Desenvolvimento de Sistemas

Padrões de projeto: conceitos, principais padrões, aplicabilidade, tendências de mercado

Introdução aos padrões de projeto

Como visto anteriormente no curso, a programação orientada a objetos permite criar *softwares* mais adaptados à reutilização, à manutenção, à atualização do código e tem um poder muito maior de escalabilidade se comparada ao paradigma de programação estruturada.

Porém, em um mundo real e complexo, existem milhões de formas de resolver os problemas que aparecem no mundo da programação, e, com o passar dos anos, foram se criando formas que demonstram os melhores caminhos a serem seguidos conformes as necessidades do sistema.

Para se ter uma ideia, na década de 1970, um arquiteto chamado Christopher Alexander buscou padrões para um projeto de construção considerado bem elaborado e de alta qualidade. A partir daí, construiu o primeiro catálogo de padrões, que descrevia boas práticas que poderiam ser seguidas para agilizar e qualificar diversos tipos de projeto já feitos anteriormente e resolver problemas complexos na área da arquitetura.

Então, em 1994, quatro amigos se inspiraram em Christopher para adaptar os ensinamentos dele ao desenvolvimento de *software*. Para isso, pesquisaram soluções para diversos projetos de *software*, e daí surgiu o primeiro catálogo de padrões de

projeto, ou *design patterns*, chamado de *Padrões de Projeto: Soluções Reutilizáveis de* Software *Orientado a Objetos* – em vez de padrões de projeto voltados à arquitetura, eram voltados a *softwares*.

Esse catálogo deixou os quatro amigos Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides conhecidos mundialmente como GoF, ou *gang of four* (em português: “gangue dos quatro”), e se tornou uma referência quando se fala em estudo e utilização de padrões de projeto de *software*.

Conceitos de padrões de projeto

Pode-se dizer que um padrão de projeto é uma descrição de um problema que acontece frequentemente em alguma área – mais especificamente na computação, no caso deste curso. É uma forma de solucionar o problema que já aconteceu algumas vezes anteriormente.

Isso tudo gera uma grande vantagem ao programador, que não precisa mais recriar a programação, mas apenas utilizar boas práticas e experiências adquiridas por outros programadores anteriormente na solução de um problema parecido.

A ideia é ter definido o cerne da solução para um problema genérico, com sugestões de práticas de programação, de modo que outras pessoas possam usar essas sugestões para problemas semelhantes.

Algo muito importante que deve ser entendido é que, quando se escolhe um padrão de projeto para o sistema, não será encontrado um código pronto, no qual apenas bastaria um “copia e cola” para os problemas estarem solucionados. É necessário considerar os caminhos traçados anteriormente que podem ser adequados à realidade do projeto.

Na área de *softwares*, os padrões de projeto foram divididos pela GoF em **criacionais**, **estruturais e comportamentais**, e, dentro dessas categorias, foram catalogados 23 tipos de padrões de projeto. Obviamente, um programador não conhecerá a fundo todos os padrões, mas deve, sim, conhecer os principais.

Este material, portanto, explora os principais padrões de projeto dentro das três categorias citadas. São elas:

Padrões criacionais

Propõem soluções flexíveis para criação de objetos. São padrões de projeto para situações em que é preciso ter um controle maior na hora de criar um objeto, e não simplesmente utilizar a palavra reservada “*new*” como se faz normalmente.

Com as categorias criacionais, é possível definir caminhos a serem seguidos na criação dos objetos do sistema, especificando regras para tanto e permitindo uma independência do sistema com relação à forma como os objetos são criados e compostos. São exemplos de padrões criacionais: *abstract factory, factory method, singleton, builder* e *prototype*.

Padrões estruturais

São padrões que ajudam a dar flexibilidade à composição de classes e objetos, trabalhando diretamente com as associações e as dependências entre classes. São padrões estruturais: *proxy, adapter, facade, decorator, bridge, composite* e *flyweight*.

Padrões comportamentais

Operam na interação e na divisão de responsabilidades entre classes e objetos. São padrões comportamentais: *strategy, observer, template method, visitor, chain of responsibility, command, interpreter, iterator, mediator, memento* e *state*.

Este material se concentra em alguns padrões mais relevantes, mas você pode pesquisar mais informações sobre os demais.

Principais padrões de projeto

Com base no catálogo completo de padrões, a seguir serão abordados alguns dos mais relevantes.

Padrões criacionais

Padrão *abstract factory*

É um padrão que permite produzir objetos derivados de um tipo abstrato sem especificar suas classes concretas, ou seja, o código que utilizará um objeto não precisa dizer explicitamente de que classe este deve ser; precisa apenas saber de que classe abstrata deriva ou que interface implementa.

O termo "*factory*" (em português: "fábrica") do nome se refere a uma classe que criará outros objetos segundo critérios definidos. Todos esses objetos precisam implementar uma interface ou derivar uma classe específica, para que a classe "fábrica" consiga retornar uma referência a objeto genérica. Essa referência apontará para uma de várias possibilidades de classe concreta que implemente a interface ou a classe genérica.

O desafio é conseguir extrair a lógica dos objetos para um *abstract factory* e assegurar uma implementação para cada contexto, garantindo que todos os objetos criados tenham algum tipo de relacionamento.

Problema

Você foi designado para criar um sistema de locação de jogos, precisando criar, assim, objetos que remetam a jogos de esporte ou RPG (*role-playing game*) e que possam ser jogos novos ou clássicos. Primeiramente, observe as classes que seriam correspondentes aos jogos. Uma classe abstrata **Games** será a base de tudo:

```
abstract class Games {
  private String nome;

  public Games(String nome) {
      this.nome = nome;
  }
  public String toString() {
      return nome;
  }
}
```

Com base nela, você terá a classe **Esporte** e **Rpg**, que também são abstratas:

```
abstract class Esportes extends Games {
  public Esportes(String nome) {
      super(nome);
//o comando super(), faz com que chamamos o mesmo atributo da
Classe pai.
  }
}

abstract class Rpg extends Games {
  public Rpg(String nome) {
      super(nome);
  }
}
```

Por fim, tem-se as classes concretas. Considere recentes os jogos FIFA e Elden Ring e clássicos os jogos Winning Eleven e Zelda:

```
class WiningEleven extends Esportes {
  public WiningEleven() {
      super("Winning Eleven");
  }
}

class Fifa extends Esportes {
  public Fifa() {
      super("Fifa");
  }
}
```

```
class Zelda extends Rpg {
  public Zelda() {
      super("Zelda");
  }
}

class EldenRing extends Rpg {
  public EldenRing() {
      super("Elden Ring");
  }
}
```

Note que as classes são derivadas de **Esporte** ou **Rpg**.

O sistema precisa de uma classe para criar e armazenar os jogos sugeridos. O primeiro ímpeto ao programar seria incluir instanciações com a palavra reservada **new**:

```
public class SugestaoGame {
  private Esportes gameEsporte;
  private Rpg gameRpg;

  public Esportes getGameEsporte() {
      return gameEsporte;
  }

  public Rpg getGameRpg() {
      return gameRpg;
  }

  public void gerar(byte opc)
  {
      if(opc == 1)
      {
          gameEsporte = new Fifa();
          gameRpg = new EldenRing();
      }
      else
      {
          gameEsporte = new WiningEleven();
          gameRpg = new Zelda();
      }
  }
}
```

Na classe principal, você teria algo assim:

```
  public class AbsFac {

    public static void main(String [] args) {
          Scanner sc = new Scanner(System.in);
          System.out.println("Informe 1-Games Antigos ou 2-Games novos");
          byte opc = sc.nextByte();

          SugestaoGame sugestao = new SugestaoGame();
          sugestao.gerar(opc);

          System.out.println("Esportes: " + sugestao.getGameEsporte().toStr
ing());
          System.out.println("Rpg: " + sugestao.getGameRpg().toString());
    }
  }
```

Não há nada de errado com essa abordagem, mas ela é pouco flexível. Imagine se você quisesse incluir um novo tipo de jogo (de aventura, por exemplo) ou se, além de **SugestaoGames**, você tivesse vários pontos de instanciação no sistema e precisasse trocar **Fifa** ou algum outro jogo mais recente. O trabalho de ajuste poderia ser grande, levando certamente a falhas.

Solução

A solução é criar métodos que criam e retornam objetos de determinada classe – neste caso partindo da classe **Games** – e que também ocultam o tipo desses objetos por meio de uma interface. Isso significa que tudo será tratado como **Game**, mas outro objeto saberá decidir que classe instanciar.

O primeiro passo é criar uma interface para as fábricas (essa é a fábrica abstrata em si), a qual indicará os métodos que toda fábrica deve implementar.

```
interface Modelo {
  Esportes getEsportes();
  Rpg getRpg();
}
```

Passa-se à implementação da interface **Modelo**, na qual, pelos comandos **new**, serão criados os objetos concretos. Veja o exemplo com a classe **FabricaAntigos** para construção dos *games* antigos:

```
class FabricaAntigos implements Modelo {
  public Esportes getEsportes() {
      return new WiningEleven();
  }
  public Rpg getRpg() {
      return new Zelda();
  }
}
```

Também foi criada uma classe **FabricaNovos** para construção dos *games* novos:

```
class FabricaNovos implements Modelo {
  public Esportes getEsportes() {
      return new Fifa();
  }
  public Rpg getRpg() {
      return new EldenRing();
  }
}
```

A classe **SugestaoGame** será positivamente afetada da seguinte maneira:

```
public class SugestaoGame {
  private Esportes gameEsporte;
  private Rpg gameRpg;
  private Modelo fabricaModelo;

  public SugestaoGame(Modelo fabrica)
  {
      fabricaModelo = fabrica;
  }

  public Esportes getGameEsporte() {
      return gameEsporte;
  }

  public Rpg getGameRpg() {
      return gameRpg;
  }

  public void gerar()
  {
      gameEsporte = fabricaModelo.getEsportes();
      gameRpg = fabricaModelo.getRpg();
  }
}
```

Veja que a classe agora não precisa mais saber a opção escolhida ou a forma de instanciar concretamente um jogo. Basta que ela confie na fábrica que lhe será fornecida via construtor. Na classe principal, o código fica da seguinte forma:

```
public class AbsFac {

  public static void main(String [] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Informe 1-Games Antigos ou 2-Games novos");
        byte opc = sc.nextByte();

        Modelo modelo = null;
        switch (opc) {
          case 1: modelo = new FabricaAntigos(); break;
          case 2: modelo = new FabricaNovos(); break;
        }

        SugestaoGame sugestao = new SugestaoGame(modelo);
        sugestao.gerar(opc);

        System.out.println("Esportes: " + sugestao.getGameEsporte().toStr
ing());
        System.out.println("Rpg: " + sugestao.getGameRpg().toString());
  }
}
```

O código principal funciona como uma configuração para a classe **SugestaoGame**, definindo a fábrica que será utilizada e retirando essa responsabilidade da classe de sugestão.

O fato é que, com essa implementação, o usuário não precisa entender toda implementação que existe nas famílias de classes nem modificar os objetos instanciados. Além disso, torna-se mais fácil e seguro trocar as classes concretas para jogos ou mesmo adicionar novos jogos e gêneros.

Crie uma estrutura parecida com o código mostrado, utilizando as técnicas aprendidas no padrão *abstract factory*, porém mudando o tema para jogos de tabuleiro.

Padrão *factory method*

É um padrão de projeto criacional, que resolve o problema de criar objetos de produtos sem especificar suas classes concretas. Bastante semelhante ao *abstract factory*, a diferença aqui é que a criação fica concentrada em um único método em vez de em várias classes. O *factory method* serve bem para situações mais simples de instanciação, que não constituem toda uma família de objetos.

Problema

Um sistema precisa lidar com formas geométricas (polígonos). Existe a necessidade de classes para **Triangulo**, **Retangulo** e **Pentagono**, cada uma com um número de lados específico.

Há, para esse sistema, uma interface **Poligono**:

```
public interface Poligono {
  public int getNumeroDeLados();
}
```

Depois, podem ser vistas as classes concretas que são implementações de **Poligono**:

```
public class Triangulo implements Poligono{
  @Override
  public int getNumeroDeLados() {
      return 3;
  }
}
```

```
public class Retangulo implements Poligono{
  @Override
  public int getNumeroDeLados() {
      return 4;
  }
}
```

```
public class Pentagono implements Poligono{
  @Override
  public int getNumeroDeLados() {
      return 5;
  }
}
```

Solução

Em vez de instanciar essas classes diretamente em vários pontos do sistema, você pode usar um método específico que, informado o número de lados, instancie o objeto adequado.

```
public class FabricaPoligono {

  public static Poligono getPoligono(int numeroDeLados) {
      switch(numeroDeLados)
      {
          case 3:
              return new Triangulo();

          case 4:
              return new Retangulo();

          case 5:
              return new Pentagono();

          default:
              return null;
      }
  }
}
```

Um exemplo de uso está no trecho a seguir:

```
Poligono forma = FabricaPoligono.getPoligono(3); //criará um triangulo
```

Padrões estruturais

Padrão *composite*

É aplicado a objetos que são formados pela composição de objetos similares. Geralmente é utilizado para montar estruturas hierárquicas (como menus com níveis e subníveis) ou estruturas de árvore. O *composite* deixa os clientes tratarem objetos individuais e composições de objetos do mesmo modo.

