ruep

Revista UNILUS Ensino e Pesquisa
v. 13, n. 30, jan./mar. 2016
ISSN 2318-2083 (eletrônico)

O USO DOS PADRÕES DE PROJETO GOF NA ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

Claudio Costa Matos

Graduando no curso de Tecnologia em Análise e Desenvolvimento de Sistemas do Centro Universitário Lusíada (UNILUS).

claudiocm_7@hotmail.com

Fernando Kauffmann Barbosa

Professor Mestre no curso de Tecnologia em Análise e Desenvolvimento de Sistemas do Centro Universitáio Lusíada (UNILUS) e membro do Núcleo Acadêmico em Estudos e Pesquisas em Estatística e Qualidade na Educação (CNPq).

professor@fernandokb.pro.br

Artigo recebido em março de 2016 e aprovado em abril de 2016.

RESUMO

Os Padrões de Projetos GoF ("Gang of Four") surgiu nos meados da década de 90 por quatro projetistas de software e que são aplicados, nos dias atuais, no desenvolvimento de sistemas dentro do paradigma orientado a objetos. São compostos por vinte e três padrões e divididos em três grupos: Padrões de Criação, Padrões Estruturais e Padrões Comportamentais. A utilização dos padrões de projeto visa a melhoria na qualidade do desenvolvimento de sistemas, permitindo o fácil entendimento de suas funcionalidades e diminuição e/ou soluções de problemas complexos através da reutilização de soluções já aprimoradas e testadas anteriormente.

Palavras-Chave: Padrões de Projeto. GoF. Engenharia de Software.

THE USE OF GOF DESIGN PATTERNS IN THE ANALYSIS AND DEVELOPMENT OF SYSTEMS

ABSTRACT

The GoF ("Gang of Four") design patterns emerged in the mid-90s by four software designers and they are applied nowadays in the development of systems within the paradigm of object-oriented. They are composed of twenty-three patterns and divided into three groups: Creational Patterns, Structural Patterns and Behavioral Patterns. The use of design patterns aimed at improving the quality of system development, allowing easy understanding of its functionality and decrease and/or complex problems solutions by reusing solutions have improved and tested previously.

Keywords: Design Patterns. GoF. Software Engineering.

Revista UNILUS Ensino e Pesquisa
Rua Dr. Armando de Salles Oliveira, 150
Boqueirão, Santos - São Paulo
11050-071

http://revista.lusiada.br/portal/index.php/ruep
revista.unilus@lusiada.br

Fone: +55 (13) 3202-4100

INTRODUÇÃO

Padrões de Projeto apresentam soluções a problemas que, dentro de um determinado contexto, surgem com maior frequência. Para o desenvolvedor de sistemas os padrões de projeto são mecanismos significantes, pois tais padrões já foram aprimorados, testados e utilizados com base nas experiências de outros desenvolvedores de sistemas.

A utilização dos padrões de projeto não garante a solução para todos os problemas encontrados em um desenvolvimento de sistemas. Deve-se ter o conhecimento do problema, generalizá-la e, consequentemente, aplicar o padrão adequado para solucionar o problema encontrado, contudo, pode existir problemas que os padrões não resolvam, mas podem simplificar a complexidade desses problemas.

A importância da aplicação dos padrões de projeto gera confiança em relação a eficácia no desenvolvimento de sistemas, devido a reutilização de soluções. Um grande problema pode ser decomposto em problemas menores. Assim, podem-se encontrar algumas similaridades com problemas resolvidos anteriormente e aplicar os padrões para obter as soluções desses problemas menores. Solucionado os problemas menores consequentemente soluciona o problema principal.

"O padrão é uma descrição do problema e essência de uma solução, onde pode der reutilizada em diversos casos. O padrão não é uma especificação detalhada, pode-se pensar em uma descrição de conhecimento e experiência acumulados." (SOMMERVILLE, 2003).

O uso dos padrões de projeto como referência (nome do padrão) para soluções de problemas identifica o seu comportamento e propósito permitindo o seu entendimento no processo de desenvolvimento de sistemas. Utilizar o nome do padrão facilita no entendimento de sua funcionalidade do que detalhar como uma implementação deveria trabalhar.

