Desenvolvimento de Sistemas

Princípios de projeto: coesão, acoplamento, ocultamento de informação e integridade em orientação a objetos; SOLID – princípio da responsabilidade única, princípio do aberto/fechado, princípio da substituição de Liskov, princípio da segregação de interface, princípio da inversão de dependência; injeção de dependência

A construção de um sistema de *software* às vezes pode ser como um castelo de cartas, no qual cada parte – no caso, cada funcionalidade – é construída de maneira meticulosa sobre outras, dependendo de uma anterior. Se a etapa anterior falha, ou seja, se a carta é retirada ou não está firme o suficiente, o castelo (ou o sistema) pode ruir completamente.

Planejar uma boa arquitetura de código, escrever classes de qualidade e planejar a interação entre elas ajuda a construir um sistema orientado a objetos de qualidade. Alguns princípios de projeto, ou seja, práticas específicas de codificação que se preocupam não só com a funcionalidade a ser executada, mas também com a maneira como ela é implementada, podem ajudar nesse planejamento.

Antes dos princípios de projeto, tem-se as propriedades de projetos, que são recomendações mais genéricas para a construção de uma boa estrutura de código. Destacam-se como propriedades a coesão, o acoplamento e o ocultamento de informação. São itens que ajudam a planejar e organizar as classes e as

funcionalidades do sistema de maneira que ele seja de fácil compreensão, manutenção e extensão. Os princípios partem das propriedades para construir recomendações mais concretas que o desenvolvedor poderá seguir para atender a essas propriedades.

Este conteúdo, assim, aborda as propriedades e os princípios mais utilizados no desenvolvimento de *software* orientado a objetos. Confira mais detalhes a seguir.

Coesão e acoplamento

É possível que você já tenha se deparado, durante seus estudos sobre orientação a objetos, com as expressões “alta coesão” e “baixo acoplamento”. Inclusive, pode ser que elas tenham ficado vagas em um primeiro momento. Trata-se de conceitos que surgiram no contexto de análise e projeto e que, muitas vezes, são negligenciados ou desconhecidos pelos programadores, o que é um erro, visto que podem impactar significativamente a qualidade do sistema.

É importante separar a prática de programação estruturada da prática orientada a objetos.

O foco da programação estruturada (ou procedural) está na funcionalidade e na separação dessas funções que manipulam dados em procedimentos reusáveis, ou seja, na modularidade do sistema.

Por outro lado, na orientação a objetos, a subdivisão do sistema não está simplesmente em funções separadas, mas, sim, em um mapeamento de objetos de domínio do sistema, mais complexos e mais completos.

As funções ou os procedimentos, na programação estruturada, podem lidar com diferentes aspectos do sistema, processando dados de natureza diversa em um mesmo bloco de código, o que muitas vezes dificulta a manutenção. Já na

programação orientada a objetos, há um “tema” – a classe –, e o recomendado é que as funcionalidades presentes sejam relacionadas a esse “tema” apenas, separando as responsabilidades e facilitando a manutenção e o reúso.

Sendo assim, nesse contexto surgem os termos de coesão e acoplamento entre classes:

Coesão

A coesão é o conceito que define o quanto as operações presentes em um objeto estão relacionadas umas com as outras. Todos os métodos e todos os atributos de uma classe devem estar voltados para a implementação do mesmo serviço, representado pela classe. Toda classe, por sua vez, deve ter uma única responsabilidade no sistema.

Por exemplo, imagine que, em um sistema de vendas, exista uma classe **Cliente**. Todos os métodos dela precisam manipular dados de clientes; caso haja métodos ou atributos que se refiram, por exemplo, a vendas (como um cadastramento de venda com base na classe **Cliente**), pode-se dizer que há **baixa coesão** nessa classe.

Veja a proposta para uma classe **Venda** desse mesmo hipotético sistema:

```
  public class Venda {
    private LocalDate data;
    private String status;
    private double valorTotal;
    private String tipoPagamento;
    private String numeroCartao;
    private String nomeCartao;
    private LocalDate vencimentoBoleto;
    private LocalDate dataPagamento;

    public void registraVenda(){
        //grava venda no banco de dados
    }

    public void aplicaDesconto(double desconto){
        //atualiza o valor da venda com desconto informado
    }

    public void realizaPagamento(){
        //executa rotinas relativas a pagamento
    }

    public void trocaFormaPagamento(){
        //altera forma de pagamento dessa venda
    }

    public void notificaClienteVencimento(){
        //envia email ao cliente informando que a fatura de pagamento vence
u
    }
  }
```

Observe com atenção os atributos presentes nessa classe. Todos eles se referem especificamente à **Venda**? Quanto aos métodos, quais deles estão tratando de outros assuntos, alheios à **Venda** (embora relacionados de alguma maneira)?

Veja bem: **data**, **status** e **valorTotal** são intrinsecamente dados de venda. Os métodos **registraVenda()** e **aplicaDesconto()** também são manipulações diretas de venda – o primeiro persistiria os dados da venda, e o

segundo recalcularia o total da venda.

Por outro lado, por mais que o pagamento de uma venda seja um assunto relacionado, ele não faz parte da venda em si, podendo ser separado. Assim, **tipoPagamento**, **numeroCartao**, **nomeCartao**, **vencimentoBoleto** e **dataPagamento** são atributos que poderiam todos fazer parte de uma nova classe chamada **Pagamento**, deixando a classe **Venda** livre dessa responsabilidade adicional e mal vinda.

Além disso, os métodos **realizaPagamento()**, **trocaFormaPagamento()** e **notificaClienteVencimento()** não são operações com as quais **Venda** tem que se preocupar, e sim problemas para a classe **Pagamento**.

Após essa separação, poderia se ter uma ligação entre as duas classes, ou seja, uma **Venda** tem um **Pagamento**. Veja uma refatoração proposta para a classe citada:

```
public class Venda {
  private LocalDate data;
  private String status;
  private double valorTotal;
  private Pagamento pagamento;

  public void registraVenda(){
      //grava venda no banco de dados
  }

  public void aplicaDesconto(double desconto){
      //atualiza o valor da venda com desconto informado
  }
}
```

```
  public class Pagamento {
    private String tipoPagamento;
    private String numeroCartao;
    private String nomeCartao;
    private LocalDate vencimentoBoleto;
    private LocalDate datapagamento;

    public void realizaPagamento(){
        //executa rotinas relativas a pagamento
    }

    public void trocaFormaPagamento(){
        //altera forma de pagamento dessa venda
    }

    public void notificaClienteVencimento(){
        //envia email ao cliente informando que a fatura de pagamento vence
u
    }
  }
```

É possível dizer agora que as classes **Venda** e **Pagamento** estão mais coesas.