Problema

É necessário criar um menu para um sistema em desenvolvimento. Então, é criada uma classe **MenuItem**, que precisa ser unida para formar uma estrutura mais ampla.

Solução

Em primeiro lugar, você deve criar uma classe abstrata que representa um componente do menu (seja um item do menu, seja um menu):

```
abstract class MenuComponente {
  private String nome;
  private String numero;

  public MenuComponente(String numero, String nome) {
      this.numero= numero;
      this.nome = nome;
  }
  public String toString() {
      if (nome != null) {
          return "\t" + numero+ " - " + nome;
      }
      return numero;
  }
  public abstract void print();
}
```

Veja o código com a classe concreta **MenuItem**, derivada de **MenuComponente**, que serve para representar um item do menu:

```
class MenuItem extends MenuComponente {
  public MenuItem(String descricao, String nome) {
      super(descricao, nome);
  }
  public void print() {
      System.out.println(super.toString());
  }
}
```

Finalizada a estrutura de classes, parte-se agora para a aplicação do método em si. Veja a seguir a classe **Menu**, que realmente implementará a composição de um menu. Isso se dará essencialmente pelo uso de uma lista de itens, do tipo **List<MenuComponente>**, ou seja, a classe **Menu** representa uma categoria que, abaixo dela, contará com subitens de menu.

```
class Menu extends MenuComponente {
  private List<MenuComponente> componentes;

  public Menu(String descricao) {
      super(descricao, null);
      componentes = new ArrayList<MenuComponente>();
  }
  public void add(MenuComponente componente) {
      componentes.add(componente);
  }
  public void print() {
      System.out.println(">> "  + super.toString());
      for (MenuComponente c: componentes) {
          c.print();
      }
  }
}
```

Para finalizar, a classe principal **Comp**, que mostra um exemplo real de uso pelo cliente, usará a classe **Menu** e realizará a composição desta com subitens:

```
  public class Comp {
    public static void main(String[] args) {
        new Comp().montarMenu();
    }
    public void montarMenu() {
        Menu parte = new Menu("Games"); //comando que cria o objeto Menu, c
om o componente do tipo lista.

        parte.add(new MenuItem("1", "Ação"));
        parte.add(new MenuItem("2", "Esportes"));

        Menu parte2 = new Menu("Filmes");
        parte2.add(new MenuItem("1", "Comédia"));
        parte2.add(new MenuItem("2", "Drama"));
        parte2.add(new MenuItem("3", "Ficção Científica"));

        Menu parte3 = new Menu("Séries");
        parte3.add(new MenuItem("1", "The Walking Dead"));
        parte3.add(new MenuItem("2", "La casa de papel"));
        parte3.add(new MenuItem("3", "Wikings"));

        Menu principal = new Menu("Menu");
        principal.add(parte);
        principal.add(parte2);
        principal.add(parte3);
        principal.print();
    }
  }
```

Note que, com um único método **print()**, é possível mostrar na tela uma estrutura completa de menus e submenus:

```
run:
>> Menu
>> Games
    1 - Ação
    2 - Esportes
>> Filmes
    1 - Comédia
    2 - Drama
    3 - Ficção Científica
>> Séries
    1 - The Walking Dead
    2 - La casa de papel
    3 - Wikings
BUILD SUCCESSFUL (total time: 0 seconds)
```

Utilize o padrão *composite* para montar um questionário de múltipla escolha com as questões e as alternativas de resposta.

Padrão *adapter*

Permite que objetos com interfaces que não têm compatibilidade colaborem entre si, mediante o uso desse padrão, que cria um objeto especial que converte a interface de um objeto para que o outro consiga entendê-lo.

Problema

Há uma classe importante para o projeto, mas a interface dela é incompatível com o restante do código.

Como exemplo, imagine um sistema de tocador de arquivos MP3. O objetivo é expandi-lo para suportar também arquivos de vídeo MP4 e arquivos de *playlist* VLC. O projeto conta com duas interfaces: uma interface **MediaPlayer** (destinada a mp3) e uma nova interface **MediaPlayerAvancado** (para outros tipos de arquivo):

```
public interface MediaPlayer {
  public void play(String tipoAudio, String arquivo);
}

public interface MediaPlayerAvancado {
  public void playVlc(String arquivo);
  public void playMp4(String arquivo);
}
```

A princípio, você poderia considerar a classe **AudioPlayer** como a seguinte:

```
public class AudioPlayer implements MediaPlayer {

  @Override
  public void play(String tipo, String arquivo) {
        System.out.println("Tocando mp3. Arquivo: " + arquivo);
  }
}
```

O problema agora está em como agregar **MediaPlayerAvancado** mantendo o método **play()** de **AudioPlayer** como o único necessário para a execução do arquivo de mídia.

Solução

Crie um adaptador que permita que essa classe se comunique com as classes com que ela precisa trocar informações.

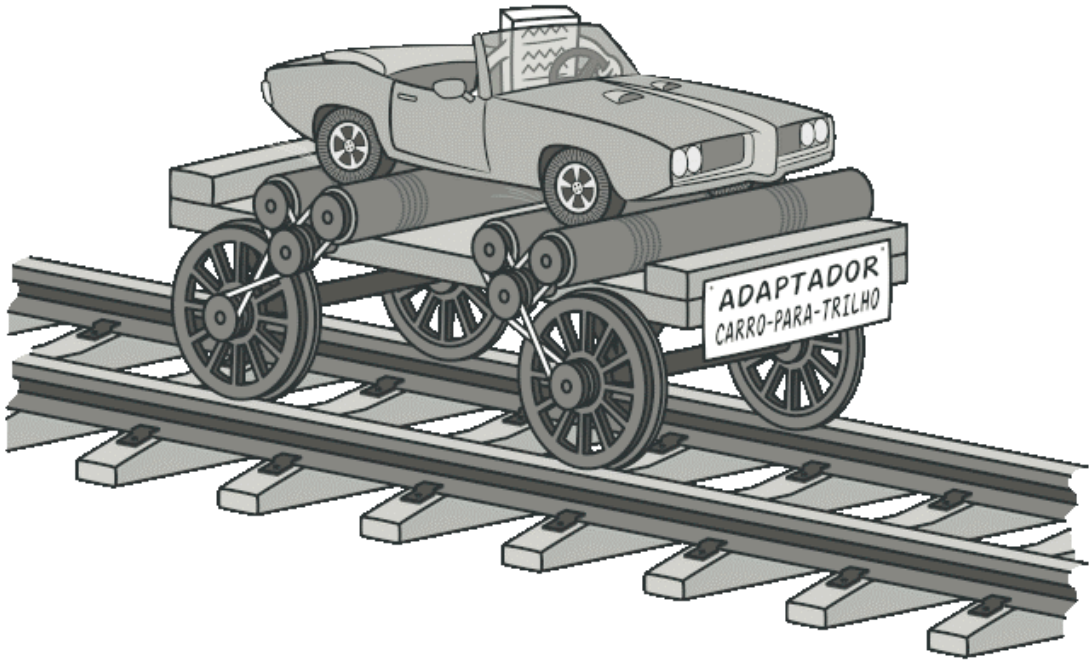


Figura 1 – Carro comum adaptado para andar em trilhos
Fonte: Refactoring Guru (c2014-2022)

Para este exemplo, será expandido o código criando classes concretas para **MediaPlayerAvancado** e criando um adaptador para ele. Primeiramente, serão criadas as duas classes concretas para o tocador de VLC e para o de MP4, ambas implementando **MediaPlayerAvancado**:

```
public class VlcPlayer implements MediaPlayerAvancado{
  @Override
  public void playVlc(String arquivo) {
      System.out.println("Tocando vlc. Arquivo: "+ arquivo);
  }

  @Override
  public void playMp4(String arquivo) {
      //não faz nada
  }
}

public class Mp4Player implements MediaPlayerAvancado{

  @Override
  public void playVlc(String arquivo) {
      //não faz nada
  }

  @Override
  public void playMp4(String arquivo) {
      System.out.println("Tocando mp4. Arquivo: "+ arquivo);
  }
}
```

Depois, será criada a classe **Adapter** que implementará a interface original – **MediaPlayer** – e que ajustará para que funcione com objetos de **MediaPlayerAvancado**:

```
public class MediaAdapter implements MediaPlayer {

  MediaPlayerAvancado playerAvancado;

  public MediaAdapter(String tipo){

      if(tipo.equalsIgnoreCase("vlc") ){
        playerAvancado = new VlcPlayer();
      }else if (tipo.equalsIgnoreCase("mp4")){
        playerAvancado = new Mp4Player();
      }
  }

  @Override
  public void play(String tipo, String arquivo) {

      if(tipo.equalsIgnoreCase("vlc")){
        playerAvancado.playVlc(arquivo);
      }
      else if(tipo.equalsIgnoreCase("mp4")){
        playerAvancado.playMp4(arquivo);
      }
  }
}
```

Note que, a partir de **MediaAdapter**, será possível usar a mesma assinatura do **MediaPlayer** comum, destinado a MP3, mas com arquivos de MP4 e VLC. É ajustada em seguida a classe **AudioPlayer**:

```
  public class AudioPlayer implements MediaPlayer {
    MediaAdapter mediaAdapter;

    @Override
    public void play(String tipo, String arquivo) {

        //suporte original a mp3
        if(tipo.equalsIgnoreCase("mp3")){
          System.out.println("Tocando mp3. Arquivo:" + arquivo);
        }
        //graças ao MediaAdapter, podemos também tocar vlc, mp4 (e futurame
nte outros tipos)
        else if(tipo.equalsIgnoreCase("vlc") || tipo.equalsIgnoreCase("mp
4")){
          mediaAdapter = new MediaAdapter(tipo);
          mediaAdapter.play(tipo, arquivo);
        }
        else{
          System.out.println("Formato não suportado");
        }
    }
  }
```

Agora, sim, você pode testar tudo a partir da classe principal:

```
  public class AdapterTeste {
    public static void main(String[] args) {
        AudioPlayer audioPlayer = new AudioPlayer();

        audioPlayer.play("mp3", "musica1.mp3");
        audioPlayer.play("mp4", "titanic.mp4");
        audioPlayer.play("vlc", "best-videos.vlc");
        audioPlayer.play("avi", "teste.avi");
    }
  }
```

O resultado será algo como isto:

```
Tocando mp3. Arquivo: musica1.mp3
Tocando mp4. Arquivo: titanic.mp4
```

```
Tocando vlc. Arquivo: best-videos.vlc
Formato não suportado
```

Considere a interface **Pato** e a interface **Galinha** e duas implementações para elas:

```
public interface Pato {
  public void quack();
  public void voa();
}

public class PatoReal implements Pato{

  @Override
  public void quack() {
      System.out.println("Quack quack...");
  }

  @Override
  public void voa() {
      System.out.println("Pato voando alto");
  }
}
```

```
public interface Galinha {
  public void cacareja();
  public void voa();
}

public class GalinhaCaipira implements Galinha {

  @Override
  public void cacareja() {
      System.out.println("Pó pó pó pó...");
  }

  @Override
  public void voa() {
      System.out.println("Galinha voando baixo");
  }
}
```

> Implemente um adaptador de **Galinha** para que assuma os comportamentos de **Pato**.

Padrões comportamentais

Padrão *state*

É um padrão de projeto comportamental que permite que um objeto altere o comportamento quando seu estado interno for alterado. O padrão *state* trabalha com a delegação da lógica referente a diferentes estados de um objeto.