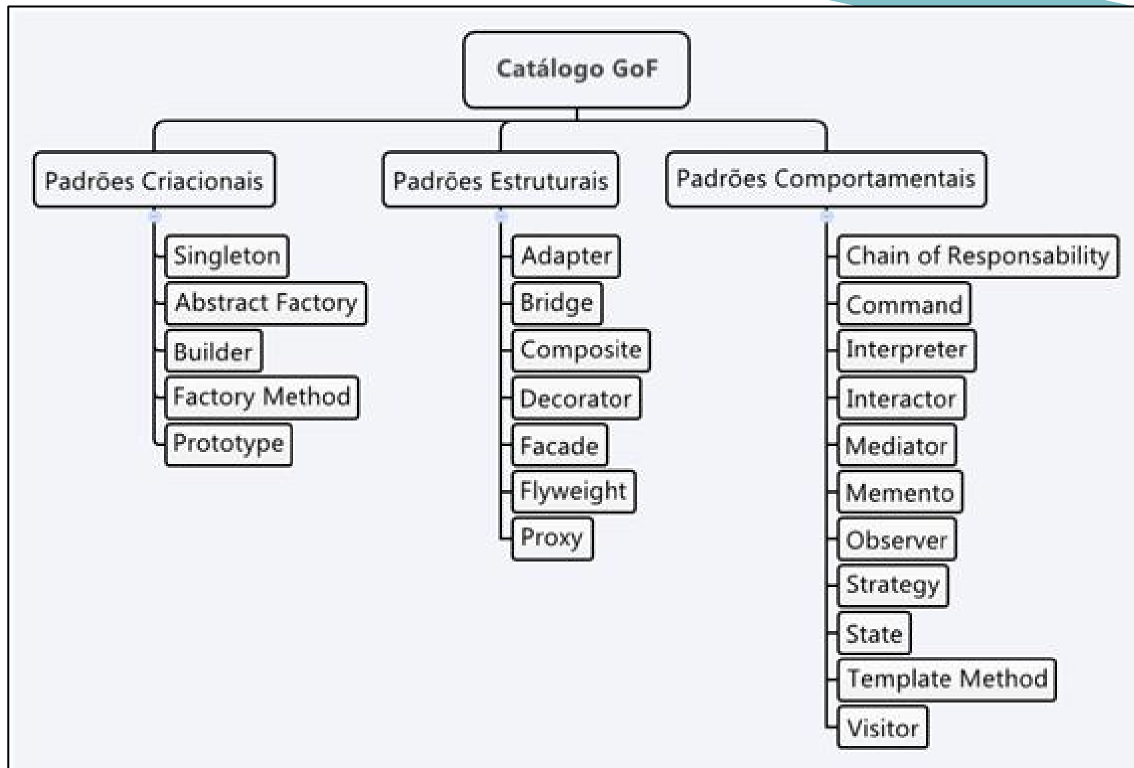
Um padrão possui alguns elementos relevantes, são eles (GAMMA et al., 2007) (GUERRA, 2014):

a) **Nome do Projeto**: nome de referência (uma ou duas palavras) que descreve um problema, solução ou consequência de um projeto;
b) **Problema**: descrição de um problema a ser resolvida através de algoritmos, estruturas de classe ou objetos, que tem soluções inflexíveis e que precisam ser analisados para obtenção de uma resposta;
c) **Solução**: define os elementos que fazem parte do padrão de projeto, seus relacionamentos, comportamentos e responsabilidades;
d) **Consequências**: O padrão também trata da estrutura, da dinâmica, das consequências negativas e positivas da aplicação. Além de oferecer outras opções que complementem para compensar algumas desvantagens e vantagens. Nem sempre é necessário usar padrão para resolver um determinado contexto, pois este é apenas um dos meios praticáveis para desenvolver projetos que tragam resultados, concretos e rápidos;
e) **Reuso e Herança**: uma das principais características na programação orientada a objetos é a utilização da herança, através dela que o reuso tem aumentado o potencial dos métodos de programação orientado a objetos e também dos padrões de projetos, pois foi graças a sua utilização que este conceito tem conquistado grandes desenvolvedores;

Padrões de Projeto GoF

Um catálogo dos padrões de projeto foi feito por quatro projetistas (conhecidos como "*Gang of Four*" - GoF) com base no paradigma de orientação a objetos, cujos padrões (vinte e três padrões coletados) (figura 1) foram organizados no livro "*Design Patterns: Elements of Reusable Object-Oriented Software*" (GAMMA et al., 1995).

