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.

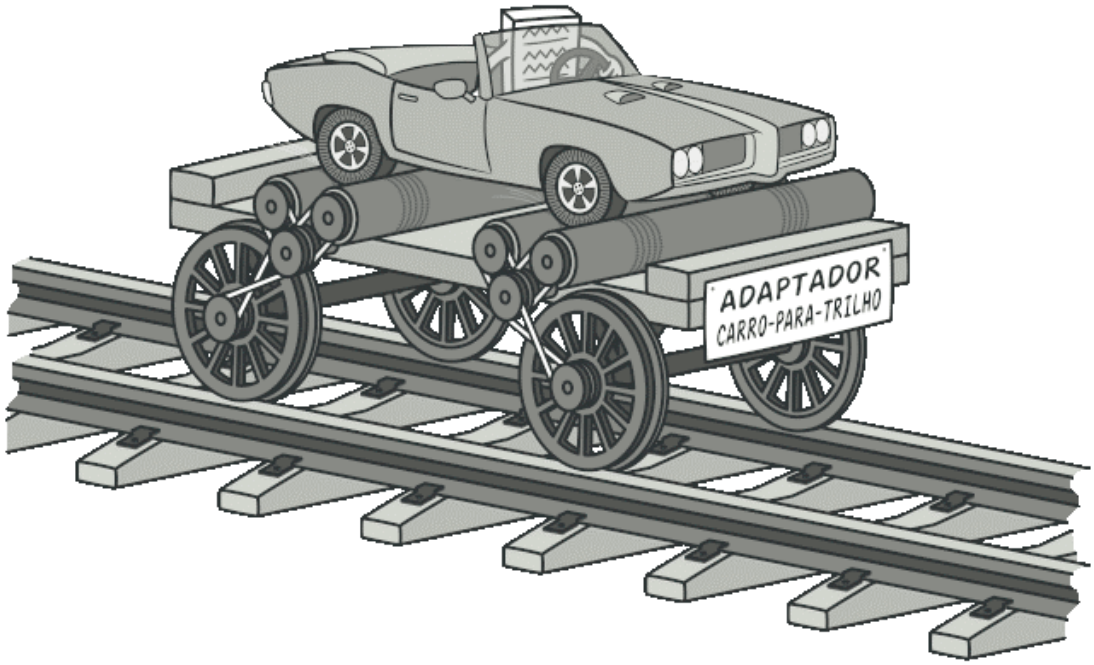


Figura 1 – Carro comum adaptado para andar em trilhos
Fonte: Refactoring Guru (c2014-2022)

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.

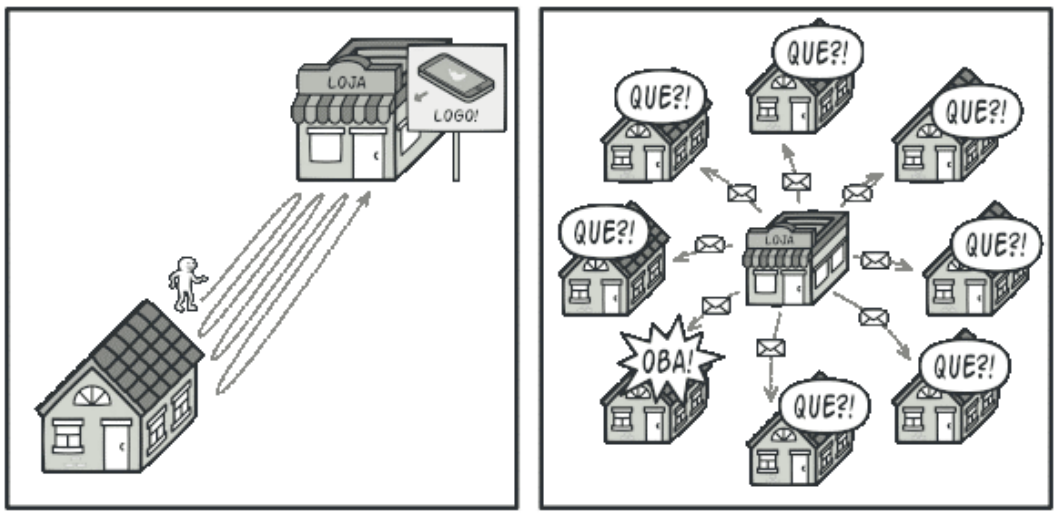


Figura 2 – Representação de problemas do não uso do *observer*
Fonte: Refactoring Guru (c2014-2022)

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.

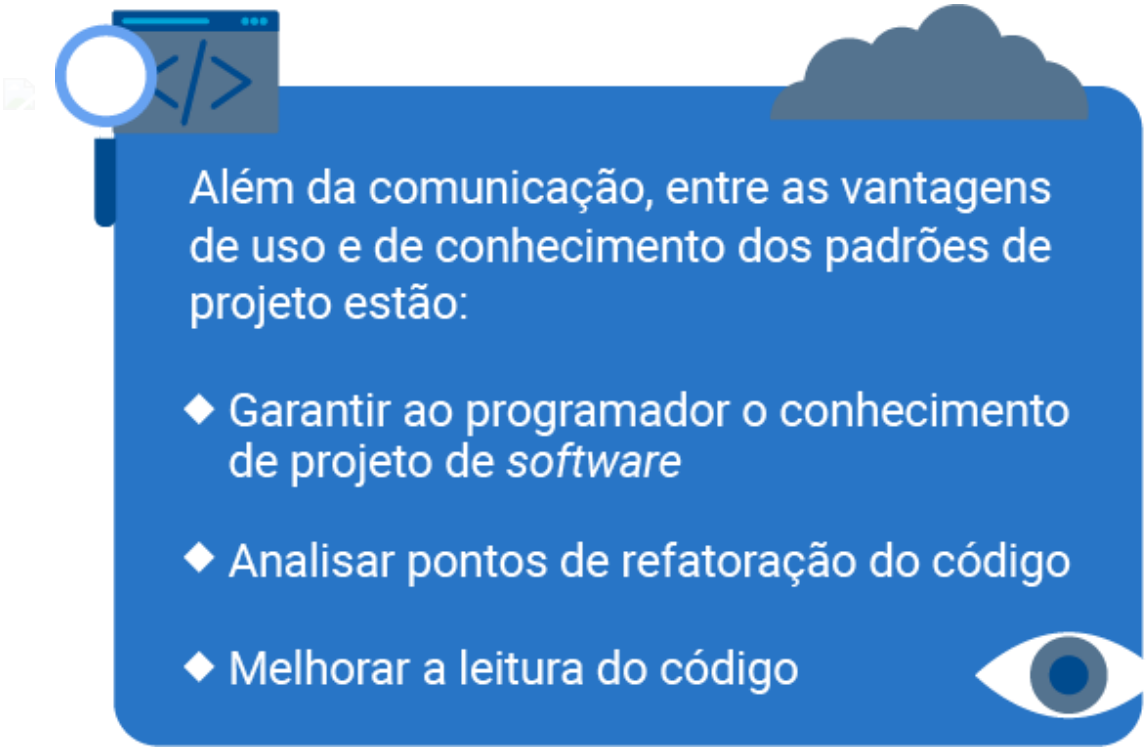
O fato de conhecer os principais padrões de projeto e saber que eles já resolveram problemas semelhantes aos que podem ser encontrados na aplicação em questão é o que torna um projeto aplicável. Isso dá uma garantia de que será utilizada uma solução que já foi testada e validada diversas vezes, e não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