Os exemplos citados omitiram o código dos métodos, pois o importante a notar aqui é a estrutura da classe em si. Também foram omitidos *getters* e *setters* para fins de clareza.

Confira a seguir outro exemplo com duas classes – uma não coesa (à esquerda) e outra com mais coesão (à direita):

```
public class FazTudo {
  public void obtemConexaoBancoDeDados(){
      //realiza conexão
  }

  public Usuario obtemDetalhesUsuario(){
      //retorna dados do usuário
  }

  public void validaDadosUsuario(Usuario u){
      //verifica se os dados do usuário estão corretos
  }

  public void enviaEmail(){
      //manda um email qualquer
  }

  public void validaEmail(Email e) {
      //valida se o email está correto
  }
}
```

```
public class BancoDeDados {
  public void obtemConexaoBancoDeDados(){
      //realiza conexão
  }
}

public class Usuario {
  public Usuario obtemDetalhesUsuario(){
      //retorna dados do usuário
  }
}

public class Email {
  public void enviaEmail(){
      //manda um email qualquer
  }
}

public class Validacao {
  public void validaDadosUsuario(Usuario u){
      //verifica se os dados do usuário estão corretos
  }

  public void validaEmail(Email e) {
      //valida se o email está correto
  }
}
```

Classes **FazTudo**, por incrível que pareça, são comuns em códigos de sistema com manutenção ruim. Note ainda no exemplo que a classe **Validacao** pode ser questionável – talvez o mais adequado fosse criar uma classe para validação de *e-mail* e outra para validação de usuário. Isso dependeria do contexto geral do sistema e das demais classes.

O princípio da coesão pode ser aplicado não só a classes, mas a estruturas mais simples, como métodos. Veja este exemplo:

```
public double senoOuCoseno(double x, String opcao) {
  double resultado;
  if(opcao.equals("seno"))
      resultado = Math.sin(x);
  else
      resultado = Math.cos(x);
  return resultado;
}
```

Basicamente, o método faz duas coisas: calcular o seno ou o cosseno de acordo com o parâmetro informado. O recomendável, nesse caso, seria a separação em dois métodos distintos.

Em suma, se um módulo ou uma classe realiza uma tarefa somente ou tem um propósito claro, então o módulo ou a classe em questão tem **alta coesão**. Por outro lado, se o módulo tenta encapsular mais de um propósito ou não tem um propósito claro, apresenta assim **baixa coesão**.

Acoplamento

O acoplamento é o grau de ligação entre duas classes. Sabe-se que as classes podem depender umas das outras (como no exemplo entre **Venda** e **Pagamento**). A maneira como tal dependência acontece define a força do acoplamento: o alto acoplamento ocorre em classes “engessadas”, que acabam limitadas por essa dependência, enquanto o baixo acoplamento garante mais liberdade para as classes envolvidas.

O acoplamento entre duas classes **A** e **B** aumenta à medida que:

- **A** tem um atributo que referencia a classe **B**
- **A** invoca algum método de um objeto de **B**
- **A** tem um método que referencia a classe **B** (por retorno ou parâmetro)
- **A** é derivada de **B**

O baixo acoplamento é o relacionamento em que uma classe interage com outra com base em interfaces simples e em que uma classe não precisa saber (ou não está dependente) **como** a outra classe é implementada internamente.

Considera-se um “acoplamento aceitável” entre as classes **A** e **B** quando:

- A classe **A** usa apenas métodos públicos da classe **B**
- A interface provida por **B** é estável, não tendo mudanças frequentes de assinatura em seus métodos

Ainda assim, nem sempre um alto acoplamento é algo facilmente evitável ou exatamente indesejável em uma classe. Tudo depende muito da complexidade e das operações das classes. O objetivo ao observar o acoplamento não é eliminá-lo completamente das classes, pois é natural que uma classe necessite de outra.

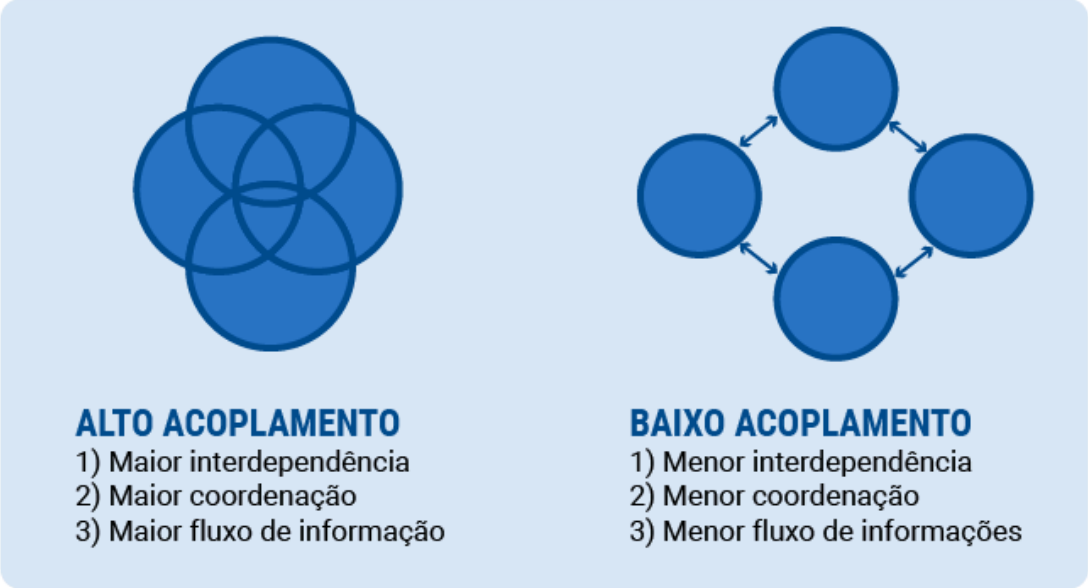


Figura 1 – Representação do alto grau de acoplamento *versus* baixo grau de acoplamento
Fonte: Adaptado de GeeksforGeeks (2020)

Veja no exemplo uma classe para **Agenda**:

```
  public class Agenda {
    private ArrayList<Pessoa> contatos;

    public Agenda() {
        contatos = new ArrayList<Pessoa>();
    }

    public void adicionarContato(Pessoa p){
        contatos.add(p);
    }

    public Pessoa buscacontato(String nome){
        Pessoa resultado = null;
        int i =0;
        while(i < contatos.size() && !contatos.get(i).getNome().equals(nom
e)) {
            i++;
        }

        if(i < contatos.size())
            resultado = contatos.get(i);

        return resultado;
    }
  }
```