Figura 1 - Catálogo GoF.



Os Padrões de Criação abstraem o processo de instanciação. Torna um sistema independente de como seus objetos são criados, compostos e representados. Enquanto uma criação de classe através da herança varia a instancia, a criação de objeto delega a instancia para outro objeto.

Nos Padrões Estruturais é feita descrição da elaboração, associação e a organização entre objetos e classes/interfaces. Tem a possibilidade de relacionar objetos em estruturas mais complexas, ou definir como as classes são compostas ou herdadas a partir de outras.

Já nos Padrões Comportamentais, definem o processo de comunicação entre objetos e classes.

Segundo Gamma et al. (2007, p. 24-25), a intenção de cada padrão é listado a seguir:

a) **Abstract Factory**: "Fornece uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas."
b) **Adapter**: "Converte a interface de uma classe em outra interface esperada pelos clientes. O Adapter permite que certas classes trabalhem em conjunto, pois de outra forma seria impossível por causa de suas interfaces incompatíveis."
c) **Bridge**: "Separa uma abstração da sua implementação, de modo que as duas possam variar independentemente."
d) **Builder**: "Separa a construção de um objeto complexo da sua representação, de modo que o mesmo processo de construção possa criar diferentes representações."
e) **Chain of Responsibility**: "Evita o acoplamento do remetente de uma solicitação ao seu destinatário, dando a mais de um objeto a chance de tratar a solicitação. Encadeia os objetos receptores e passa a solicitação ao longo da cadeia até que um objeto a trate."
f) **Command**: "Encapsula uma solicitação como um objeto, desta forma permitindo que você parametrize clientes com diferentes solicitações, enfileire ou registre (log) solicitações e suporte operações que podem ser desfeitas."
g) **Composite**: "Compõe objetos em estrutura de árvore para representar hierarquias do tipo partes-todo. O Composite permite que os clientes tratem objetos individuais e composições de objetos de maneira uniforme."
h) **Decorator**: "Atribui responsabilidades adicionais a um objeto dinamicamente. Os decorators fornecem uma alternativa flexível a subclasses para extensão da funcionalidade."

i) **Façade**: "Fornece uma interface unificada para um conjunto de interfaces em um subsistema. O Façade define uma interface de nível mais alto que torna o subsistema mais fácil de usar."
j) **Factory Method**: "Define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe a ser instanciada. O Factory Method permite a uma classe postergar (defer) a instanciação às subclasses."
k) **Flyweight**: "Usa compartilhamento para suportar grandes quantidades de objetos, de granularidade fina, de maneira eficiente."
l) **Interpreter**: "Dada uma linguagem, define uma representação para sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças nessa linguagem."
m) **Iterator**: "Fornece uma maneira de acessar seqüencialmente os elementos de uma agregação de objetos sem expor sua representação subjacente."
n) **Mediator**: "Define um objeto que encapsula a forma como um conjunto de objetos interage. O Mediator promove o acoplamento fraco ao evitar que os objetos se refiram explicitamente uns aos outros, permitindo que você varie suas interações independentemente."
o) **Memento**: "Sem violar o encapsulamento, captura e externaliza um estado interno de um objeto, de modo que o mesmo possa posteriormente ser restaurado para este estado."
p) **Observer**: "Define uma dependência um-para-muitos entre objetos, de modo que, quando um objeto muda de estado, todos os seus dependentes são automaticamente notificados e atualizados."
q) **Prototype**: "Especifica os tipos de objetos a serem criados usando uma instância prototípica e criar novos objetos copiando esse protótipo."
r) **Proxy**: "Fornece um objeto representante (surrogate), ou um marcador de outro objeto, para controlar o acesso ao mesmo."
s) **Singleton**: "Garante que uma classe tenha somente uma instância e fornece um ponto global de acesso para ela."
t) **State**: "Permite que um objeto altere seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado de classe."
u) **Strategy**: "Define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. O Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam."
v) **Template Method**: "Define o esqueleto de um algoritmo em uma operação, postergando a definição de alguns passos para subclasses. O Template Method permite que as subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura."
w) **Visitor**: "Representa uma operação a ser executada sobre os elementos da estrutura de um objeto. O Visitor permite que você defina uma nova operação sem mudar as classes dos elementos sobre os quais opera."