Problema

Uma ação pode atingir dois estados: ligado ou desligado em uma televisão, e você precisa alterá-los sem alterar a classe principal do programa, e sim as classes dos estados específicos – neste caso, ligado ou desligado.

```
public interface Funcionamento {
  void clicarBotao();
}
```

É criada uma classe para implementar o primeiro estado do objeto, que é o estado “ligado”:

```
public class Ligada implements Funcionamento {

  @Override
  public void clicarBotao() {

      System.out.println("luz ligada");
  }
}
```

É criada, em seguida, uma classe para implementar o segundo estado do objeto, que é o estado “desligado”:

```
public class Desligada implements Funcionamento {

  @Override
  public void clicarBotao() {
      System.out.println("Luz desligada");
  }

}
```

A partir daí, finaliza-se a estrutura de classes e parte-se para a aplicação do método em si:

```
public class TV {
  private Funcionamento tv;

  public TV(Funcionamento tv) {
      super();
      this.tv = tv;
  }

  public Funcionamento getTv() {
      return tv;
  }

  public void setTv(Funcionamento tv) {
      this.tv = tv;
  }


  public void clicarBotao() {
      tv.clicarBotao();
  }
}
```

Para finalizar, é importante testar na classe principal, para que você veja um exemplo real de onde podem ser mudados os estados do objeto de “ligado” para “desligado”, e vice-versa:

```
  public static void main(String[] args) {
    Ligada on = new Ligada();
    TV t1 = new TV(on);
    t1.clicarBotao();
    System.out.println("");
    Desligada off = new Desligada();
    TV t2 = new TV(off);
    t2.clicarBotao();
  }

}
```

Logo após rodar o programa, o resultado encontrado é este:

```
  run:
  luz ligada

  Luz desligada
  BUILD SUCCESSFUL (total time: 0 seconds)
```

Padrão *observer*

É um padrão de projeto comportamental que permite definir um mecanismo de assinatura para notificar múltiplos objetos sobre quaisquer eventos que aconteçam com o objeto que eles estão observando. É semelhante a uma assinatura de jornal ou revista, porém, nesse padrão, a editora que publica é chamada de **Subject**, ou **Sujeito**; e os assinantes, de **Observer**, ou **Observador**.

Problema

O padrão *observer* resolve problemas que envolvem a atualização de vários objetos com base na modificação de uma informação específica. Por exemplo, imagine quando os clientes de uma loja estão procurando determinado produto. Seria ruim que os clientes fossem à loja todos os dias para questionar se o produto está disponível. O mais adequado é fazer com que a loja mantenha uma lista de clientes interessados e, assim que o produto chegar, notifique-os. É assim que o *observer* funciona.

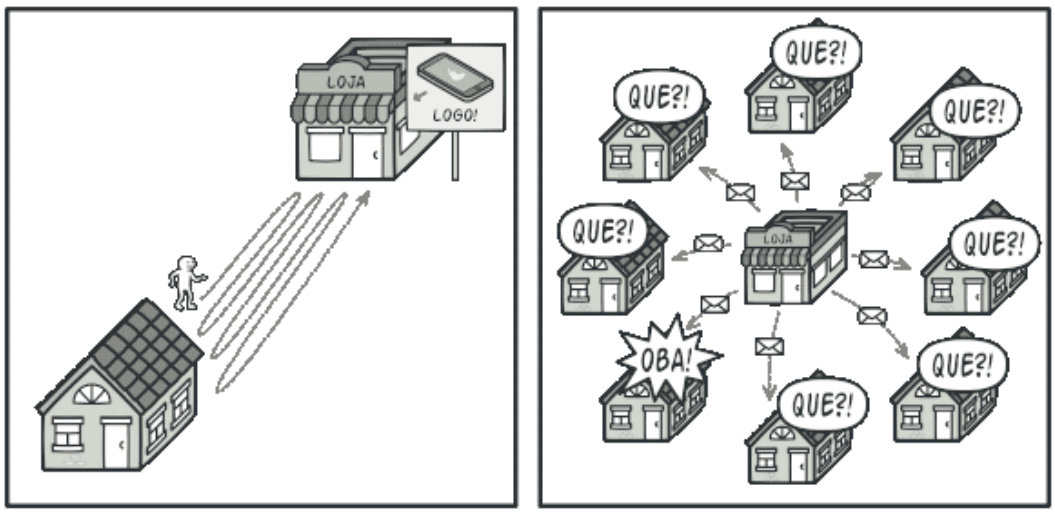


Figura 2 – Representação de problemas do não uso do *observer*
Fonte: Refactoring Guru (c2014-2022)

Para exercitar mais, pense em um sistema que precisa lidar com condições de clima e mostrar, em diferentes telas ou notificações, dados sobre a temperatura, a umidade e a pressão. Esse sistema teria:

- Um *display* para mostrar a temperatura, a umidade e a pressão atuais (dados que precisam ser atualizados constantemente)
- Uma notificação para quando a temperatura alterar
- Um *display* com estatísticas da temperatura do dia (temperatura mínima até agora; temperatura máxima até agora)

Como manter tudo isso atualizado sem equívocos?

Solução

Com o padrão *observer*, você tornará a classe responsável por controlar os dados brutos de temperatura um **Sujeito** (ou **Subject**), que manterá uma lista de interessados em seus dados e passará as informações a todos esses interessados, chamados de **Observadores** (ou **Observers**), assim que forem atualizadas. Cada um dos três elementos descritos (*display* atualizado, notificação e *display* de estatísticas) receberá essa atualização e agirá da maneira que julgar conveniente.

Então, antes de tudo, definem-se as interfaces principais:

```
public interface Subject {
  public void registraObservador(Observer o);
  public void removeObservador(Observer o);
  public void notificaObservadores();
}
```

Subject é a interface que será implementada pela classe que fornece a informação. É a partir dessa interface que os **observadores** serão notificados.

```
public interface Observer {
  public void atualizar(float temperatura, float umidade, float pressao);
}
```

Observer é uma interface que precisa ser implementada por todas as classes que querem acompanhar uma informação. Aqui se define o método **atualizar()**, responsável por receber a nova informação (neste caso, dados de clima) e atualizar o seu estado, realizando ações próprias a partir disso.

De modo geral, essas são as interfaces que estarão presentes em toda implementação do padrão *observer*.

A interface a seguir tem relação específica com a aplicação deste conteúdo, pois define o comportamento dos *displays* (recursos que mostrarão na tela as informações de clima):

```
public interface Display {
  public void mostrarDados();
}
```

Parte-se então às implementações concretas, sendo a do sujeito a primeira:

```
  public class DadosClima implements Subject{
    private List<Observer> observadores;

    private float temperatura;
    private float umidade;
    private float pressao;

    public DadosClima(){
        observadores = new ArrayList<Observer>();
    }

    public void alteraDados(float temperatura, float umidade, float pressa
o){
        this.temperatura = temperatura;
        this.umidade = umidade;
        this.pressao = pressao;
        notificaObservadores();
    }

    //adiciona o Observer recebido por parâmetro na lista "observadores"
    @Override
    public void registraObservador(Observer o) {
        observadores.add(o);
    }

    //retira um Observer na lista de "observadores"
    @Override
    public void removeObservador(Observer o) {
        observadores.remove(o);
    }

    //notifica cada observador para que eles se atualizem
    @Override
    public void notificaObservadores() {
        for(Observer o : observadores){
            o.atualizar(temperatura, umidade, pressao);
        }
    }
  }
```

A classe **DadosClima** gerencia as informações climáticas e, quando ocorre alteração (método **alteraDados()**), informa a todos os objetos da atualização (chamando **notificaObservadores()**).

Agora será implementado cada um dos observadores:

```
  public class DisplayAtualizado implements Observer, Display{
    private float temperatura;
    private float umidade;
    private float pressao;

    @Override
    public void atualizar(float temperatura, float umidade, float pressao)
{
        this.temperatura = temperatura;
        this.umidade = umidade;
        this.pressao = pressao;
        mostrarDados(); //imediatamente após atualizar informações, mostra
na tela
    }

    @Override
    public void mostrarDados() {
        System.out.println("Dados Atualizados: " + temperatura + "º,"
                + " umidade " + umidade
                +"%, pressão " + pressao);
    }
  }
```

DisplayAtualizado mantém uma cópia dos dados de clima e os mostra na tela sempre que são atualizados:

```
  public class DisplayEstatistica implements Observer, Display{
    public float tempMinima = Float.MAX_VALUE;
    public float tempMaxima = Float.MIN_VALUE;

    @Override
    public void atualizar(float temperatura, float umidade, float pressao)
{
        //se a temperatura atual é menor que a mínima até agora, troca a mí
nima
        if(tempMinima > temperatura)
            tempMinima = temperatura;
        //se a temperatura atual é maior que a mínima até agora, troca a má
xima
        if(tempMaxima < temperatura)
            tempMaxima = temperatura;

        mostrarDados();
    }

    @Override
    public void mostrarDados() {
        System.out.println("Minima do dia: " + tempMinima + "º; Máxima do d
ia: " + tempMaxima + "º");
    }

  }
```

DisplayEstatistica mantém os dados da mínima e da máxima temperaturas registradas no dia até então, mostrando-os sempre que são atualizados:

```
  public class DisplayNotificacao implements Observer, Display {
    public float temperaturaAtual = 0;

    @Override
    public void atualizar(float temperatura, float umidade, float pressao)
{
        if(temperaturaAtual != temperatura){
            temperaturaAtual = temperatura;
            mostrarDados();
        }
    }

    @Override
    public void mostrarDados() {
        System.out.println("Atenção! A temperatura mudou para " + temperatu
raAtual + "º");
    }
  }
```

DisplayNotificacao mostra uma mensagem de alerta sempre que a temperatura atual muda. No código principal, já é possível colocar o sistema para funcionar:

```
public class ObserverTest {

  public static void main(String[] args) {
      DadosClima dados = new DadosClima();

      DisplayAtualizado displayAtualizado = new DisplayAtualizado();
      dados.registraObservador(displayAtualizado);

      DisplayEstatistica displayEstatistica = new DisplayEstatistica();
      dados.registraObservador(displayEstatistica);

      DisplayNotificacao displayNotificacao = new DisplayNotificacao();
      dados.registraObservador(displayNotificacao);

      dados.alteraDados(10, 50, 30);
      dados.alteraDados(15, 40, 29);
      dados.alteraDados(17, 30, 29);
      dados.alteraDados(17, 40, 32);
  }

}
```

Foi criada uma instância para cada objeto de *display*, assim como foi registrado cada um deles como observadores do objeto “dados” da classe **DadosClima**. Depois os dados foram alterados várias vezes. O resultado da execução deve se assemelhar ao seguinte:

```
Dados Atualizados: 10.0º, umidade 50.0%, pressão 30.0
Mínima do dia: 10.0º; Máxima do dia: 10.0º
Atenção! A temperatura mudou para 10.0º

Dados Atualizados: 15.0º, umidade 40.0%, pressão 29.0
Mínima do dia: 10.0º; Máxima do dia: 15.0º
Atenção! A temperatura mudou para 15.0º

Dados Atualizados: 17.0º, umidade 30.0%, pressão 29.0
Mínima do dia: 10.0º; Máxima do dia: 17.0º
Atenção! A temperatura mudou para 17.0º

Dados Atualizados: 17.0º, umidade 40.0%, pressão 32.0
Mínima do dia: 10.0º; Máxima do dia: 17.0º
```

Aplicabilidade dos padrões de projeto

Algumas considerações são importantes antes de conhecer a aplicabilidade dos padrões de projeto, e, entre elas, está o conhecimento de que nem sempre os padrões devem ser utilizados nas aplicações. Eles devem ser usados para resolver problemas específicos, e não simplesmente para dizer que a programação está seguindo determinado padrão que deu certo em outro projeto.

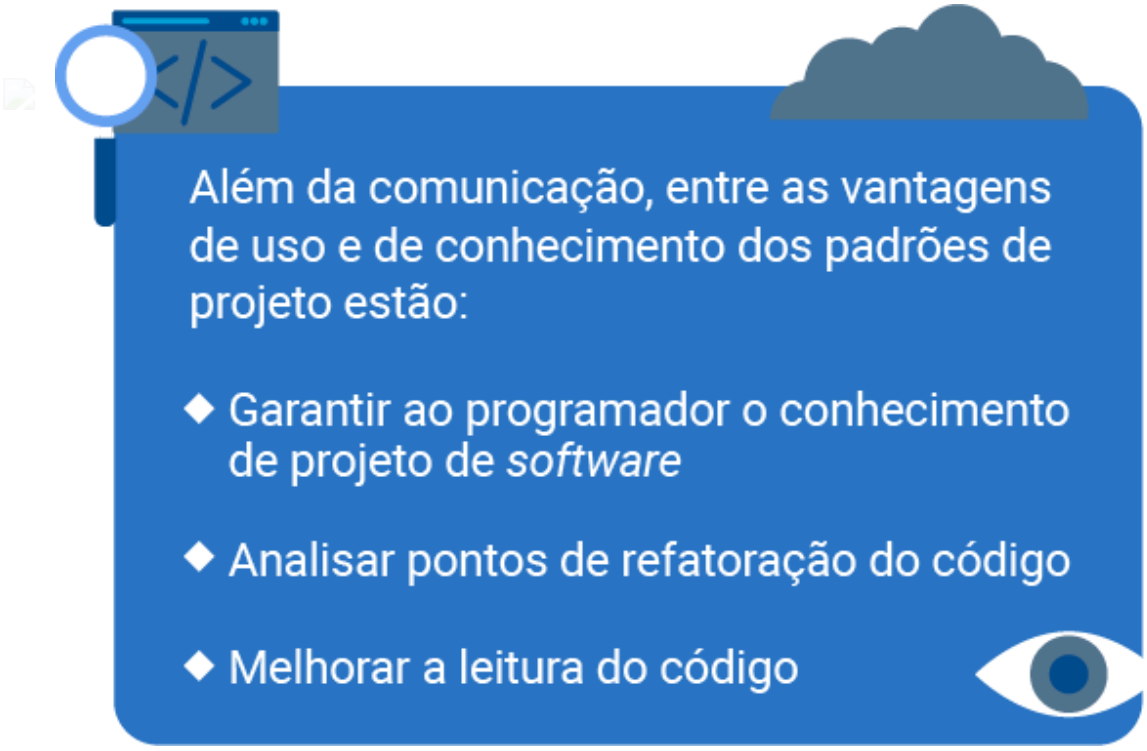
O fato de conhecer os principais padrões de projeto e saber que eles já resolveram problemas semelhantes aos que podem ser encontrados na aplicação em questão é o que torna um projeto aplicável. Isso dá uma garantia de que será utilizada uma solução que já foi testada e validada diversas vezes, e não simplesmente uma solução pensada naquele momento pelo programador.