Pode-se dizer que **Agenda** está altamente acoplada com a classe **ArrayList** (veja a linha destacada no código). Isso porque a lista de **Pessoa** (que constitui a lista de contatos da agenda) é explicitamente desse tipo. No construtor, há um novo objeto sendo criado com o tipo **ArrayList**. É possível diminuir o acoplamento de **Agenda** usando a interface **List** em vez de **ArrayList**:

```
  private List<Pessoa> contatos;
```

Como este exemplo está lidando apenas com os métodos públicos de **List**, **(get()**, **add())**, que são uma interface estável (Java não deve alterar a assinatura dos métodos existente em um futuro próximo), tem-se um acoplamento aceitável. Ainda há certa dependência entre a classe **Agenda** e a classe **ArrayList**, uma vez que o objeto **contatos** é instanciado com esse tipo no construtor de **Agenda**.

Porém, note que, como não está sendo usado nenhum método explicitamente de **ArrayList**, seria muito simples alterar a estrutura de dados usada caso necessário – por questões de *performance*, por exemplo –, desde que ela implemente **List**.

Veja outro exemplo de acoplamento:

```
class Volume {
  public static void main(String args[]) {
      Caixa c = new Caixa(5,5,5);
      System.out.println(c.volume);
  }
}

class Caixa {
  public int volume;
  Caixa(int length, int width, int height) {
      this.volume = length * width * height;
  }
}
```

A classe **Volume** é dependente do atributo público **volume** de **Caixa**. Imagine que você queira alterar a visibilidade ou mesmo o nome desse atributo. Então, a classe **Volume** e todas as outras que dependem de **Caixa** terão que passar por alterações.

Uma proposta melhor é a seguinte:

```
class Volume {
  public static void main(String args[]) {
      Caixa c = new Caixa(5,5,5);
      System.out.println(c.getVolume());
  }
}

class Caixa {
  private int volume;
  Caixa(int length, int width, int height) {
      this.volume = length * width * height;
  }
  public int getVolume() {
      return volume;
  }
}
```

Em vez de depender diretamente do atributo **volume**, agora a classe **Volume** depende de um método **getVolume()**, e, se houver mudanças no atributo, não será necessário mexer em nada na classe **Volume**.

Algumas das desvantagens do alto acoplamento são:

- Uma mudança em uma classe pode forçar um “efeito dominó” de mudanças em outras classes.
- Uma classe pode se tornar mais difícil de ser reutilizada ou testada porque suas dependências precisam também ser incluídas.
- A compilação pode ser mais lenta quando tais dependências são entre módulos de sistema ou bibliotecas separadas.

É desejável, portanto, que o código **maximize a coesão** e **minimize o acoplamento** em suas classes.

Ocultamento de informação e integridade em orientação a objetos

A integridade conceitual é basicamente a propriedade ou o cuidado com o projeto para que ele não se torne apenas um amontoado de funcionalidades sem coesão e coerência entre elas.

O usuário de um sistema deve se familiarizar com as telas, por exemplo. Assim, se forem apresentados botões de ação (como **OK** e **Cancelar**) em determinado local em uma tela, é preciso obedecer ao mesmo padrão nas demais; se forem apresentados os dados recuperados de uma pesquisa em uma tabela com funções de ordenação em uma tela, em outra tela eles não devem ser apresentados em um arquivo ou em modo de texto livre.

Em código, a falta de integridade também pode ser observada (e corrigida) quando:

- É verificado que algumas variáveis usam padrão *camel case* (por exemplo: **minhaVariavel**) e outras usam *snake case* (por exemplo: **minha_variavel**)
- Parte do sistema usa uma biblioteca ou uma ferramenta para construir telas do sistema, por exemplo, e outra parte utiliza outra biblioteca
- Em uma parte do sistema, resolve-se um problema usando determinada estrutura de dados e em outra parte, para um problema semelhante, aplica-se outra estrutura de dados
- Determinada parte do sistema armazena dados em banco de dados e outra, em arquivos

Os itens citados não são regras, mas exemplos de falta de integridade em um projeto orientado a objetos. É importante, acima de tudo, que haja padronizações definidas pelo time de desenvolvimento sobre temas como escrita de código, espaçamento, bibliotecas a serem usadas, entre outros.

Outro ponto que pode ajudar na qualidade do código do sistema é o ocultamento de informação. Sabe-se que Java, assim como outras linguagens, traz os modificadores de acesso *private* e *protected*, que encapsulam implementações e informações de uma classe. Obviamente, uma classe pode se tornar inútil se tudo nela for privado. Assim, foram expostos apenas alguns métodos e com base em suas assinaturas.

O conjunto de métodos públicos de uma classe é a interface desta, e é essa interface que deve servir para a interação com as demais classes. Entre as vantagens do ocultamento de informação, estão:

- Duas classes que ocultam suas informações podem ser mais facilmente desenvolvidas em paralelo, por dois desenvolvedores, pois uma não interfere na implementação da outra.
- Classes com sua implementação encapsulada geralmente podem ser trocadas por outras equivalentes que sejam mais eficientes (*vide* o exemplo de **Agenda** discutido anteriormente).
- O entendimento da classe fica mais fácil, pois o desenvolvedor não precisará saber todos os detalhes envolvidos nas operações dela, mas, sim, efetivamente o que a classe faz, o que espera de entradas e o que oferece de saídas em seus métodos.

Os métodos *getters* e *setters* nas classes Java são exemplos de ocultamento de informação. Veja um exemplo em que **não** há esse ocultamento:

```
 public class Notas {

public Hashtable notas;


public Notas() {

notas = new Hashtable();

}
 }

 public class Escola {

public static void main(String[] args) {

Notas n = new Notas();

n.notas.put("João Silva", 10.0);

n.notas.put("Pedro Souza", 7.5);

}
 }
```

Observe que a classe **Notas** mantém seu atributo **notas** público. No código principal, manipulou-se diretamente a estrutura de dados de **Hashtable**, o que não é uma boa prática (pode bagunçar todas as notas). Ao contrário, o recomendável é o seguinte:

```
public class Notas {
  private Hashtable<String, Double> notas;

  public Notas() {
      notas = new Hashtable<String, Double>();
  }

  public void cadastra(String aluno, double nota) {
      notas.put(aluno, nota);
  }
}

public class Escola {
  public static void main(String[] args) {
      Notas n = new Notas();
      n.cadastra("João Silva", 10.0);
      n.cadastra("Pedro Souza", 7.5);
  }
}
```

Agora foi encapsulado o atributo **nota**, que é manipulável apenas a partir do método público **cadastra()**, que pode inclusive realizar outras operações e validações úteis nessa função. Caso você quisesse trocar a estrutura **Hashtable** por outra, poderia fazê-lo sem preocupação em **Notas**, pois isso está oculto para as demais classes.

