UNIVERSIDADE FEDERAL DE JUIZ DE FORA
ESPECIALIZAÇÃO EM DESENVOLVIMENTO DE SISTEMAS COM TECNOLOGIA JAVA

PADRÕES DE PROJETO COM JAVA

TRABALHO DE CONCLUSÃO

RAFAEL FRANCELINO FERREIRA MENDES VIEIRA

Juiz de Fora, Minas Gerais, Brasil
2011

PADRÕES DE PROJETOS COM JAVA

por

Rafael Francelino Ferreira Mendes Vieira

Trabalho de conclusão apresentado à Universidade Federal de Juiz de Fora para obtenção de especialização de Desenvolvimento de Sistemas com Tecnologia Java.

Juiz de Fora, Minas Gerais, Brasil

2011

UNIVERSIDADE FEDERAL DE JUIZ DE FORA
CURSO DE ESPECIALIZAÇÃO

A comissão examinadora abaixo assinada,
aprova o trabalho de conclusão:

PADRÕES DE PROJETOS COM JAVA

elaborado por

RAFAEL FRANCELINO FERREIRA MENDES VIEIRA

como requisito parcial para obtenção de especialização em

DESENVOLVIMENTO DE SISTEMAS COM TECNOLOGIA JAVA

_____

José Honório Glanzmann

_____

Eduardo Barrére

_____

Tarcísio de Souza Lima

Juiz de Fora, Minas Gerais, Brasil
2011

Dedico este trabalho com muito amor e imensa gratidão à madrinha Maria Natalina Ferreira, que contribuiu muito ao longo desta caminhada.

AGRADECIMENTOS

A Deus, meu único salvador, pelo dom da vida e por ter me ajudado todos os dias nesta caminhada.

Aos meus pais, Moacir Francelino e Regina Mirian, que lutaram com muita ousadia junto comigo para que este sonho tornasse realidade. Por estarem comigo nas horas difíceis e nas horas de alegria.

A minha noiva, Eliúde Damásio, por ter compreendido minhas ausências e por ter me oferecido o mais belo e doce amor.

Às queridas tias: Cornélia, Maria José, Regina e Maria Mafalda pelo grande apoio e dedicação que tiveram por mim.

Aos meus amigos, pelas orações e pensamentos positivos para que eu pudesse alcançar meus objetivos.

“Uma descoberta, seja feita por um menino na escola ou por um cientista trabalhando na fronteira do conhecimento, é em sua essência uma questão de reorganizar ou transformar evidências, de tal forma que se possa ir além delas assim reorganizadas, rumo a novas percepções”.

Jerone Bruner

RESUMO

A utilização do paradigma orientação a objetos traz como benefício o reuso de artefatos produzidos no desenvolvimento de softwares. Esse paradigma é utilizado na linguagem de programação Java, agregando simplicidade, robustez, segurança, portabilidade e ótimo desempenho. Os padrões de projeto são experiências orientadas a objeto que descrevem soluções para problemas comumente encontrados no desenvolvimento de softwares. Este trabalho descreve orientação a objetos e seus conceitos; a linguagem de programação Java e seus benefícios; e padrões de projeto, trazendo ao final exemplos de como e onde utilizá-los.

Palavras chave: Orientação a objetos. Java. Padrões de Projeto. Software.

ABSTRACT

The use of object-oriented paradigm brings benefit to the reuse of artifacts produced in software development. This paradigm is used in the Java programming language, combining simplicity, robustness, security, portability and optimal performance. The design patterns are object-oriented experiences that describe solutions to problems commonly encountered in software development. This paper describes object orientation and its concepts, the Java programming language and its benefits, and design patterns, bringing the final examples of how and where to use them.

Keywords: Object Orientation. Java. Design Patterns. Software.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

CSS	*Cascading Style Sheets*
DAO	*Data Access Object*
DBA	*Database Administrator*
HTML	*HyperText Markup Language*
IP	*Internet Protocol*
JVM	*Java Virtual Machine*
MVC	*Model-View-Controller*
OO	Orientação a Objetos
POO	Programação Orientada a Objetos
SQL	*Structured Query Language*
TCP	*Transmission Control Protocol*
URL	*Uniform Resource Locator*

SUMÁRIO

INTRODUÇÃO

A produção de *software* cresce cada vez mais, na medida em que empresas tomam a decisão de automatizar seus processos, em busca de competitividade, redução de custos e melhorias nos processos de negócio.

A necessidade de desenvolver sistemas com alta qualidade, de maneira rápida e eficaz, faz com que haja uma busca de tecnologias e técnicas que suportem esse contexto. A Programação Orientada a Objetos (POO), a linguagem Java e a utilização de Padrões de Projetos, juntos, ganham espaço neste contexto, apresentando várias vantagens quando utilizados corretamente.

O objetivo desse artigo é apresentar os principais conceitos e benefícios da utilização de padrões de projeto, orientação a objetos e Java no desenvolvimento de softwares.