Os exemplos de padrões explicados anteriormente trazem a aplicabilidade para os problemas específicos que precisam ser solucionados. Assim, para problemas semelhantes, pode-se analisar o uso do mesmo padrão.

Tendências de mercado

Existe certa discussão sobre a utilidade dos padrões de projeto hoje em dia, pois são um conceito definido há muito tempo. Eles são, sim, fundamentais, até pelo fato de terem se tornado fundamentos importantes na programação. O programador que conhece os padrões certamente está à frente no mercado, como um profissional de qualidade. Inclusive, as empresas seguem solicitando essa experiência dos candidatos.

Um dos pontos mais importantes é que os padrões facilitam a comunicação. Você pode sugerir o uso do *adapter*, por exemplo, para determinada situação, e a pessoa, caso conheça esse padrão, já terá uma ideia de uma solução composta para o problema que estiver enfrentando.



Não há exatamente uma tendência de padrões, mas, à medida que as aplicações e a programação se desenvolvem, novos padrões são propostos ou alguns clássicos voltam à tona.

A sugestão é sempre acompanhar fóruns e *sites* especializados em programação para ter ideia de quais são as situações comuns que se apresentam e de quais são as soluções sugeridas, fazendo ligações com possíveis problemas que você mesmo já enfrentou em algum momento.

Encerramento

Os padrões de projeto são um conhecimento que você deve ter, pois com certeza serão muito úteis no seu caminho dentro da carreira de programador. Lembre-se de que eles podem ser utilizados em todas as linguagens com suporte à programação orientada a objetos, e não somente na linguagem Java.

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.

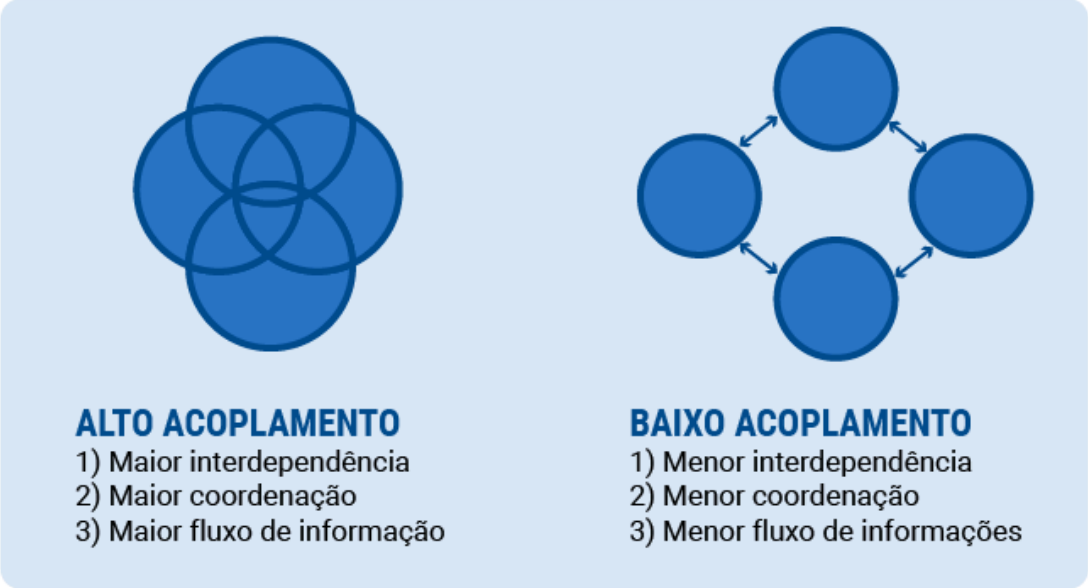


Figura 1 – Representação do alto grau de acoplamento *versus* baixo grau de acoplamento
Fonte: Adaptado de GeeksforGeeks (2020)

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:

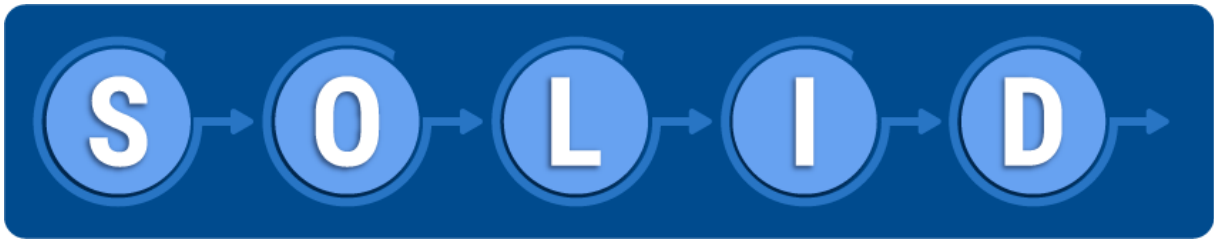
```
public class EnviadorEmail {

  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.



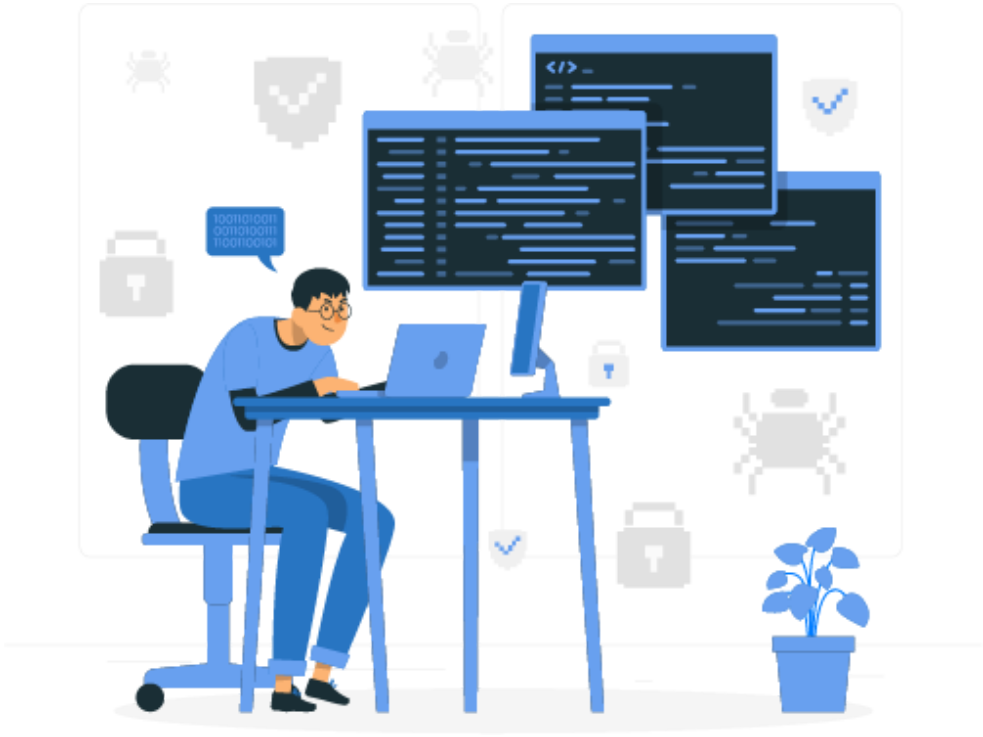
Aplicação dos princípios SOLID

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.

Desenvolvimento de Sistemas

Processos de *software*: importância e aplicabilidade, metodologias ágeis, rotina de desenvolvimento com metodologia ágil

Continuando os estudos sobre a utilização das metodologias ágeis, você pode imaginar agora uma produção de *notebooks*. Muitos processos estão envolvidos. Há toda a parte plástica que tem que ser injetada, bem como a estrutura metálica que suporta todos os componentes.

Inclusive, sobre os componentes, existem vários que são fabricados por outros fabricantes que não são necessariamente os fabricantes do próprio *notebook*. A placa-mãe tem muitos componentes de vários fabricantes diferentes.

Finalmente, toda a integração é feita, o sistema operacional é instalado e o produto final deve funcionar perfeitamente, sem nenhum tipo de falha. Porém, nessa integração há milhares de componentes sujeitos a diversas falhas.

Figura 1 – Linha de produção de *notebooks*
Fonte: QNC (2019)

Comparando com *notebooks*, o desenvolvimento de *software* também tem vários processos, incluindo a integração final, ou seja, a aplicação pronta para o usuário final. Obviamente, tais processos são muito mais intelectuais do que físicos quando se trata de *notebooks*. Mesmo assim, todo o processo envolve várias tarefas que vão sendo realizadas por várias pessoas ao mesmo tempo, dependendo da complexidade do projeto.

Hoje ainda se ouve falar sobre grandes programadores que desenvolveram grandes projetos sozinhos ou pelo menos começaram projetos sozinhos. Um grande exemplo é o finlandês Linus Torvalds, que iniciou, sozinho, como um projeto para uso pessoal, o desenvolvimento do famoso Linux. Após divulgar suas intenções em 1991, hoje existe um projeto em pleno funcionamento com milhares de desenvolvedores em todo o mundo trabalhando em conjunto.

A maioria das empresas, nos dias de hoje, trabalha com equipes, o que demonstra a importância dos processos.

Como já visto anteriormente, as metodologias tradicionais como a em cascata (*waterfall*) eram baseadas em processos de engenharia que derivaram de áreas de projeto mecânica, aeronáutica, civil, enfim, áreas nas quais os

projetos eram sempre planejados inicialmente em detalhes, eram sequenciais e muito bem documentados.

Logo os coordenadores começaram a perceber que o desenvolvimento de *software* era diferente e não poderia seguir os mesmos métodos. As aplicações sofrem várias mudanças ao longo do desenvolvimento e, muitas vezes, o dono do projeto não tem uma noção exata do que precisa, o que é bem diferente de um projeto de ponte ou avião, por exemplo.

Então, em 2001, um grupo de profissionais criou um novo método de controle de processos de *software* chamado de “manifesto ágil”. O principal ponto, para este conteúdo, é o uso de **ciclos curtos e iterativos de desenvolvimento**. Dessa maneira, o sistema vai sendo desenvolvido gradativamente. O que é mais urgente tem mais prioridade, o que o cliente mais precisa é o que será trabalhado primeiro hoje.

Assim que o módulo de *software* for validado e aprovado pelo cliente, inicia-se um novo ciclo, ou uma nova **iteração**, de desenvolvimento com a próxima prioridade. Cada iteração costuma durar de duas a quatro semanas em média. Dessa forma, o sistema completo vai sendo criado de modo incremental e termina quando o cliente estiver satisfeito com todos os requisitos, que devem estar em pleno funcionamento.

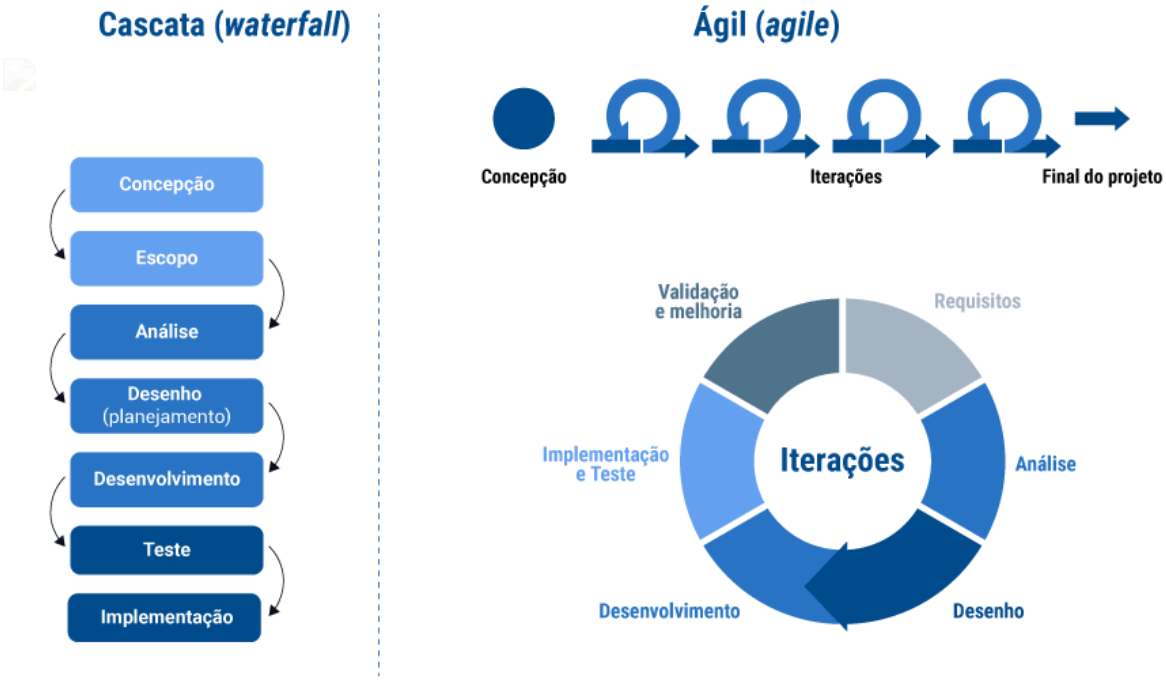


Figura 2 – Comparação de metodologias: cascata x ágil
Fonte: Adaptado de InovaLab (2018)

As metodologias ágeis usam o modelo de iterações que preza a aprendizagem e a melhoria contínua. Tanto o cliente quanto o time de projeto não sabem exatamente o que querem até que tudo esteja concluído e aprovado. O projeto como um todo é dividido em pequenas partes que já podem ser utilizadas pelo cliente logo após serem concluídas. Essas partes costumam ser chamadas de “mínimo produto viável” (MVP).

Com essa técnica, vários problemas são minimizados, pois as iterações curtas, também chamadas de “*sprints*” na metodologia Scrum, promovem um ambiente de produtividade constante. No método Scrum, a equipe planeja apenas as tarefas necessárias para aquela entrega específica (MVP). Esses objetivos de curto prazo têm, assim, menos chances de atrasos e erros. Reuniões diárias e curtas tornam o processo mais transparente para todos, pois as conversas tratam do que foi feito ontem e do que será feito hoje.

Outro destaque é a documentação do projeto. A máxima é a de que **o tempo que a documentação levou para ser gerada não pode custar mais que o seu próprio valor para o projeto**.

Todas essas novas técnicas diminuíram consideravelmente os problemas dos métodos tradicionais. Você conhece o projeto do balanço na árvore que foi desenvolvido usando o método em cascata? Veja:

Figura 3 – Projeto do balanço na árvore
Fonte: Adaptado de ClipaTec Informática (2011)

É possível listar alguns problemas tradicionais nos métodos antigos:

- O próprio cliente pode não saber exatamente o que quer.
- As primeiras reuniões com o resumo do produto são vagas ou incompletas.
- O planejamento não corresponde à realidade.
- A implantação não tem consistência.
- Não existe alinhamento com as expectativas do cliente.
- Não existe uma visão do produto final.

Todos os problemas citados podem ser consideravelmente minimizados com a aplicação das iterações rápidas proporcionadas pelos métodos ágeis. Pequenos erros não impedem grandes acertos. Projetos falham, e, nas metodologias tradicionais, tais falhas só serão descobertas depois de muito tempo e a um custo mais elevado. Com o uso de *sprints*, desde o início já é possível avaliar se o projeto é viável para o time e o cliente.

Metodologias ágeis

As metodologias ágeis mais presentes no mercado hoje são estas: XP (*extreme programming*), Scrum e *kanban*. Veja a seguir mais detalhes sobre a rotina de cada uma delas.

Complemente seus estudos sobre o tema relendo o conhecimento **Desenvolvimento ágil: cultura ágil, metodologias, rotinas, *frameworks***, desta unidade curricular.

Rotina de desenvolvimento com metodologia ágil

Afinal, como funciona o método Scrum na prática? Como seria a rotina diária de uma fábrica de *software*, por exemplo? Claro que as várias empresas de desenvolvimento de *software* têm seus próprios métodos, os quais são adaptados às suas necessidades.

Confira a seguir um exemplo ilustrativo de um sistema de biblioteca:

Projeto de *software*

O projeto de *software* compreende as definições do projeto, ou seja, os requisitos do sistema:

- Telas da aplicação (*front-end*)
- Operações e banco de dados (*back-end*)
- Requisitos funcionais e não funcionais

Papéis

É necessário definir os papéis de todos os envolvidos no projeto:

- *Product owner* (dono do produto): representante da biblioteca.
- *Scrum master*: especialista e facilitador.
- Time de desenvolvimento: dois desenvolvedores *front-end* e dois desenvolvedores *back-end*.

Primeira reunião de planejamento (*sprint planning*)

Todos se reúnem pela primeira vez a fim de planejar o desenvolvimento do sistema, ou seja, os artefatos do Scrum. Após a definição do planejamento geral, os eventos (do Scrum) são determinados.

- O *scrum master* guia a reunião, que deve durar no máximo oito horas.
- O *product owner* apresenta o *product backlog* ou a lista das histórias com as prioridades.
- A equipe define o *sprint backlog* com a lista das histórias para a primeira *sprint*:
  ◦ Tela de cadastro de livros e usuários
  ◦ Banco de dados com as tabelas de livros e usuários
- As histórias então são divididas em tarefas:
  ◦ Tela inicial da aplicação (nome, logo, informações etc.) com os *links* (botões) para as outras telas (duas por enquanto)
  ◦ Tela de cadastro de livros com todos os campos necessários
  ◦ Tela de cadastro de usuários com todos os campos necessários
  ◦ Banco de dados com as tabelas completas de livros e usuários
  ◦ Verificação das inserções corretas no banco (se as informações digitadas nas duas telas alimentarão o banco corretamente)
  ◦ Verificação de erros e testes finais
- A equipe vota (*planning poker*) no tempo necessário para a primeira *sprint* e preenche o quadro do Scrum com as tarefas prioritárias ou com o *backlog* da *sprint*. Para facilitar, confira um exemplo de quadro inspirado no *kanban*. Esse quadro normalmente fica fixado na parede e deve estar visível para todo o time:

Sprint backlog	A fazer	Fazendo	Teste		Pronto
			Fazendo	Feito	
Tarefa 1					
Tarefa 2					
Tarefa 3					

Quadro 1 – Modelo de quadro inspirado no *kanban*

Fonte: Senac EAD (2022)

Reuniões diárias (*daily scrum*) durante a *sprint*