Princípios SOLID

Criado por Robert Martin e Michael Feathers, **SOLID** é um acrônimo referente a cinco práticas recomendáveis a projetos orientados a objetos, por meio das quais serão atingidos a alta coesão e o baixo acoplamento.

- **S**: *single responsibility principle* (princípio da responsabilidade única).
- **O**: *open/closed principle* (princípio do aberto/fechado).
- **L**: Liskov *substitution principle* (princípio da substituição de Liskov).
- **I**: *interface segregation principle* (princípio da segregação de interfaces).
- **D**: *dependency inversion principle* (princípio da inversão de dependência).

Cada um desses princípios tem suas motivações e suas técnicas, e, desde que foram propostos em 2000, eles revolucionaram o mundo do desenvolvimento orientado a objetos por trazerem recomendações que estimulam o desenvolvedor a criar sistemas de manutenção, compreensão e flexibilidade melhores, de maneira que o *software* possa crescer em tamanho, mas reduzindo sua complexidade.

Esses princípios são genéricos e, portanto, podem ser aplicados a qualquer linguagem orientada a objetos. Veja a seguir exemplos de uso com Java:

S: princípio da responsabilidade única

Princípio: uma classe deve ter um, e somente um, motivo para mudar.

O princípio define que uma classe deve ser especializada em um único assunto, em uma única tarefa ou em uma única ação para executar o sistema. Mais do que isso, pode-se interpretar “responsabilidade”, nesse princípio, como “motivo para alterar uma classe” – e este deve ser único.

Trata-se de um princípio diretamente ligado à propriedade de coesão de classe. O primeiro exemplo com a classe **Venda** ilustra o princípio da responsabilidade única sendo violado, assim como ocorre neste exemplo:

```
public class Orcamento {
  public double calculaSomaTotal(){ /* código */ }
  public List listaItens(){ /* código */ }
  public void adicionaItem(){ /* código */ }
  public void removeItem(){ /* código */ }
  public void imprimeOrcamento(){ /* código */ }
  public void mostraOrcamentoEmTela(){ /* código */ }
  public void mostraTotalEmTela(){ /* código */ }
  public void gravar(){ /* código */ }
  public void atualizar(){ /* código */ }
  public void excluir(){ /* código */ }
  public void obtemTodos(){ /* código */ }
  public void encerraConexao(){ /* código */ }
}
```

A classe **Orcamento**, apesar de trazer operações que, de certa maneira, são relativas a orçamento, é um método de responsabilidades muito distintas.

- Os métodos **calculaSomaTotal()**, **listaItens()**, **adicionaItens()** e **removeItem()** são relativos, de fato, à construção e à consulta de objetos de orçamentos.
- Os métodos **imprimeOrcamento()** e **mostraOrcamentoEmTela()** são responsáveis pela visualização de dados (seja em impressão, seja em tela).
- Os métodos **gravar()**, **atualizar()** e **excluir()** são operações de persistência – muito possivelmente de banco de dados.

Então, o ideal seria separar a classe **Orcamento** em três, como a seguir:

```
public class Orcamento {
  public double calculaSomaTotal(){ /* código */ }
  public List listaItens(){ /* código */ }
  public void adicionaItem(){ /* código */ }
  public void removeItem(){ /* código */ }
}

public class OrcamentoVisualizacao {
  public void imprimeOrcamento(){ /* código */ }
  public void mostraOrcamentoEmTela(){ /* código */ }
  public void mostraTotalEmTela(){ /* código */ }
}

public class OrcamentoRepositorio {
  public void gravar(){ /* código */ }
  public void atualizar(){ /* código */ }
  public void excluir(){ /* código */ }
  public void obtemTodos(){ /* código */ }
  public void encerraConexao(){ /* código */ }
}
```

No trabalho de detecção desses problemas de violação do princípio da responsabilidade única (e também dos outros princípios), podem acontecer refatorações, analisando-se a classe pronta, como é o caso de **Orcamento**.

A responsabilidade única pode ser violada também nos próprios métodos. Suponha um método que tenha que obter todos os orçamentos e somar os totais destes. Nesse momento, ignore os detalhes do código de conexão com banco de dados, pois você aprenderá isso posteriormente no curso. Atente-se às diversas responsabilidades que o método tem:

```
  public double calculaTotalOrcamentos(){
    /*1) conexão com banco de dados*/
    Connection conexao = DriverManager.getConnection("jdbc:mysql://localhos
t/meuBancoDeDados");
    PreparedStatement stmt = conexao.prepareStatement("select * from orcame
nto");
    ResultSet rs = stmt.executeQuery();

    /*2) cálculo de total */
    double total = 0;
    while (rs.next()) {
        total = total + rs.getDouble("valor");
    }

    /*3) visualização */
    System.out.println("O valor total de orçamentos é de R$" + total);

    /*4) encerramento da conexão */
    rs.close();
    stmt.close();
    conexao.close();
  }
```

Primeiramente, o método precisa se conectar ao banco de dados, depois calcular o total (que é a motivação principal do método), mostrar esse total na tela e, ao fim, encerrar a conexão. São muitas coisas para apenas um método.

É possível separar essas ações em métodos separados (mesmo em classes separadas):

```
public double calculaTotalOrcamentos(){
  /*1) conexão com banco de dados*/
  OrcamentoRepositorio bancoDados = new OrcamentoRepositorio();
  List lista = bancoDados.obtemTodos();

  /*2) cálculo de total */
  double total = 0;
  for(Orcamento o : lista){
      total = total + o.valor;
  }

  /*3) visualização */
  OrcamentoVisualizacao visualizacao = new OrcamentoVisualizacao();
  visualizacao.mostraTotalEmTela(total);

  /*4) encerramento da conexão */
  bancoDados.encerra();
}
```

Note que o código fica mais legível, pois é possível ignorar, por exemplo, como de fato acontece a conexão com o banco de dados – é o método **obtemTodos()** da classe **OrcamentoRepositorio** que sabe como fazer tal conexão. Você está treinando aqui também a propriedade de ocultamento de informação, vista anteriormente.

