Interfaces que possuem muitos comportamentos são difíceis de manter e evoluir e devem ser evitadas.

```java
public abstract class Funcionario {

	private String nome;

	public String getNome() {
		return nome;
	}

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

	public abstract double getSalario();

	public abstract double getComissao();
}
```

```java
public class Vendedor extends Funcionario{

	private double salario;
	private int totalVendas;

	public Vendedor(double salario, int totalVendas) {
		this.salario = salario;
		this.totalVendas = totalVendas;
	}

	@Override
	public double getSalario() {
		return this.salario + this.getComissao();
	}

	@Override
	public double getComissao() {
		return this.totalVendas * 0.2;
	}

}
```

```java
public class Desenvolvedor extends Funcionario{

	private double salario;

	public Desenvolvedor(double salario) {
		this.salario = salario;
	}

	@Override
	public double getSalario() {
		return this.salario;
	}

	@Override
	public double getComissao() {
		return 0d;
	}

}
```

Qual o problema da implementação anterior?

- O comportamento de getComissao() faz sentido para o cargo de Desenvolvedor?
- O cálculo do salário é calculado com base nas horas trabalhadas e contratadas, não tendo relação com o total de vendas em um período.
- Como resolver?

```java
public interface Convencional {
  public double getSalario();
}
```

```java
public interface Comissionavel {

        public double getComissao();

}
```

```java
public abstract class Funcionario implements Convencional{

	private String nome;
	private double salario;

	public String getNome() {
		return nome;
	}

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

	@Override
	public double getSalario() {
		return this.salario;
	}

	public void setSalario(double salario) {
		this.salario = salario;
	}
}
```

```java
public class Vendedor extends Funcionario implements Comissionavel{

	private double salario;
	private int totalVendas;

	public Vendedor(double salario, int totalVendas) {
		this.salario = salario;
		this.totalVendas = totalVendas;
	}

	@Override
	public double getSalario() {
		return this.salario + this.getComissao();
	}

	@Override
	public double getComissao() {
		return this.totalVendas * 0.2;
	}

	@Override
	public String toString() {
		return "Vendedor [salario=" + salario + ", totalVendas=" +
	}
}
```

Tenha cuidado!

- O ISP nos alerta em relação às classes "gordas", que causam acoplamentos bizarros e prejudiciais as regras de negócio.
- É necessário tomar cuidado para não exagerar, verificando se a segregação é realmente necessária.

Princípio da Inversão de Dependências (DIP)

Depend on abstractions, not on concretions.

- "Depender de abstrações e não de classes concretas".;
- Uncle Bob quebra a definicao em dois sub-items:
  - "Módulos de alto nível não devem depender de módulos de baixo nível".
  - "As abstrações não devem depender de detalhes. Os detalhes devem depender das abstrações".

O que podemos concluir dos princípios?

- Qual o objetivo deles?
- Por que devemos utilizá-los?
- Alguns pontos:
  - Don't repeat yourself
  - Code-Smells

Feedback da aula