As reuniões diárias servem para que todos os membros do time se comuniquem e se atualizem de suas situações nas tarefas.

- Todos devem participar.
- A reunião deve ser breve (15 minutos em média).
- Cada membro do time responderá a estas três perguntas:
  - O que eu fiz ontem?
  - O que eu farei hoje?
  - Algum problema ou algum impedimento para realizar a tarefa?

Figura 5 – Reunião diária (*daily scrum*)
Fonte: Método Ágil (s.d.)

Revisão da *sprint* (*sprint review*)

As tarefas estão prontas, e o time de desenvolvedores fará a demonstração para o *product owner* e para os demais convidados. Se alguma tarefa precisar de ajustes, ela volta a ser comentada na próxima *sprint*. O

product owner deve aprovar as tarefas para que o processo continue com novas tarefas.

- Todos devem participar.
- O time de desenvolvimento demonstra as tarefas feitas.
- O *product owner* e algum convidado utilizador devem validar as tarefas.
- A apresentação deve durar no máximo quatro horas e ocorre normalmente na sexta-feira.

Retrospectiva da *sprint* (*sprint retrospective*)

A retrospectiva é o último evento de uma *sprint*. Depois do encerramento, inicia-se uma nova *sprint*, com novas tarefas ou com tarefas anteriores que necessitem de ajustes.

- Somente o time *scrum* participa.
- Identificam-se melhorias, dificuldades, sugestões e reclamações.
- O evento dura no máximo três horas e ocorre normalmente na sexta-feira.

Após a retrospectiva, uma nova iteração ou uma nova *sprint* se repete com a reunião de planejamento a fim de definir as próximas tarefas, sempre de acordo com as prioridades apontadas pelo *product owner*. Normalmente, essas reuniões ocorrem nas segundas-feiras para o reinício das *sprints*. No novo ciclo, novas tarefas serão executadas. Se houver algum problema ou alguma parte que o time não concluiu, a(s) tarefa(s) em questão retorna(m) para um retrabalho.

Encerramento

Você viu neste conhecimento a importância da aplicação de metodologias ágeis nos processos de desenvolvimento de *software* e um exemplo prático de como elas funcionariam na rotina de uma empresa.

Vive-se em um mundo em constante evolução e transformação. Para sobreviver e prosperar, é preciso então estar aberto à adaptação e ao entendimento das novas tecnologias. Os modelos de negócios também estão cada vez mais rompendo todas as barreiras dos métodos tradicionais de gestão mais estáticos e não adaptativos. Assim, a adoção de métodos ágeis está mais alinhada à nova realidade atual e mais adequada aos desafios do futuro.

Desenvolvimento de Sistemas

Desenvolvimento ágil: cultura ágil, metodologias, rotinas, *frameworks*

O que é desenvolvimento ágil?

Você aprendeu, logo no início de seus estudos, as metodologias ágeis Scrum e *kanban*, que são frequentemente utilizadas. Esses conjuntos de orientações e ferramentas que organizam um modo de trabalhar (*frameworks*) são formas dinâmicas de desenvolver um projeto, seja um projeto de *software*, seja um projeto pessoal.

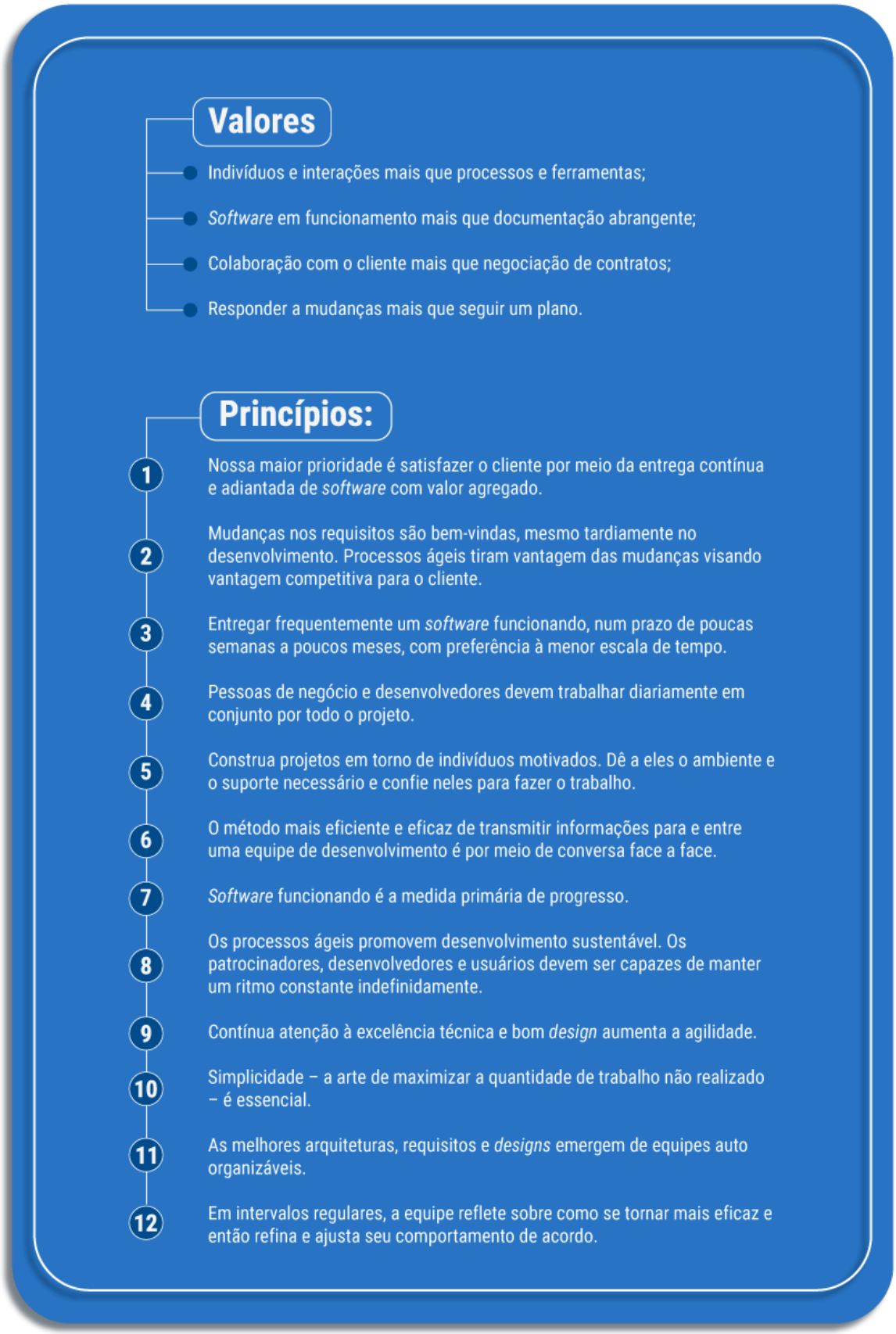
Ainda que seja muito útil conhecer vários tipos de processos e de ferramentas disponíveis, também é imprescindível conhecer a estrutura de princípios e valores que fundamenta tais metodologias, pois, para se tornar um desenvolvedor ágil, é necessário compreender a visão geral de agilidade. Pode-se dizer, portanto, que o desenvolvimento ágil se trata de um conjunto de fundamentos e processos que guia a execução das metodologias ágeis.

Vale também ressaltar que, embora seja possível aplicar metodologias ágeis a outros projetos que não sejam de desenvolvimento de *software*, o contexto deste conteúdo abrange a perspectiva do desenvolvimento de *software*.

Em 2001, um grupo de desenvolvedores se reuniu para criar uma declaração contendo uma representação de qual seria a melhor forma de desenvolver um *software*. Naquele período, as metodologias utilizadas não estavam sendo satisfatórias, portanto, filtrar e agrupar aspectos que promovessem eficiência e eficácia nos projetos era uma forma de produzir novos e melhores resultados.

Assim, com base nessa análise, foram definidos quatro valores e 12 princípios, os quais posteriormente fundamentariam as metodologias ágeis. O conjunto desses princípios e desses valores foi chamado de “manifesto ágil”.

Veja os valores e os princípios do manifesto ágil que guiam as práticas ágeis:


Valores
Indivíduos e interações mais que processos e ferramentas;
Software em funcionamento mais que documentação abrangente;
Colaboração com o cliente mais que negociação de contratos;
Responder a mudanças mais que seguir um plano.
Princípios:
1 Nossa maior prioridade é satisfazer o cliente por meio da entrega contínua e adiantada de *software* com valor agregado.
2 Mudanças nos requisitos são bem-vindas, mesmo tardiamente no desenvolvimento. Processos ágeis tiram vantagem das mudanças visando vantagem competitiva para o cliente.
3 Entregar frequentemente um *software* funcionando, num prazo de poucas semanas a poucos meses, com preferência à menor escala de tempo.
4 Pessoas de negócio e desenvolvedores devem trabalhar diariamente em conjunto por todo o projeto.
5 Construa projetos em torno de indivíduos motivados. Dê a eles o ambiente e o suporte necessário e confie neles para fazer o trabalho.
6 O método mais eficiente e eficaz de transmitir informações para e entre uma equipe de desenvolvimento é por meio de conversa face a face.
7 *Software* funcionando é a medida primária de progresso.
8 Os processos ágeis promovem desenvolvimento sustentável. Os patrocinadores, desenvolvedores e usuários devem ser capazes de manter um ritmo constante indefinidamente.
9 Contínua atenção à excelência técnica e bom *design* aumenta a agilidade.
10 Simplicidade – a arte de maximizar a quantidade de trabalho não realizado – é essencial.
11 As melhores arquiteturas, requisitos e *designs* emergem de equipes auto organizáveis.
12 Em intervalos regulares, a equipe reflete sobre como se tornar mais eficaz e então refina e ajusta seu comportamento de acordo.

Além de conhecer os fundamentos do desenvolvimento ágil, é muito importante entender o real significado de **agilidade** dentro do contexto de desenvolvimento de *software*.

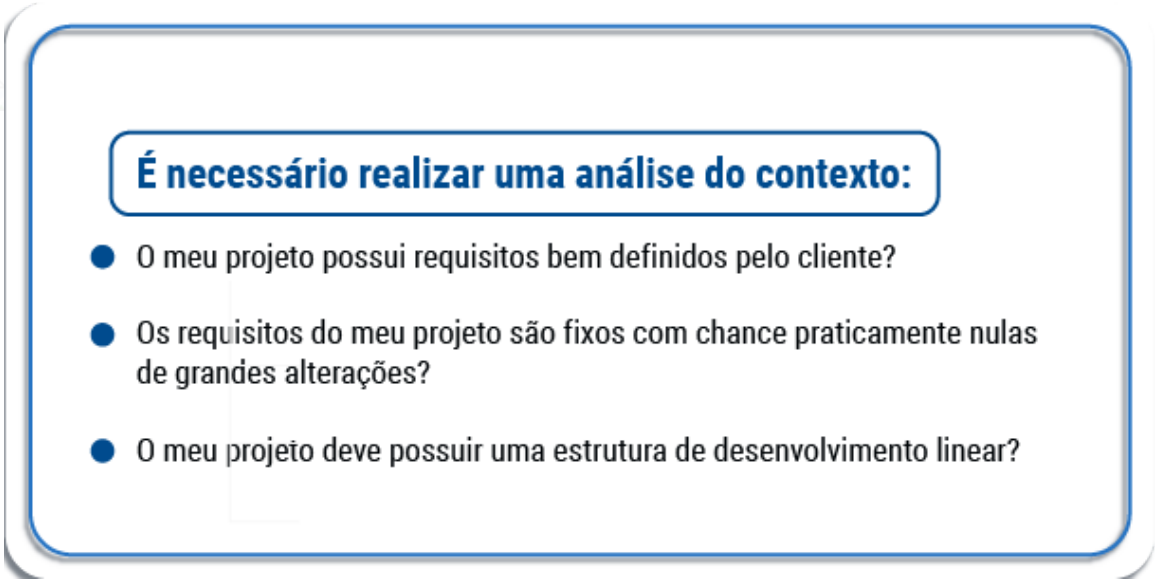
Em um primeiro contato com o termo, é comum relacionar a ideia de desenvolvimento ágil com a rapidez na entrega do *software*, com um desenvolvimento muito mais rápido em relação a outras metodologias. Entretanto, essa visão limitada pode acabar frustrando empresas, desenvolvedores e clientes, que passarão a ter uma expectativa irreal.

Ainda que possa de fato acelerar o processo de desenvolvimento, ao analisar o manifesto ágil, é possível observar que muitos dos princípios e dos valores estão relacionados à entrega contínua do projeto funcional em prazos menores, à aceitação de mudanças de projeto e à adaptação rápida a elas, à realização de análises frequentes de como o desenvolvimento está ocorrendo, à simplificação de processos burocráticos, entre outras.

Todos esses aspectos fazem com que as dificuldades e os problemas que possam surgir sejam solucionados mais rapidamente. É um trabalho em equipe que requer práticas simplificadas dentro da rotina. Portanto, não se trata somente de uma possível rapidez de entrega, mas, sim, da adoção de hábitos ágeis, da facilidade de processos, entre outros.

“Entrega contínua” é a capacidade de o sistema ser constantemente atualizado para o cliente, incluindo novas funcionalidades ou correções de defeitos.

Ainda, embora as metodologias ágeis possam chamar muita atenção como algo dinâmico e moderno, elas não serão obrigatoriamente a melhor solução para o desenvolvimento de um projeto.



Embora não seja regra, se você respondeu “sim” para as perguntas, provavelmente a metodologia em cascata ou outras convencionais se encaixem melhor no desenvolvimento do seu projeto.

Sistemas que envolvam licitações ou serviços específicos para órgãos públicos se beneficiam dessas metodologias, pois os requisitos do sistema provêm de regras previamente definidas e bem estruturadas. Outros sistemas mais simples, como gerenciamento de estoque, gerenciamento de vendas em supermercado e sistema de elaboração de orçamentos, são exemplos de projetos compostos de procedimentos padronizados com baixa tendência de mudanças ou atualizações, nos quais também não há uma real necessidade de aplicar metodologias ágeis.

Lembre-se de que, no método em cascata, cada etapa (de requisitos, de projeto, de implementação, de testes e de implantação) só pode iniciar quando a anterior estiver concluída. Outras metodologias, como a em espiral, que prevê períodos cíclicos de desenvolvimento (como as *sprints* do método Scrum), não são ágeis pelo excesso de burocracia e de documentação envolvidas.

Existem casos em que o tempo de colocação de um produto no mercado é um requisito importante, visto que, muitas vezes, determinado projeto depende do momento de entrada no mercado para prosperar e reagir contra a