O: princípio do aberto/fechado

Princípio: classes devem estar abertas para extensão, mas fechadas para modificação.

O princípio estimula o programador a pensar se, em vez de modificar os códigos internos de uma classe, não seria melhor estendê-la de alguma maneira, seja por hierarquia, seja por associações. Padrões de projeto podem ajudar a construir um código que seja extensível.

Valente (2020, p. 176) resume que “o princípio aberto/fechado tem como objetivo a construção de classes flexíveis e extensíveis, capazes de se adaptarem a diversos cenários de uso, sem modificações em seu código fonte”. Isso quer dizer que, quando algum comportamento novo for incluído em uma classe, é preferível estender o código-fonte original a alterá-lo.

O exemplo a seguir contém classes para um sistema que envia mensagens de *e-mail*. A solução não está obedecendo ao princípio do aberto/fechado. Veja o código e tente identificar o porquê:

```
  public class Email {
    private String destinatario;
    private String conteudo;
    private String tipo;

    public String getTipo() {
        return tipo;
    }

    public void setTipo(String tipo) {
        this.tipo = tipo;
    }

    public String getDestinatario() {
        return destinatario;
    }

    public void setDestinatario(String destinatario) {
        this.destinatario = destinatario;
    }

    public String getConteudo() {
        return conteudo;
    }

    public void setConteudo(String conteudo) {
        this.conteudo = conteudo;
    }

    public void enviaEmailTexto(){
        System.out.println("Enviando " + conteudo + " para " + destinatario
+ " como texto sem formatação");
    }

    public void enviaEmailHTML(){
        System.out.println("Enviando " + conteudo + " para " + destinatario
+ " como HTML");
    }
  }

  public class EnviadorEmail {

    public void enviar(List<Email> emails) {
        for(Email email : emails) {
            switch(email.getTipo()) {
                case "Texto":
                    email.enviaEmailTexto();
                    break;
```

```
                case "HTML":
                    email.enviaEmailHTML();
                    break;
            }
        }
    }

}
```

Há uma classe que cuida de *e-mails* de vários tipos (HTML [*hypertext markup language*] e texto simples, por enquanto). Se você quisesse implementar o tipo "*e-mail* criptografado", o que deveria fazer? Observe os passos a seguir.

Crie um método **enviaEmailCriptografado()** em **Email**:

```
  public void enviaEmailCriptografado(){
    System.out.println("Enviando " + conteudo + " para " + destinatario + "
com criptografia");
  }
```

E, na classe **EnviadorEmail**, altere o método **enviar()**:

```
public void enviar(List<Email> emails)
{
    for(Email email : emails)
    {
        switch(email.getTipo())
        {
            case "Texto":
                email.enviaEmailTexto();
                break;

            case "HTML":
                email.enviaEmailHTML();
                break;

            case "Criptografado":
                email.enviaEmailCriptografado();
                break
        }
    }
}
```

Está sendo, portanto, violado o princípio do aberto/fechado duas vezes, pois foi necessário fazer interferências diretas no código das classes apenas para um tipo novo de *e-mail*. Problemas poderiam surgir, pois, ao modificar o código já existente para incluir uma nova funcionalidade, há um grande risco de introduzir problemas no que já estava funcionando bem.

Para adaptar essa situação ao princípio, a proposta é criar uma superclasse abstrata para **Email**, com classes derivadas que implementem sua forma de enviar e ajustar o método **enviar()** da classe **EnviadorEmail**. Veja:

```
public abstract class Email {
  protected String destinatario;
  protected String conteudo;
  protected String tipo;

  public String getTipo() {
      return tipo;
  }

  public void setTipo(String tipo) {
      this.tipo = tipo;
  }

  public String getDestinatario() {
      return destinatario;
  }

  public void setDestinatario(String destinatario) {
      this.destinatario = destinatario;
  }

  public String getConteudo() {
      return conteudo;
  }

  public void setConteudo(String conteudo) {
      this.conteudo = conteudo;
  }

  public abstract void enviaEmail();
}
```

```
public class EmailTexto extends Email{

  @Override
  public void enviaEmail() {
      System.out.println("Enviando " + conteudo + " para " + destinatario
+ " como texto sem formatação");
  }

}
```

```
  public class EmailHTML extends Email {

    @Override
    public void enviaEmail() {
        System.out.println("Enviando " + conteudo + " para " + destinatario
+ " como HTML");
    }

  }
```

```
  public class EmailCripto extends Email{

    @Override
    public void enviaEmail() {
        System.out.println("Enviando " + conteudo + " para " + destinatario
+ " com criptografia");
    }

  }
```

Tem-se então uma classe-base **Email** que define o comportamento **enviaEmail()**, implementado por cada uma de suas derivadas – **EmailTexto**, **EmailHTML** e **EmailCripto**. Agora o código de **EnviadorEmail** ficará muito simples:

```
public class EnviadorEmail {

  public void enviar(List emails) {
      for(Email email : emails) {
          email.enviaEmail();
      }
  }

}
```

Observe que, com base nessas alterações, as quais tornam a solução condizente com o princípio do aberto/fechado, é possível criar novos tipos de *e-mail*, como “*e-mail* de imagens”, sem precisar alterar o código de nenhuma das outras classes. A solução (aberto) está sendo estendida sem modificar seu código já implementado (fechado).

Note que você também está praticando a separação de responsabilidades.

L: princípio da substituição de Liskov

Princípio: uma classe derivada deve ser substituída pela sua superclasse sem quebrar a aplicação.

Barbara Liskov, professora no Instituto de Tecnologia de Massachusetts, definiu em 1987 que, em um código orientado a objetos, deve ser possível usar qualquer classe derivada em vez de uma superclasse e manter o comportamento original, sem modificações. Para isso, é necessário que os objetos derivados se comportem da mesma maneira que a superclasse.

Como se trata aqui de “contratos”, ou seja, comportamentos (métodos) da superclasse implementados ou sobrescritos em uma subclasse, dois pontos são importantes:

- Os parâmetros do método da classe derivada devem ser os mesmos, de mesmo tipo e mesma ordem que aparecem na superclasse.
- O tipo de retorno do método deve ser idêntico ao tipo da superclasse.