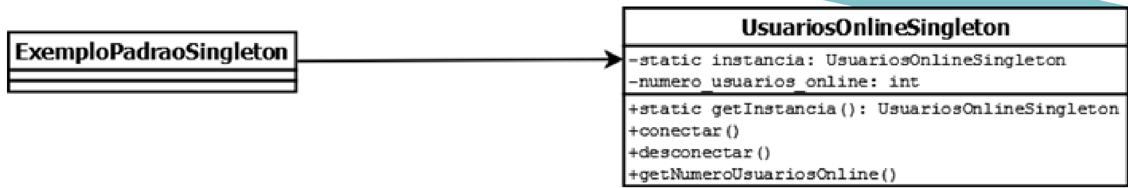
Para exemplificar o uso do padrão de projeto GoF será ilustrado um padrão de cada categoria (Criacional, Estrutural e Comportamental) utilizando o diagrama de classes como modelagem (LIMA, 2011) e a linguagem orientada a objetos JAVA como implementação.

Exemplo - Padrão *Singleton* (categoria Criacional)

Nesse padrão uma única instância da classe define o construtor da classe Singleton como privado (private) e o método que retornará a instância dessa classe como estático (static).

No exemplo a seguir, o uso do padrão Singleton permitirá a contagem de número de usuários online. Nota-se que na aplicação principal ExemploPadraoSingleton três objetos instanciam a mesma classe UsuariosOnlineSingleton, compartilhando a mesma informação armazenada no atributo numero_usuarios_online. O diagrama de classes é ilustrado na figura 2 e sua codificação nos quadros 1 e 2.

Figura 2 - Exemplo de diagrama de classes para o uso do padrão *Singleton*.



Quadro 1 - Implementação da classe *UsuariosOnlineSingleton*.

Arquivo: UsuariosOnlineSingleton.java

```
01  public class UsuariosOnlineSingleton {
02          private static UsuariosOnlineSingleton instancia;
03          private int numero_usuarios_online = 0;
04
05          private UsuariosOnlineSingleton() {
06          }
07          public static UsuariosOnlineSingleton getInstancia() {
08              if (instancia == null) {
09                  instancia = new UsuariosOnlineSingleton();
10              }
11              return instancia;
12          }
13          public void conectar() {
14              numero_usuarios_online += 1;
15          }
16          public void desconectar() {
17              numero_usuarios_online -= 1;
18              if (numero_usuarios_online < 0) {
19                  numero_usuarios_online = 0;
20              }
21          }
22          public void getNumeroUsuariosOnline() {
23              System.out.println("Número de usuários online.: " + numero_usuarios_online);
24          }
25  }
```

Quadro 2 - Implementação da classe principal *ExemploPadroSingleton*.