competitividade da concorrência.

Se o prazo de entrega passar desse período, talvez o projeto de *software* perca o sentido e o valor para o cliente. Diante dessa perspectiva, para que o produto se destaque no mercado, muitas vezes os requisitos precisam ser alterados para que não haja prejuízos para o cliente.

Em outros casos, os requisitos podem nem estar completamente definidos logo no início do projeto, seguindo o fluxo de necessidades, e é aí que surge o seguinte questionamento: como criar um processo capaz de administrar a imprevisibilidade?

Os métodos ágeis, então, entram em ação e se apresentam como uma vantagem competitiva. É necessário ter agilidade para se adaptar a mudanças e apresentar entregas de valor para o cliente e para o ambiente de negócios.

Sistemas de redes sociais, de lojas, de tele-entrega, de transporte e outros que necessitam frequentemente de novas funcionalidades e atualizações são exemplos de aplicações que, muitas vezes, são influenciadas até mesmo por questões econômicas, sociais e sazonais. Esses são apenas alguns exemplos de sistemas que seriam beneficiados pelo uso de metodologias ágeis, nos quais cada entrega particionada acrescenta valor para o cliente.

Como foi possível observar, dois dos principais aspectos da metodologia ágil são a dinamicidade e a fluidez ao entrar em cenários de mudança. Em contextos normais, mudanças são caras, principalmente se não forem controladas e se não forem bem administradas, existindo processos adequados para lidar com a situação. Uma das características mais atrativas da metodologia ágil é justamente a capacidade de reduzir os custos da mudança no processo de *software*.

Observe a seguir um gráfico comparativo entre a agilidade e o custo das mudanças em relação a processos convencionais:

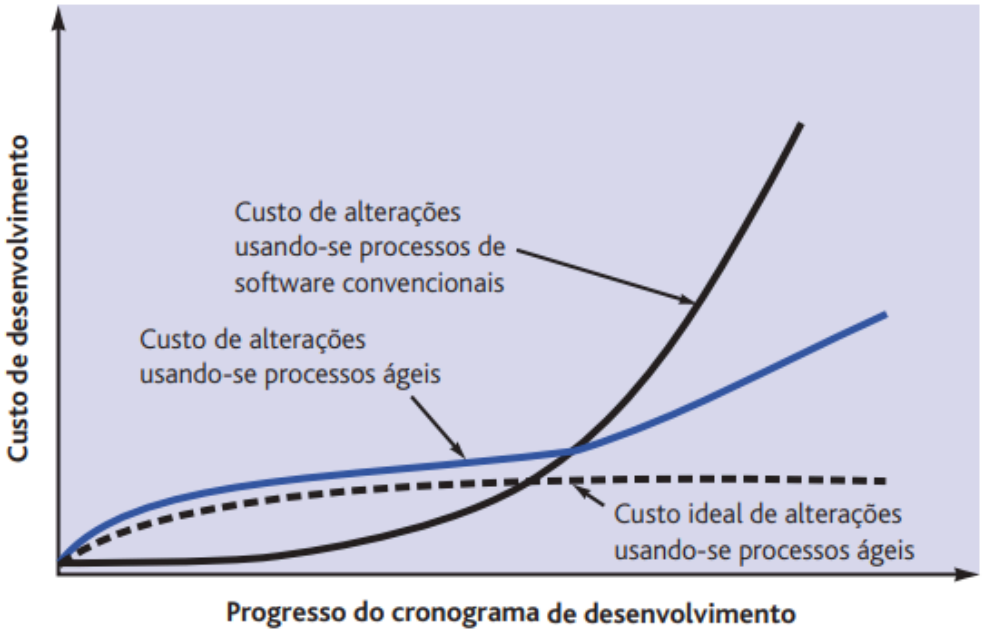


Figura 1 – Custos de alterações em relação ao tempo de desenvolvimento
Fonte: Pressman; Maxim (2016)

No processo de reunir requisitos logo no início do projeto, se houver a necessidade de alterar um detalhamento ou aumentar uma lista de funções, o projeto não será afetado negativamente, pois ainda será possível adaptar os processos.

Quanto mais perto do fim do projeto, mais difícil se torna para as metodologias mais “rígidas” alterarem sua estrutura, fazendo com que o processo de modificação seja mais custoso. Isso porque, por vezes, as mudanças podem surgir quando o sistema já está em etapa de testes e acaba envolvendo alterações em diversas áreas em que o desenvolvimento já foi concluído.

As metodologias ágeis tendem a reduzir esse problema, pois as alterações ao longo do desenvolvimento se dividem em tarefas menores conforme surgem. Além disso, o impacto das modificações reduz o custo geral desse processo.

As entregas incrementais aos clientes se tornam bastante benéficas por essa perspectiva, pois permitem que o desenvolvimento seja avaliado e que possíveis mudanças sejam identificadas o mais breve possível, podendo moldar também a estrutura de como o projeto deve seguir posteriormente, evitando retrabalhos futuros.

Rotinas

É disseminado exageradamente que as práticas ágeis são contra processos, documentações e todo e qualquer processo burocrático, mas não é assim que funciona.

Desenvolvimento ágil não significa que nenhum documento é criado ou que processos não são utilizados; significa, na verdade, que somente os documentos que serão consultados mais adiante no processo de desenvolvimento serão criados, priorizando reais necessidades, da mesma forma que algum processo obrigatoriamente burocrático de uma empresa, se realmente necessário, poderá continuar sendo burocrático.

Outro aspecto muito importante é a percepção de que desenvolver um *software* de forma ágil não se trata somente de seguir uma metodologia ágil.

Não é possível simplesmente definir que serão utilizadas metodologias ágeis sem antes fazer com que práticas ágeis já façam parte da rotina. Quando a rotina é adaptada à visão de agilidade, uma estrutura adequada é desenvolvida para implementar metodologias, mesmo que outras áreas permaneçam mais burocráticas ou requeiram alguns procedimentos.

Veja a seguir algumas práticas que podem ser incorporadas no dia a dia para simplificar processos e proporcionar melhores resultados para a equipe:

Programação em par

Uma das práticas que podem trazer benefícios para a rotina dos programadores é a programação em par. A ideia é que duas pessoas trabalhem no desenvolvimento de um mesmo código, compartilhando monitor, teclado e *mouse*.

Um dos componentes da dupla (o “líder”) ficará responsável pela parte escrita do código, enquanto a outra pessoa (o “navegador”) terá a responsabilidade de observar, analisar e revisar o que está sendo desenvolvido. O papel de cada um da dupla poderá ser modificado com certa frequência caso necessário.

Ainda que pareça um trabalho rigoroso, ele traz diversos benefícios a longo prazo, pois melhora a qualidade do código com base na troca de experiências, conhecimentos e informações entre as pessoas, diminuindo a necessidade de uma futura refatoração do código.

Outro benefício proporcionado pelo uso dessa prática é o foco de desenvolvimento da equipe, pois todos estarão muito envolvidos nos códigos e em frequente observação.

É necessário, antes de aplicar a programação em par, analisar a quantidade de colaboradores disponível para realizar esse processo, bem como os prazos estipulados para o projeto.

Embora possa trazer benefícios a longo prazo (como redução de tempo em refatoração de código e maior compreensão do código inteiro), para situações que requeiram mais rapidez, talvez a

programação em par não seja a prática mais adequada.

Em substituição, pode ser usada a revisão de código (*code review*), uma evolução da programação em pares que define que todo código desenvolvido deve ser, sim, revisado, mas se possível em outro momento, de maneira assíncrona, por outro desenvolvedor.

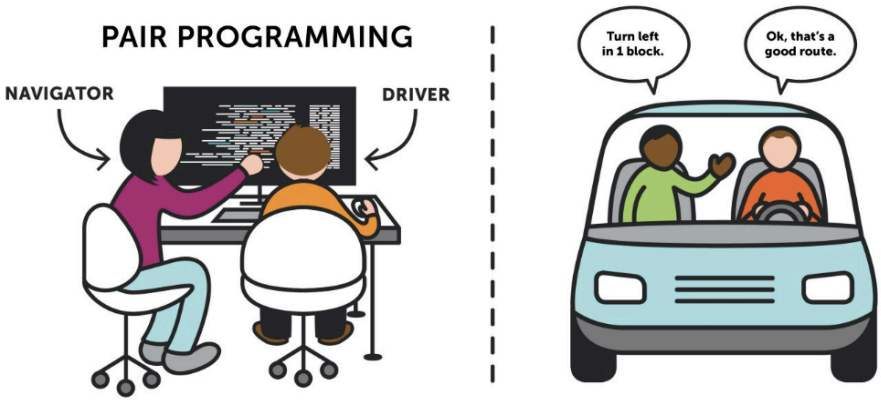


Figura 2 – Programação em par
Fonte: Mass (2020)

Veja mais alguns benefícios da programação em par:

- Muitos erros são identificados ao longo do desenvolvimento, e não nas etapas finais do projeto, como em estágios de teste.
- A identificação de erros no fim do projeto é estatisticamente menor.
- O código é menor e tem um *design* melhor devido às contribuições do observador.
- A equipe resolve problemas mais rapidamente.
- A equipe compreende melhor os problemas encontrados e as práticas de construção de *software*.
- O projeto de *software* acaba com várias pessoas conhecendo melhor cada peça da solução que foi desenvolvida, facilitando possíveis ajustes ou incrementos no código.
- A equipe desenvolve as habilidades de trabalho em equipe, comunicando-se com mais fluidez e dinamicidade.

Apesar de benéfica, essa prática realmente pode se tornar cansativa se realizada com frequência. Dessa forma, é importante realizar pausas ao longo do dia.

É importante também revezar duplas, para que toda a equipe fique mais engajada no desenvolvimento e melhore a comunicação.

Code review

Uma prática importante que deve ser implementada em equipes é a colaboração no código. Técnicas como a revisão de código (*code review*) auxiliam a equipe no desenvolvimento do senso de responsabilidade coletiva pelo que está sendo criado e, ao mesmo tempo, aumentam a base de conhecimento de todos os envolvidos.

Tal como o nome sugere, o objetivo dessa prática é realizar uma revisão a cada versionamento do projeto, não devendo ser um processo muito demorado, visando a encontrar falhas, *bugs* e possíveis pontos de melhoria no desenvolvimento.

O programador, assim que conclui uma funcionalidade e está pronto para incluí-la no código principal do projeto (e no repositório), submete esse arquivo a outro desenvolvedor, que analisa o código e aponta melhorias, caso necessário. O arquivo então retorna ao programador, que realiza os ajustes e o submete novamente à revisão até que não haja mais ajustes necessários.

O ideal é que, no ambiente da equipe, haja tempo para analisar o código que está sendo produzido e dar um retorno sobre ele.

Há ainda outros pontos positivos que podem ser observados na code *review*:

- Aumento da qualidade do código
- Maior confiança no momento de entrega do projeto
- Ganho de conhecimento tanto por parte do desenvolvedor quanto por parte de quem está realizando a revisão

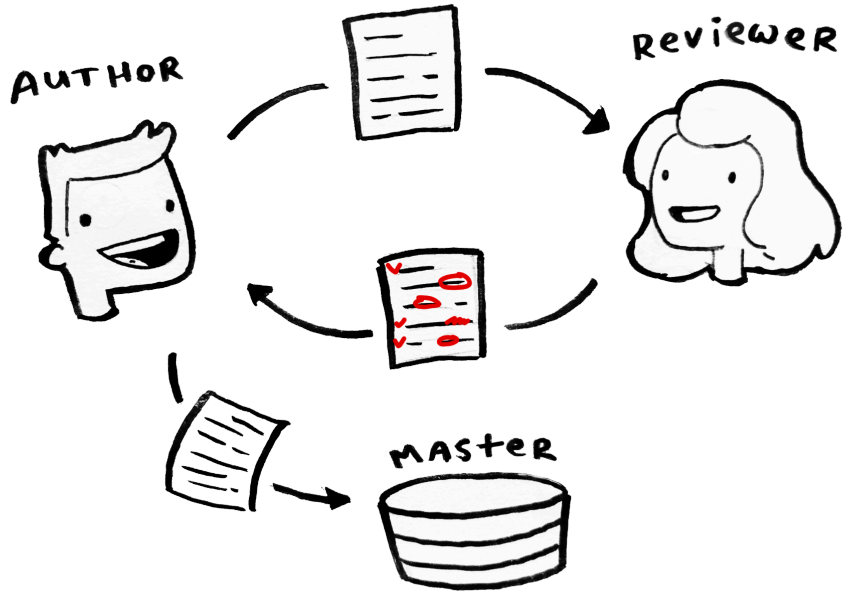


Figura 3 – Processo de *code review*

Fonte: Matic (2020)

TDD

Outra prática interessante a ser adotada é o *test-driven development* (desenvolvimento guiado por testes), ou TDD. Trata-se de uma prática que depende de testes automatizados, classes e métodos específicos que invocam métodos de outras classes do sistema, verificando se o resultado gerado por eles é o esperado.

Veja um exemplo de teste automatizado:

```
public class ClasseDoSistema{
    public int calculaSoma(int a, int b){
        int soma = a + b;
        return soma;
    }
}
```

```
public class TestesDoSistema{
  @Test
  public void testaSoma(){
      int resultado = calculaSoma(2,2);
      //verifica se o resultado é 4, o esperado para 2+2
      assertEquals(4,resultado);
  }
}
```

Essas classes de teste podem ser executadas automaticamente, permitindo detectar erros e alguma alteração realizada no código.

No TDD, primeiro são desenvolvidos os métodos de teste e, depois, os de funcionalidade. No exemplo anterior, primeiramente foi desenvolvido **testaSoma()** para depois ser implementado **calculaSoma()**. A primeira execução do teste acusará falha, pois não há implementação no método testado. Diz-se, nesse momento, que os testes estão “vermelhos”, ou seja, falhos.

Com as definições dos resultados esperados, completa-se o código da funcionalidade e se executa novamente o teste até que a execução fique “verde”, ou seja, seja realizada com sucesso (o resultado esperado é obtido). O objetivo a partir daí será desenvolver um código completo que resulte em sucesso nos testes anteriormente idealizados. Após, poderá ser realizada a refatoração do código para aprimorá-lo.

Dessa maneira, é possível observar todas as possíveis falhas do código, podendo resolver todos os problemas, e guiar o modo como o *software* é desenvolvido. Além disso, na hora de realizar testes de *software*, provavelmente o código necessitará de menos ajustes, pois a análise de falhas e testes já ocorreu durante o desenvolvimento.

Porém, cabe um ponto de atenção: **essa prática requer um período de adaptação para que seja aplicada.**

O TDD traz os seguintes benefícios:

- Mais clareza do código que já foi produzido
- Mais qualidade do produto, pois os testes automatizados darão segurança de que uma nova funcionalidade incluída no sistema não prejudicará outras funcionalidades já existentes
- Refinamento do código de forma mais rápida e objetiva às necessidades

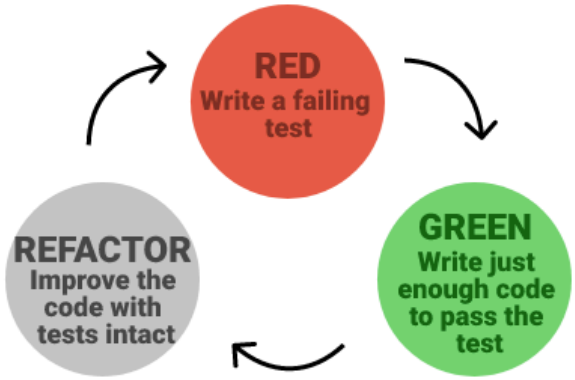


Figura 4 – Processo de TDD

Fonte: Stemmler (2021)

Design incremental

O *design* incremental da base de código consiste em uma técnica que visa a uma estrutura de código simples desde a sua concepção. O ponto-chave é que, embora simples, a estrutura pode ser melhorada de forma contínua, progressivamente.

A parte do *design*, no caso de *softwares* mais complexos, é muito importante para garantir robustez e qualidade futura, afinal, por eles terem um período mais longo de desenvolvimento e um conjunto maior de requisitos, é possível que mudanças sejam solicitadas com mais frequência conforme as necessidades do cliente.

O *design* incremental é uma forma de economizar tempo, direcionando o desenvolvimento diretamente para as demandas necessárias com mais frequência.

Alguns dos benefícios em utilizar tal técnica são:

- Um *design* que é frequentemente aprimorado desenvolve uma estrutura melhor para aqueles que trabalham no desenvolvimento dele.
- Diversas situações tornam difícil a tentativa de prever detalhes do *design*. Logo, não há como definir rapidamente um *design* final para a estrutura de código. O uso do *design* incremental faz com que não seja gasto mais tempo do que o necessário em algo que possivelmente será alterado.
- Em códigos antigos, reescrevê-los utilizando o *design* incremental aumenta a vida útil do *software*.

Considerando que as metodologias ágeis buscam um processo de entrega contínua, novas decisões na estrutura do *software* podem ser definidas após o projeto ser disponibilizado em sua primeira versão para os clientes.

Padronização do código

Mais uma prática bastante interessante é a padronização do código desenvolvido. São consideradas boas práticas de programação: realizar a documentação de código; definir estilos e padrões de *design*; indentar etc. Quando você desenvolve um código, por exemplo, precisa considerar que ele não passará somente pelas suas mãos. Portanto, é imprescindível que o código seja de fácil compreensão e mantenha sempre padrões de semântica.

Há ainda outros benefícios em utilizar a padronização de códigos:

- A leitura do código se torna mais fácil para todos os desenvolvedores e para si mesmo.
- A aplicação de melhorias ao código e a manutenção dele se tornam tarefas menos “pesadas”, fluindo com mais facilidade.
- A adoção da padronização do código torna os algoritmos mais profissionais.
- Reduz-se a necessidade de manutenção do código.

Time box

A *time box* (caixa de tempo) tem um conceito simples: é uma forma de gerir o tempo mediante a definição de uma meta e de um intervalo de tempo para a realização de uma tarefa. Isso evita a procrastinação, pois você já definiu o que vai fazer e quanto tempo terá disponível para tanto.

É importante, inclusive, que o tempo predefinido seja adequado à tarefa, afinal, caso venha a ser estabelecido mais tempo do que o necessário, o objetivo real dessa prática acabará não sendo atingido.

São alguns dos benefícios da utilização da *time box*:

- Maior foco ao desenvolver uma tarefa
- Otimização do tempo, evitando passar mais que o necessário em um mesmo código
- Aplicação em diversos contextos além da programação
- Conclusão mais frequente das metas, o que gera motivação

Comunicação

A comunicação com os clientes e com a própria equipe, dentro do mundo ágil, objetiva praticidade e objetividade.

É importante ser acessível e facilitar a troca de informações sempre que possível. Reuniões auxiliam bastante no processo de comunicação, mas devem seguir os padrões de agilidade, tendo um tempo predefinido (*time box*) e objetividade.

Por causa do cenário atual, o trabalho é, em diversas situações, realizado a distância. Então, é interessante investir também em ferramentas que auxiliem no processo de comunicação, tais como Slack, Teams, Discord, entre outras.

Mesmo que não existam muitas técnicas-padrão a serem seguidas, é importante sempre ter bom senso. As pessoas não são iguais; elas têm capacidades e personalidades diferentes. É necessário respeitar os colegas em todos os processos de comunicação para um bom ambiente de equipe.

Buscando a melhoria contínua, você pode observar o manifesto ágil, em especial o item “indivíduos e interações mais que processos e ferramentas”. Ao analisá-lo, é possível entender que, embora práticas ágeis sejam muito benéficas, é imprescindível melhorar a interação entre as pessoas para alcançar o sucesso no time e para ter uma *performance* melhor.

Para finalizar, lembre-se de que as práticas mencionadas, ainda que sejam utilizadas dentro de algumas metodologias ágeis, podem ser aplicadas individualmente.

Frameworks e metodologias

Como visto ao longo deste conteúdo, o desenvolvimento ágil consiste em um conjunto de princípios e valores utilizado como forma de nortear todo o processo de organização e de desenvolvimento de tarefas.

Quando se fala em metodologias e *frameworks* ágeis, está-se falando em formas de **aplicar** os princípios ágeis a um projeto, ou seja, formas de sair somente da teoria e vincular a agilidade a um projeto.

Pode-se dizer então que *frameworks* são um conjunto de ferramentas úteis que, quando aplicadas à rotina, trazem determinados resultados que inclusive podem ser aplicados a contextos diferentes.

Já uma metodologia é um conjunto de métodos (incluindo ferramentas) que fazem parte de um processo bem definido, orientando os passos que o projeto deve tomar. Ainda que os conceitos tenham suas diferenças, muitas vezes o termo “*framework*” é difundido como metodologia, ou então o termo “metodologia” é difundido como *framework*. Sendo assim, talvez você encontre em livros e na Internet os processos e as ferramentas que serão apresentados sendo definidos das duas formas.

Algumas das metodologias mais utilizadas no mercado você já deve conhecer. Que tal recapitular?

Scrum

O que é

Scrum é considerado um *framework* que auxilia pessoas, times e organizações a gerarem valor por meio de soluções adaptativas para problemas complexos.

Processo

Em resumo, o método Scrum requer que um *scrum master* promova um ambiente em que ocorram os seguintes processos:

- Um *product owner* ordena o trabalho para problemas complexos em um *product backlog*.
- O *scrum team* transforma uma seleção do trabalho em um incremento de valor durante uma *sprint*.
- O *scrum team* e os seus stakeholders inspecionam os resultados e se preparam para a próxima sprint por meio de reuniões frequentes.
- O processo é repetido até a finalização do projeto.

Para relembrar conceitos gerais do Scrum de forma mais detalhada, leia a obra *The Scrum Guide*, um guia oficial desenvolvido pelos criadores do Scrum, gratuito e disponível em português.

Kanban

O que é

O *kanban* é uma estratégia para otimizar o fluxo de valor por meio de um processo que utiliza facilitação visual e limitação de *work in progress* – WIP (trabalho em progresso) de um sistema puxado. Essa facilitação visual ocorre por quadros com colunas e cartões.

Processo

O processo deve ocorrer da seguinte forma:

- Visualização do *workflow*: clareza do fluxo de trabalho, por meio de quadros, para que todos possam enxergá-lo.
- Limitação do *work in progress*: definição – por estado(s) do trabalho, por pessoa, por faixa, por tipo de trabalho, para todo o sistema *kanban* etc. – do número máximo de itens de trabalho permitidos de cada vez.
- Gestão ativa dos itens de trabalho em andamento: monitoramento frequente da execução de processos.
- Inspeção e adaptação da definição de *workflow* do time: verificação da eficácia do fluxo de trabalho e realização de ajustes conforme necessário.

Para relembrar conceitos gerais do *kanban* de forma mais detalhada, leia a obra *The Official Guide to The Kanban Method*, um guia oficial desenvolvido pelos criadores do *kanban*, gratuito e disponível em português.

XP

O que é

A *extreme programming* (programação extrema), ou XP, envolve um conjunto de regras e práticas constantes no contexto de quatro atividades metodológicas: planejamento, projeto, codificação e testes. O propósito é que, se uma prática está sendo benéfica, ela deve ser executada várias e várias vezes, trazendo mais qualidade ao código.

Processo

O processo deve ocorrer da seguinte forma:

- Planejamento: história de usuário, valores, critérios de teste de aceitação, plano de iteração.
- Projeto: projeto simples, cartões CRC (classe, responsabilidade, colaboração), soluções pontuais, protótipos.
- Codificação: programação em pares, testes de unidades, integração contínua.
- Testes: testes de unidades, integração contínua.

A XP não conta com um guia especialmente desenvolvido para ela, mas, para conhecê-la ainda mais, acesse o *site* oficial da metodologia para acompanhar as regras de cada uma das etapas.

Apesar dos aspectos únicos de cada metodologia, é possível combiná-las conforme necessário. Com o avanço do tempo, as organizações precisam estar adaptadas a novas realidades, pois, consequentemente, manter o mesmo conjunto de práticas nem sempre é a melhor alternativa para o negócio.

Nessa perspectiva, é sempre importante buscar a metodologia (ou as metodologias) que se aplica melhor aos processos. Também é válido considerar como a equipe se adaptará às mudanças de papéis e à estrutura de trabalho. O processo poderá ser desafiador, mas com certeza trará benefícios.

Cultura ágil e processo de transição

A cultura dentro de uma empresa é caracterizada pelo conjunto de comportamentos, atitudes, missão, visão e valores que guiam uma organização. Assim, a cultura ágil consiste em inserir os conceitos de agilidade como parte do que guia a organização, intrínsecos aos colaboradores.

Para empresas muito focadas em procedimentos burocráticos e estruturas hierárquicas, a inovação parece algo distante. Muitas empresas – e não somente equipes de desenvolvimento – têm buscado, com mais frequência, formas de utilizar metodologias ágeis para melhorar seus processos de negócio. Quando as organizações adotam essa perspectiva, criam aberturas para transformar e atualizar o ambiente de trabalho, trazendo melhores resultados para os clientes externo e interno.

Mesmo com tantos benefícios, a falta de preparo e o não entendimento da cultura ágil na empresa podem fazer com que todo esse processo de transição se torne, no lugar de uma solução, um problema ainda maior. Alguns dos problemas que podem ser vistos quando a cultura ágil não está sendo realmente entendida na empresa são os seguintes:

- Crença dos gestores de que não existe mais liderança
- Insatisfação quando orientações mais tradicionais não ágeis são rejeitadas
- Dificuldade de se “desprender” de procedimentos burocráticos

A cultura ágil não busca necessariamente aplicar agilidade em todos os setores da empresa, quebrar todas as regras, mas sim, como visto anteriormente, trazer um equilíbrio entre ter agilidade nos locais certos e seguir normas que são realmente importantes. Adequar-se a esse domínio requer empenho de todos que fazem parte da empresa.

Veja a seguir como adotar essa visão no seu cotidiano e como outros setores influenciarão no processo:

Auto-organização

“Uma equipe ágil é auto-organizada e tem autonomia para planejar e tomar decisões técnicas” (PRESSMAN, 2016). Quando se fala em metodologias ágeis, pode-se ter a perspectiva de que a auto-organização é não linear, ou seja, de que não há um cronograma fixo do que deve ser feito.

Logo, por haver alterações frequentes e mais dinamicidade no projeto, é necessária uma organização maior para que as tarefas não se tornem uma enorme bagunça. Veja algumas dicas:

- Seja proativo para executar tarefas pendentes. Não espere ordens.
- A comunicação com a equipe deve ocorrer frequentemente, inteirando-se mais das necessidades dela.
- Questione várias vezes sobre suas tarefas até que compreenda completamente o que deve ser desenvolvido. Não fique com dúvidas.
- Registre todas as informações que achar necessárias para a tarefa, evitando perdê-las.
- Atualize frequentemente os *status* do seu progresso na tarefa para uma melhor organização da equipe.

A auto-organização é um processo de cada um, que não conta com uma autoridade ou com elementos externos para ser colocada em prática. Para que a cultura ágil funcione, é necessário que toda a equipe siga princípios básicos de auto-organização por conta própria.

Visão de time

“A filosofia ágil enfatiza a competência individual (de cada membro da equipe) combinada com a colaboração em grupo como fatores críticos de sucesso para a equipe” (PRESSMAN, 2016). Em um contexto de equipe, existe uma descentralização de autoridade, permitindo que haja uma maior participação dos integrantes no andamento do projeto. Porém, não se engane: ainda haverá um gestor para orientar o time.

A diferença, nesse caso, é que a equipe poderá opinar mais sobre como os procedimentos serão realizados e terá mais autonomia para tomar decisões técnicas e para organizar e direcionar as tarefas conforme a especialidade da equipe. A estrutura-base do projeto será responsabilidade da liderança, enquanto a dinamicidade referente à organização e à delegação de tarefas caberá à equipe.

Confira algumas ações que auxiliam na introdução da cultura ágil pela perspectiva de equipe:

- A organização do trabalho deve ocorrer em grupo para uma melhor delegação de tarefas.
- A comunicação entre a equipe deve ser frequente, auxiliando nas dificuldades e disseminando informações importantes sobre o projeto.
- A equipe deve ser capaz de sincronizar as atividades que serão realizadas naquele dia e ter uma visão geral do projeto.

Comunicação com outros setores

“Suporte inadequado pode fazer com que até mesmo os bons fracassem na realização de seus trabalhos” (PRESSMAN, 2016). Embora as metodologias ágeis sejam vistas com mais frequência na área da programação, muitas empresas estão ampliando o uso dessas práticas para diversos setores.

Em razão da necessidade de aumentar a eficiência dos processos, compreender melhor as necessidades dos clientes e atender às novas demandas que surgem em um ritmo rápido, vem se tornando frequente o “ágil em escala”. “Ágil em escala” é quando um conjunto de práticas ágeis visa a abranger um número maior de grupos –nesse caso, outros setores da empresa.

Esses outros setores da empresa, ainda que passem a ter mais autonomia, seguirão com uma liderança, tal como os times de desenvolvedores. A nova visão agora é que todos os setores se comuniquem de forma a facilitar o progresso das atividades da empresa, alinhando-se para um mesmo objetivo.

Seguem alguns aspectos que tornam outros setores ágeis:

- Outros setores também passarão a ter mais autonomia e descentralização de informações, da mesma forma que já ocorre com as equipes de desenvolvimento.
- Se os setores da empresa já estiverem adaptados aos objetivos do time, ficará muito mais fácil diminuir o tempo de execução de determinados processos, pois os setores terão informações-base para dar retorno com mais agilidade.
- Os setores devem estar engajados entre si para aumentar a produtividade. Dessa forma, as demandas ficam “travadas” por menos tempo.
- A comunicação deve ser direta e objetiva entre os setores, auxiliando na redução de ruídos de comunicação.

Grande parte da cultura ágil se refere a ignorar parte do controle e confiar que todos querem ter o melhor desempenho possível, cada um fazendo sua parte e participando das reuniões diárias, trabalhando como equipe. Para auxiliar nesse processo de adaptação, também é válido aplicar as rotinas apresentadas neste conteúdo.