Caso um desses itens seja modificado, haverá um comportamento divergente na subclasse. Veja um exemplo de classes que aplicam o princípio da substituição de Liskov:

```
public class ClasseA {
  public String getMensagem(){
      return "Esta é a classe A";
  }
}

public class ClasseB extends ClasseA {
  @Override
  public String getMensagem(){
      return "Esta é a classe B";
  }
}
```

A **ClasseB** deriva da **ClasseA**, mantendo todas as suas características de comportamento.

```
public class Teste {

  public static void imprime(ClasseA a){
      System.out.println(a.getMensagem());
  }

  public static void main(String[] args) {
      ClasseA objetoA = new ClasseA();
      ClasseB objetoB = new ClasseB();

      imprime(objetoA);
      imprime(objetoB);
  }
}
```

É possível repassar ao método **imprime()** por parâmetro, com segurança, tanto um objeto de **ClasseA** quanto um de sua derivada, a **ClasseB**. Isso afirma a adesão do código à regra de Liskov.

O exemplo das classes de **Email** visto anteriormente também obedece ao princípio da substituição de Liskov, pois o comportamento original de **Email** é mantido nas classes derivadas.

O princípio da substituição de Liskov ajuda ainda a utilizar o polimorfismo com mais confiança. Podem-se usar classes derivadas referindo-se à sua classe-base sem preocupações com resultados inesperados.

I: princípio da segregação de interface

Princípio: uma classe não deve ser forçada a implementar interfaces e métodos que não serão úteis a ela.

O objetivo geral é evitar que uma classe dependa de interfaces com métodos que não vão ser usados. Em vez de uma interface com muitos métodos, é melhor quebrá-la em várias interfaces. Visto que uma classe que implementa a interface deve desenvolver código para todos os seus métodos, se a interface tem muitos comportamentos é mais provável que alguns deles não sejam importantes para a classe.

Como exemplo, será analisada uma interface que simula operações de impressoras e multifuncionais:

```
public interface Impressora {
  public void imprimir();
  public void escanear();
  public void enviarFax();
  public void copiar();
}
```

Imagine que agora o objetivo é criar uma classe concreta para representar uma impressora de jato de tinta e outra classe para representar uma multifuncional a *laser*:

```
public class MultifuncionalLaser implements Impressora{

  @Override
  public void imprimir() {
      System.out.println("Imprimindo conteúdo");
  }

  @Override
  public void escanear() {
      System.out.println("Escaneando dados");
  }

  @Override
  public void enviarFax() {
        System.out.println("Enviando fax");
  }

  @Override
  public void copiar() {
        System.out.println("Escaneando e imprimindo cópia");
  }

}
```

A classe **MultifuncionalLaser** consegue implementar todos os métodos da interface **Impressora**. Já a classe **ImpressoraJatoDeTinta** precisará implementar métodos de que não necessita, só porque estão na interface **Impressora** – afinal, uma impressora de jato de tinta simples apenas imprimirá arquivos.

```
public class ImpressoraJatoDeTinta implements Impressora {

  @Override
  public void imprimir() {
    System.out.println("Imprimindo conteúdo com jato de tinta");
  }

  @Override
    public void escanear() {
  }

  @Override
    public void enviarFax() {
  }

  @Override
    public void copiar() {
  }

}
```

É possível adaptar esse código para o princípio da segregação de interfaces quebrando a interface **Impressora** em duas. Veja a solução completa:

```
public interface Impressora {
  public void imprimir();
}

public interface Multifuncional extends Impressora {
  public void escanear();
  public void enviarFax();
  public void copiar();
}
```

Note aqui que é possível estender uma interface com outra. É uma solução opcional (afinal, uma classe pode implementar mais de uma interface), mas útil em situações como essa.

```
public class MultifuncionalLaser implements Multifuncional{

  @Override
  public void imprimir() {
      System.out.println("Imprimindo conteúdo");
  }

  @Override
  public void escanear() {
      System.out.println("Escaneando dados");
  }

  @Override
  public void enviarFax() {
        System.out.println("Enviando fax");
  }

  @Override
  public void copiar() {
        System.out.println("Escaneando e imprimindo cópia");
  }

}
```

A classe **MultifuncionalLaser** agora implementa a interface **Multifuncional**, com todos os métodos necessários.

```
public class ImpressoraJatoDeTinta implements Impressora {

  @Override
  public void imprimir() {
      System.out.println("Imprimindo conteúdo com jato de tinta");
  }

}
```

Já a classe **ImpressoraJatoDeTinta** implementa agora apenas **Impressora**, com o comportamento básico (e suficiente para essa classe) de **imprimir()**.

No próximo exemplo, você verá uma comparação entre uma versão não condizente com o princípio (à esquerda) e uma versão adaptada (à direita). A interface representa operações de pagamento, sejam com dinheiro, sejam com cartão ou Pix:

```
public interface Pagavel {

  public void realizarPagamento();

  public void recuperarDados(String chave);

  public void adicionarDados(String numeroCartao, int cvv);

}
```

```
public interface Pagavel {
  public void realizarPagamento();
}

public interface PagavelPix {
  public void recuperarDados(String chave);
}

public interface PagavelCartao {
  public void adicionarDados(String numeroCartao, int cvv);
}
```

Na interface **Pagavel**, à esquerda, seria necessário sempre implementar métodos relativos a Pix e cartões, mesmo se a classe se referisse a dinheiro ou boleto, por exemplo. A implementação à direita, por outro lado, dá liberdade para incluir apenas as operações necessárias. Veja um exemplo de classe concreta para registrar pagamento com cartão de débito:

```
public class PagamentoCartaoDebito implements Pagavel, PagavelCartao{

  @Override
  public void realizarPagamento() {
      //operações de pagamento, conexão com a operadora etc
  }

  @Override
  public void adicionarDados(String numeroCartao, int cvv) {
      //gravação dos dados do cartão, validações etc
  }

}
```

D: princípio da inversão de dependência

Princípio: a classe deve depender de abstrações, e não de implementações concretas.

A ideia é que uma classe que utiliza outra classe para suas tarefas dependa de uma interface em vez de uma classe concreta. Isso para que, se necessário, seja possível trocar a implementação sem muitos prejuízos. O conceito é, portanto, “inverter”, trocar as dependências: depender de interfaces em vez de classes concretas.

Por exemplo, imagine que a classe tem um atributo do tipo **ArrayList**. Essa classe é, assim, dependente de uma implementação específica de lista – a classe concreta **ArrayList**. Se fosse necessário trocar essa implementação por outra, poderia haver problemas se, por exemplo, a nova implementação não tivesse algum método próprio de **ArrayList**.