Arquivo: ExemploPadraoSingleton.java

```
01 public class ExemploPadraoSingleton {
02         public static void main(String[] args) {
03             UsuariosOnlineSingleton usuarios_online =
04                                     UsuariosOnlineSingleton.getInstancia();
05             UsuariosOnlineSingleton usuario01 = UsuariosOnlineSingleton.getInstancia();
06             usuario01.conectar();
07             System.out.println("Usuário 01 conectado...");
08             usuarios_online.getNumeroUsuariosOnline();
09             System.out.println();
10             UsuariosOnlineSingleton usuario02 = UsuariosOnlineSingleton.getInstancia();
11             usuario02.conectar();
12             System.out.println("Usuário 02 conectado...");
13             usuarios_online.getNumeroUsuariosOnline();
14             System.out.println();
15             usuario01.desconectar();
16             System.out.println("Usuário 01 desconectado...");
17             usuarios_online.getNumeroUsuariosOnline();
18         }
19 }
```

O resultado desse exemplo é ilustrado no quadro 3. Observa-se que os objetos usuario01, usuario02 e *usuarios_online* compartilham a mesma instância da classe *UsuáriosOnlineSingleton*, ou seja, houve uma única instância da mesma classe.

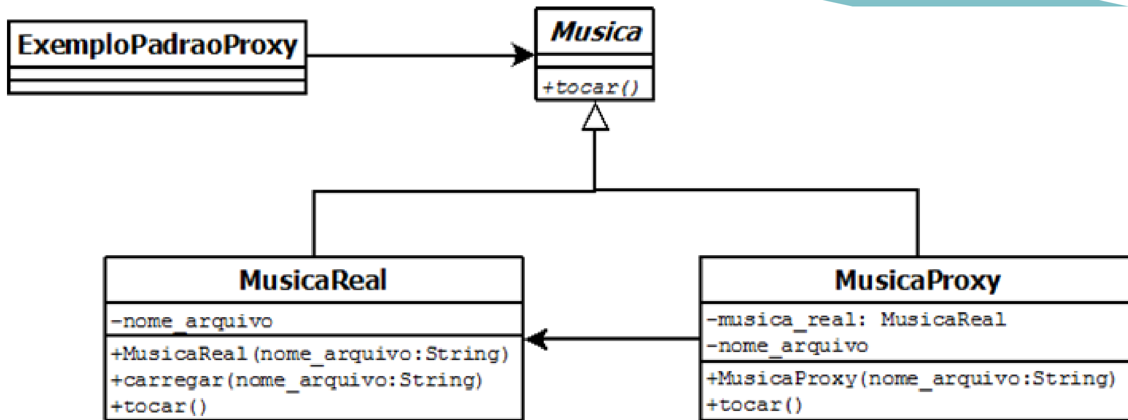
Quadro 3 - Resultado após a execução do padrão *Singleton*.

```
Usuário 01 conectado...
Número de usuários online.: 1

Usuário 02 conectado...
Número de usuários online.: 2

Usuário 01 desconectado...
Número de usuários online.: 1
```

Exemplo - Padrão Proxy (categoria Estrutural)

Para exemplificar esse padrão a figura 3 mostra uma interface para tocar músicas. Na primeira vez que a interface tocar é utilizada, é feito o carregamento do arquivo e na sequência a execução da música(classe MusicaReal). Caso utilize a interface tocar para a mesma música será utilizado a interface tocar da classe MusicaProxy, pois o arquivo já se encontra carregado.

Figura 3- Exemplo de diagrama de classes para o uso do padrão *Proxy*.



No quadro 4 ilustra a declaração da interface tocar.

Quadro 4 – Declaração da interface tocar na classe *Musica*.

Arquivo: Musica.java	
01	public interface Musica {
02	public void tocar();
03	}

A codificação da classe *MusicaReal* é mostrado no quadro 5. Nessa classe é implementado o método carregar que simula o carregamento de um arquivo e ilustra a implementação da interface tocar.

Quadro 5 - Implementação da classe *MusicaReal*.

Arquivo: MusicaReal.java	
01	public class MusicaReal implements Musica {
02	private String nome_arquivo;
03	public MusicaReal(String nome_arquivo) {
04	this.nome_arquivo = nome_arquivo;
05	carregar(nome_arquivo);
06	}
07	private void carregar(String nome_arquivo) {
08	System.out.println("Carregando o arquivo " + nome_arquivo + " ...");
09	}
10	@Override
11	public void tocar() {
12	System.out.println("Reproduzindo a música " + this.nome_arquivo + " ...");
13	}
14	}

A classe *MusicaProxy* ilustra o propósito do padrão (quadro 6). Nota-se que é feita uma instância à classe *MusicaReal* ao objeto *musica_real* caso seja nulo e na sequência solicita a interface tocar da classe *MusicaReal*. E se objeto já havia sido instanciado, simplesmente toca a música, pois o arquivo já foi carregado anteriormente (quadro 7).

Quadro 6 - Implementação da classe *MusicaProxy*.

Arquivo: MusicaProxy.java

```
01 public class MusicaProxy implements Musica {
02         private MusicaReal musica_real;
03         private String nome_arquivo;
04         public MusicaProxy(String nome_arquivo) {
05             this.nome_arquivo = nome_arquivo;
06         }
07         @Override
08         public void tocar() {
09             if (musica_real == null) {
10                 musica_real = new MusicaReal(this.nome_arquivo);
11             }
12             musica_real.tocar();
13         }
14 }
```

Quadro 7 - Implementação da classe principal *ExemploPadraoProxy*.

Arquivo: ExemploPadraoProxy.java

```
01 public class ExemploPadraoProxy {
02         public static void main(String[] args) {
03             Musica musica = new MusicaProxy("musica.mp3");
04             //Carrega o arquivo e reproduz a música
05             musica.tocar();
06             System.out.println();
07             //Reproduz a música
08             musica.tocar();
09         }
10 }
```

O resultado desse padrão é mostrado no quadro 8.

Quadro 8 - Resultado após a execução do padrão *Proxy*.

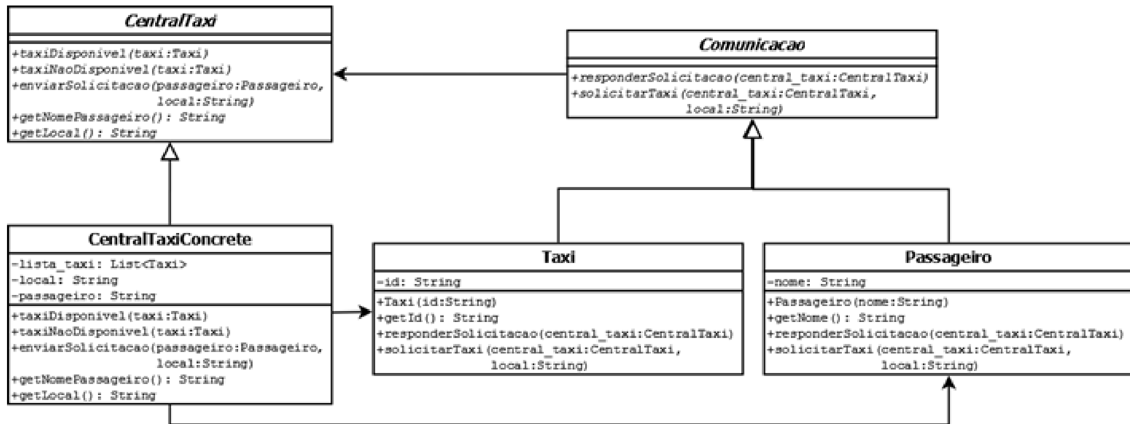
```
Carregando o arquivo musica.mp3 ...
Reproduzindo a música musica.mp3 ...

Reproduzindo a música musica.mp3 ...
```

Exemplo – Padrão *Mediator* (categoria Comportamental)

Este padrão é usado para reduzir a complexidade de comunicação entre vários objetos ou classes. O padrão fornece uma classe mediadora que normalmente lida com todas as comunicações entre diferentes classes e suporta fácil manutenção do código.

A figura 4 mostra o exemplo do uso do padrão *Mediator*. A central de taxi faz o papel de mediador entre a solicitação do passageiro com o taxi solicitado. Note que não há uma comunicação direta entre o passageiro e o taxi, ou seja, tudo passa pela central de taxi.

Figura 4 - Exemplo de diagrama de classes do padrão *Mediator*.



Inicialmente são declaradas as interfaces nas classes abstratas *CentralTaxi* e *Comunicacao* (quadros 9 e 10).