A solução, nesse caso, seria depender de **List**, interface implementada por **ArrayList**. Assim, no futuro, poderia ser trocada **ArrayList** por qualquer outra classe que implemente **List**. Trata-se de uma preocupação com a manutenção do código. Reveja o exemplo da classe **Agenda** para ilustrar essa situação:

```
public class Agenda {
  private ArrayList<Pessoa> contatos;

  public Agenda() {
      contatos = new ArrayList<Pessoa>();
  }
}
```

```
public class Agenda {
  private List<Pessoa> contatos;

  public Agenda() {
      contatos = new ArrayList<Pessoa>();
  }
}
```

De certa maneira, a classe **Agenda** adaptada para obedecer ao princípio da inversão de dependência (à direita) ainda tem alguma dependência de **ArrayList**, pois usa esse tipo no construtor. Porém, a alteração se tornará mínima caso seja necessário trocar **ArrayList** por **LinkedList** ou **Vector**, por exemplo, que também implementam **List** – na verdade, a alteração aconteceria no construtor e em mais nenhuma parte do código da classe.

Outro exemplo em que pode ser observado o princípio da inversão de dependência é na classe **EnviadorEmail**, vista anteriormente. Reveja o código:

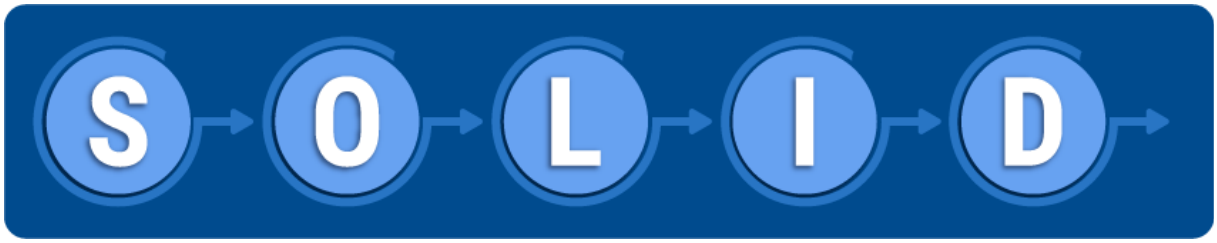
```
public class EnviadorEmail {

  public void enviar(List<Email> emails) {
      for(Email email : emails) {
          email.enviaEmail();
      }
  }

}
```

Note a dependência de duas abstrações no método **enviar()**: a interface **List** e o tipo **Email**, que é uma classe abstrata. Em um primeiro momento, seria possível apenas enviar *e-mails* de texto simples e, com isso, usar **List<EmailTexto>**, dependendo da classe concreta. No entanto, como os e-mails são variáveis, de diversos tipos, é mais recomendável aplicar List<Email>, como no exemplo.

Daí surge outra observação quanto a esse princípio: ele só é realmente útil quando se trata de um tipo que tem outros subtipos ou outras categorias, como é o caso dos *e-mails* ou das classes de impressoras vistas na subseção anterior. Caso a dependência se dê por uma classe que não tenha previsão de variação ou categorização, não se devem criar interfaces apenas para satisfazer o princípio.



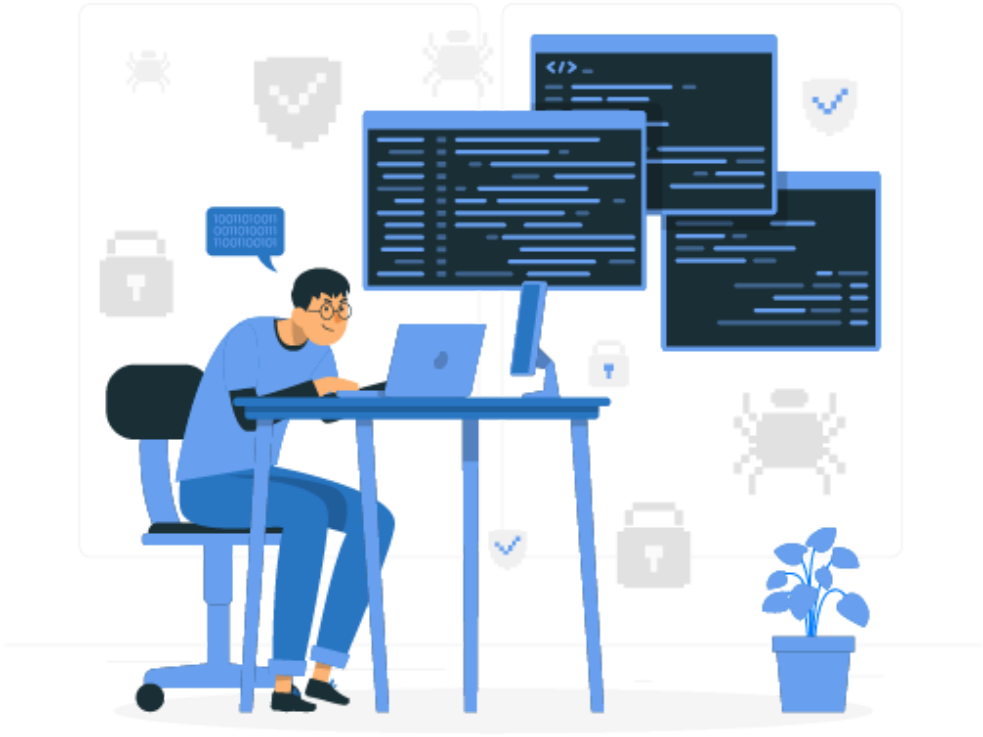
Aplicação dos princípios SOLID

Os princípios SOLID são recomendações, e não regras, para o desenvolvimento de *software*. Como ganharam notoriedade na comunidade de desenvolvimento – e no mercado de trabalho, consequentemente –, é importante que o desenvolvedor de código orientado a objetos os conheça, mas que também saiba avaliar quais e quando aplicar.

Também é recomendado que, após conhecê-los, o desenvolvedor os pratique no dia a dia, aplicando-os corriqueiramente, de modo que, em certo momento, estará usando as boas práticas de modo natural, sem pensar muito.

As vantagens da aplicação dos princípios SOLID são comprovadas pela notoriedade que eles tomaram, ajudando a tornar o *software* mais robusto, flexível, tolerante a mudanças e inteligível para novos desenvolvedores.

Injeção de dependência



Como já comentado algumas vezes, quando uma classe **A** necessita de uma classe **B**, diz-se que há uma dependência entre **A** e **B**. Essa dependência pode ser baseada em um atributo do tipo B na classe A ou em um parâmetro do tipo B em um método de A.