Quadro 9- Declaração das interfaces na classe abstrata *CentralTaxi*.

Arquivo: CentralTaxi.java	
01	public abstract class CentralTaxi {
02	public abstract void taxiDisponivel(Taxi taxi);
03	public abstract void taxiNaoDisponivel(Taxi taxi);
04	public abstract void enviarSolicitacao(Passageiro passageiro, String local);
05	public abstract String getNomePassageiro();
06	public abstract String getLocal();
07	}

Quadro 10 - Declaração das interfaces na classe abstrata *Comunicacao*.

Arquivo: Comunicacao.java	
01	public abstract class Comunicacao {
02	public abstract void responderSolicitacao(CentralTaxi central_taxi);
03	public abstract void solicitarTaxi(CentralTaxi central_taxi, String local);
04	}

Nos quadros 11 e 12 são codificadas as classes Taxi e Passageiro e implementam as interfaces da classe abstrata *Comunicacao*, respectivamente.

Quadro 11 - Implementação da classe *Taxi*.

Arquivo: Taxi.java

```
01 public class Taxi extends Comunicacao {
02         private String id;
03         public Taxi(String id){
04             this.id = id;
05         }
06         public String getId() {
07             return id;
08         }
09         @Override
10         public void responderSolicitacao(CentralTaxi central_taxi) {
11             System.out.println(this.getId() + " -> Solicitação respondida. Passageiro: "
12                     + central_taxi.getNomePassageiro() + " - Local: "
13                     + central_taxi.getLocal()+".");
14             central_taxi.taxiNaoDisponivel(this);
15         }
16         @Override
17         public void solicitarTaxi(CentralTaxi central_taxi, String local) {
18             //Nada a fazer aqui
19         }
20 }
```

Quadro 12 - Implementação da classe *Passageiro*.

Arquivo: Passageiro.java

```
01 public class Passageiro extends Comunicacao {
02         private String nome;
03         public Passageiro(String nome) {
04             this.nome = nome;
05         }
06         public String getNome() {
07             return this.nome;
08         }
09         @Override
10         public void solicitarTaxi(CentralTaxi central_taxi, String local) {
11             central_taxi.enviarSolicitacao(this, local);
12         }
13         @Override
14         public void responderSolicitacao(CentralTaxi central_taxi) {
15             //Nada a fazer aqui
```

Arquivo: Passageiro.java

```
16          }
17  }
```

No quadro 13 é feita a implementação das interfaces da classe abstrata *CentralTaxi*.

Quadro 13 - Implementação da classe *CentralTaxiConcrete*.

Arquivo: CentralTaxiConcrete.java

```
01  import java.util.ArrayList;
02  import java.util.List;
03  public class CentralTaxiConcrete extends CentralTaxi {
04          private List<Taxi> lista_taxi = new ArrayList<>();
05          private String local;
06          private String passageiro;
07          @Override
08          public String getNomePassageiro() {
09                  return this.passageiro;
10          }
11          @Override
12          public String getLocal() {
13                  return this.local;
14          }
15          @Override
16          public void taxiDisponivel(Taxi taxi) {
17                  this.lista_taxi.add(taxi);
18          }
19          @Override
20          public void enviarSolicitacao(Passageiro passageiro, String local) {
21                  this.passageiro = passageiro.getNome();
22                  this.local = local;
23                  for(Taxi taxi : lista_taxi) {
24                          System.out.println("Central -> " + taxi.getId() + "[Passageiro: " +
25                                                  passageiro.getNome() + ", Local: " + local + "]");
26                  }
27                  System.out.println();
28          }
29          @Override
30          public void taxiNaoDisponivel(Taxi taxi) {
31                  this.lista_taxi.remove(taxi);
32                  System.out.println("Central -> " + taxi.getId() + " OK.");
33                  System.out.println();
```

Arquivo: CentralTaxiConcrete.java

```
34          }
35  }
```

Pode-se ver no quadro 14 o uso do padrão *Mediator*. São criados a central de taxi, os taxistas e os passageiros e, são determinados as disponibilidades dos taxistas e as solicitações dos passageiros.

Quadro 14 - Implementação da classe principal *ExemploPadraoMediator*.

Arquivo: ExemploPadraoMediator.java

```
01  public class ExemploPadraoMediator {
02          public static void main(String[] args) {
03              CentralTaxi central_taxi = new CentralTaxiConcrete();
04              Taxi taxi01 = new Taxi("TAXI 01");
05              Taxi taxi02 = new Taxi("TAXI 02");
06              Taxi taxi03 = new Taxi("TAXI 03");
07              Passageiro passageiro01 = new Passageiro("PASSAGEIRO 01");
08              Passageiro passageiro02 = new Passageiro("PASSAGEIRO 02");
09              central_taxi.taxiDisponivel(taxi01);
10              central_taxi.taxiDisponivel(taxi02);
11              central_taxi.taxiDisponivel(taxi03);
12              passageiro01.solicitarTaxi(central_taxi, "RUA 01");
13              taxi02.responderSolicitacao(central_taxi);
14              passageiro02.solicitarTaxi(central_taxi, "RUA 02");
15              taxi01.responderSolicitacao(central_taxi);
16          }
17  }
```

O quadro 15 ilustra o resultado do padrão *Mediator*, exibindo toda comunicação entre central do taxi com o taxi e o passageiro.

Quadro 15 - Resultado após a execução do padrão *Mediator*.

```
Central -> TAXI 01[Passageiro: PASSAGEIRO 01, Local: RUA 01]
Central -> TAXI 02[Passageiro: PASSAGEIRO 01, Local: RUA 01]
Central -> TAXI 03[Passageiro: PASSAGEIRO 01, Local: RUA 01]

TAXI 02 -> Solicitação respondida. Passageiro: PASSAGEIRO 01 - Local: RUA 01.
Central -> TAXI 02 OK.

Central -> TAXI 01[Passageiro: PASSAGEIRO 02, Local: RUA 02]
Central -> TAXI 03[Passageiro: PASSAGEIRO 02, Local: RUA 02]

TAXI 01 -> Solicitação respondida. Passageiro: PASSAGEIRO 02 - Local: RUA 02.
Central -> TAXI 01 OK
```

CONSIDERAÇÕES FINAIS

Os padrões de projeto representam as melhores práticas utilizadas por desenvolvedores de software orientado a objetos experientes. Apresentam soluções para os problemas gerais que desenvolvedores de software encontram durante o seu desenvolvimento. Estas soluções foram obtidas por tentativa e erro por numerosos desenvolvedores de software sobre um período de tempo significativo.

Vale ressaltar que padrões de projeto promovem a reutilização que leva ao código mais robusto e altamente sustentável. Ele ajuda a reduzir o custo total de propriedade de software.

Com os padrões de projeto já definidos, faz o código ser fácil de entender e depurar. Isso leva a um desenvolvimento mais rápido e os novos membros de equipe de desenvolvimento de entendê-la facilmente.

As possibilidades são muitas para implementação e padronização de projetos baseado no catálogo GoF. Com a análise correta e propicia o desenvolvedor poderá trabalhar de forma dinâmica, experimental e até conclusiva a respeito de padrões.

O uso dos padrões de projeto como referência resultará em um software com qualidade significativa em todo o processo de desenvolvimento.

REFERÊNCIAS

GAMMA, Erich et al. Design Patterns: Elements of Reusable Object-Oriented Software. Londres: Addison-Wesley Longman, 1995.

GAMMA, Erich et al. Padrões de Projeto – Soluções Reutilizáveis de Software Orientado a Objetos. Porto Alegre: Bookman Companhia, 2007.

GUERRA, Eduardo. Design Patterns com JAVA – Projeto Orientado a Objetos Guiado por Padrões. São Paulo: Casa do Código, 2014.

LIMA, Adilson da Silva. UML 2.3 – Do Requisito a Solução. São Paulo: Erica, 2011.

SOMMERVILLE, Ian. Engenharia de Software. 6ª.ed. São Paulo: Addison Wesley, 2003.