Normalmente, é usado o comando **new** para criar uma nova instância de uma classe da qual se depende. Assim, seria natural que, na classe **A**, houvesse uma linha como a seguinte:

```
atributoB = new B();
```

Essa instanciação se dá, muitas vezes, no construtor. Há uma maneira, no entanto, de definir essas instâncias de dependência com base em algo externo à classe. Isso significa que outra classe ou outro módulo define o objeto concreto à referência presente em uma classe. Esse processo é chamado de **injeção de dependência**.

Veja um exemplo reconsiderando as classes de impressoras vistas anteriormente:

```
public class MultifuncionalLaser implements Multifuncional{

  @Override
  public void imprimir() {
      System.out.println("Imprimindo conteúdo");
  }

  @Override
  public void escanear() {
      System.out.println("Escaneando dados");
  }

  @Override
  public void enviarFax() {
        System.out.println("Enviando fax");
  }

  @Override
  public void copiar() {
        System.out.println("Escaneando e imprimindo cópia");
  }

}

public class ImpressoraJatoDeTinta implements Impressora {

  @Override
  public void imprimir() {
      System.out.println("Imprimindo conteúdo com jato de tinta");
  }

}
```

Imagine que há uma nova classe **Relatorio** que permite que dado relatório gerado no sistema seja impresso. Essa classe terá dependência de uma impressora. A princípio, seria possível pensar em usar uma das impressoras.

```
public class Relatorio {
  private ImpressoraJatoDeTinta impressora;

  public Relatorio(){
    impressora = new ImpressoraJatoDeTinta();
  }

  public void gerarRelatorio(){ /* codigo */ }

  public void imprimirRelatorio() {
    gerarRelatorio();
    impressora.imprimir();
  }
}
```

Observe, nas linhas marcadas em azul, como se dá a dependência da classe **Relatorio** com a classe **ImpressoraJatoDeTinta**. Pelo princípio da inversão de dependência, sabe-se que o ideal seria, no lugar de uma referência à classe concreta, usar uma referência à interface **Impressora**.

```
public class Relatorio {
  private Impressora impressora;

  public Relatorio(){
      impressora = new ImpressoraJatoDeTinta();
  }

  public void gerarRelatorio(){ /* codigo */ }

  public void imprimirRelatorio() {
      gerarRelatorio();
      impressora.imprimir();
  }
}
```

A dependência determinada pelo atributo **impressora** é resolvida no construtor com a seguinte linha:

```
impressora = new ImpressoraJatoDeTinta();
```

Isso quer dizer que **Relatorio** sempre precisará usar uma **ImpressoraJatoDeTinta**. A injeção de dependência flexibiliza a instanciação, permitindo que outra classe decida de fato. Uma das maneiras de deixar a classe **Relatorio** pronta para a injeção de dependência é, em vez de instanciar a impressora diretamente, receber a instância pronta por parâmetro no construtor.

```
public class Relatorio {
  private Impressora impressora;

  public Relatorio(Impressora objetoImpressora){
      impressora = objetoImpressora;
  }

  public void gerarRelatorio(){ /* codigo */ }

  public void imprimirRelatorio() {
      gerarRelatorio();
      impressora.imprimir();
  }
}
```

Assim, outra classe poderá verificar a necessidade de uma impressora ou outra e injetá-la no objeto de **Relatório**.

```
public class Principal {

  public static void main(String[] args) {
      Scanner entrada = new Scanner(System.in);

      Impressora algumaImpressora;
      String opcao = entrada.nextLine();

      if(opcao.equals("Laser")){
          algumaImpressora = new MultifuncionalLaser();
      }
      else {
          algumaImpressora = new ImpressoraJatoDeTinta();
      }

      Relatorio rel = new Relatorio(algumaImpressora);
      rel.imprimirRelatorio();
  }
}
```

Nesse último exemplo, está sendo instanciada uma impressora **MultifuncionalLaser** ou uma **ImpressoraJatoDeTinta** de acordo com uma entrada informada pelo usuário. A decisão de qual objeto concreto instanciar pode usar dos mais variados critérios além desse. O ponto é que o objeto foi instanciado e informado ao construtor de **Relatorio**. A dependência foi injetada a partir do código principal.

Outra maneira de injetar dependência é por métodos *setter*:

```
public class Relatorio {
  private Impressora impressora;

  public void setImpressora(Impressora impressora) {
      this.impressora = impressora;
  }

  public void gerarRelatorio(){ /* codigo */ }

  public void imprimirRelatorio() {
      gerarRelatorio();
      if(impressora != null)
          impressora.imprimir();
  }
}
```

Nesse caso, dispensou-se o construtor e manteve-se apenas um método **setImpressora()**. O código responsável pela injeção deverá usar esse método.

```
public class Principal {

  public static void main(String[] args) {
      Scanner entrada = new Scanner(System.in);

      Impressora algumaImpressora;
      String opcao = entrada.nextLine();

      if(opcao.equals("Laser")){
          algumaImpressora = new MultifuncionalLaser();
      }
      else {
          algumaImpressora = new ImpressoraJatoDeTinta();
      }

      Relatorio rel = new Relatorio();
      rel.setImpressora(algumaImpressora);
      rel.imprimirRelatorio();
  }
}
```

Há mais flexibilidade nessa implementação, mas é necessário cuidado. Isso porque o código que utiliza a classe **Relatorio** pode simplesmente não usar o método **setImpressora()** antes de chamar o método **imprimirRelatorio()**, que usa o objeto **Impressora** do qual **Relatorio** é dependente (daí a necessidade de validação sobre valor nulo no objeto **impressora** em **imprimirRelatorio()**).

Esse é um conceito importante, mas que será melhor explorado ao aplicar ferramentas e *frameworks* específicos em Java. Vale a pena comentar que praticamente todas as linguagens de programação orientadas a objetos têm algum mecanismo próprio de injeção de dependência, e as mais modernas aplicam consistentemente essa técnica em suas bibliotecas.

Encerramento

Entender as propriedades e os princípios para codificação de sistemas orientados a objetos é mais que um diferencial na profissão do desenvolvedor, é um conhecimento fundamental. A difusão dos conceitos de SOLID tem tornado essas práticas presentes ou ao menos observadas em grande parte dos projetos de *software* atuais.

É importante, no entanto, manter o olhar crítico às particularidades do sistema em desenvolvimento, pois, em algumas situações, a aplicação de um ou outro princípio não é desejável ou adequada, trazendo mais trabalho e complexidade do que benefícios.