Na primeira seção, é apresentado o conceito de programação orientada a objetos, com destaque nos benefícios da sua utilização. São descritos, juntamente com exemplos, classes, objetos, encapsulamento, herança, polimorfismo e abstração.

A segunda seção contém uma breve descrição sobre a linguagem de programação Java, com destaque nas suas principais características.

A terceira seção aborda padrões de projeto, onde são descritos suas características e benefícios quando utilizados. São apresentados três padrões muito utilizados no desenvolvimento de *softwares*: *Model-View-Controller*, *Facade* e *Data Access Object*.

Os exemplos em Java utilizados para melhor compreensão dos conceitos são partes do DB Música: Base de Dados para Aprendizado de Harmonia.

1 PROGRAMAÇÃO ORIENTADA A OBJETOS

Para Sintes (2002), a programação orientada a objetos (POO) é uma técnica para a montagem de sistemas de computador que simula o modo como os objetos são montados no mundo físico, o mundo real. Usando esse pensamento para o desenvolvimento, podem ser criados sistemas utilizando o conceito de reutilização. As funcionalidades criadas poderão ser reutilizadas em outros softwares.

Segundo Cadenhead e Lemay (2003), um software, no contexto de programação orientada a objetos, é conceituado como um conjunto de objetos que trabalham juntos para realizar uma determinada tarefa.

A POO facilita a reutilização de softwares, trechos de códigos e análises já realizadas. Com esse paradigma aumentaram as motivações para a elaboração e implementação de sistemas modulares, bem como a reutilização de bibliotecas desenvolvidas anteriormente.

A seguir, são apresentados os principais conceitos ligados a POO, a partir de exemplos de códigos em Java. As classes apresentadas compõem o

software “DB Música: Base de dados para aprendizado de Harmonia” (VIEIRA, 2009), desenvolvido como trabalho de conclusão de curso de graduação do autor. A primeira versão do DB Música foi desenvolvida utilizando a linguagem de programação PHP. A segunda versão foi desenvolvida em Java, de onde foram retirados os exemplos que compõe este trabalho.

1.1 CLASSES E OBJETOS

Uma classe é uma estrutura estática que é utilizada para instanciar objetos. Possui atributos (propriedades) e métodos (funcionalidades). É um modelo para criação de “coisas” do mundo real. Essas “coisas” são denominadas Objetos.

> Na terminologia da orientação a objetos, cada ideia é denominada *classe de objetos*, ou simplesmente *classe*. Uma classe é uma descrição dos atributos e serviços comuns a um grupo de objetos. Sendo assim, pode-se entender uma *classe* como sendo um molde a partir do qual objetos são construídos. Ainda sobre terminologia, diz-se que um objeto é uma *instância* de uma classe. (BEZERRA, 2002.).

O exemplo abaixo mostra uma classe implementada em Java, denominada “EstiloMusical”. Essa classe contém os atributos id (identificador do estilo musical), nome, paisOrigem (país que originou o estilo musical) e descrição. Há também os métodos setId, getId, setNome, getNome, setPaisOrigem, getPaisOrigem, setDescricao e getDescricao.

```
public class EstiloMusical {

   //atributos
   private Integer id;
```

```
    private String nome;
    private String paisOrigem;
    private String descricao;

    //métodos
    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getNome() {
        return nome;
    }

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

    public String getPaisOrigem() {
        return paisOrigem;
    }

    public void setPaisOrigem(String paisOrigem) {
        this.paisOrigem = paisOrigem;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

}
```

Listagem 1 – Código fonte da classe EstiloMusical, que compõe o DB Música.
Fonte: VIEIRA, 2009.

Um objeto dessa classe é qualquer estilo musical do mundo real, como por exemplo o *Jazz*. Seu nome é "*Jazz*", seu país de origem são os "Estados Unidos" e uma descrição para ele é "surgiu no século XX na região de Nova Orleans e

suas proximidades”. Essa instância é um exemplo do mundo físico, ou seja, um objeto para a classe EstiloMusical.

1.2 ENCAPSULAMENTO

Existem atributos de classe que devem ser tratados exclusivamente pelos métodos dela mesma, manipulando-os de forma correta, fornecendo mais controle e segurança sobre a aplicação. O encapsulamento permite essa forma de manipulação desses atributos.

> O encapsulamento proíbe a visualização interna de um objeto. Os atributos são associados a métodos e só podem ser acessados por estes. A interface para cada classe é definida de forma a revelar o menos possível sobre seu funcionamento interno. Os métodos possuem acesso aos atributos da classe para a qual foram definidos, enquanto os atributos de uma classe só podem ser manipulados por métodos desta classe (SINTES, 2002).

No exemplo da classe “EstiloMusical”, citado acima (listagem 1), os atributos estão encapsulados. O acesso a eles é feito somente através dos métodos *Get* e *Set*. Um valor é atribuído ao *nome* somente pelo método *setNome* e esse mesmo valor só é recuperado através do método *getNome.*

1.3 HERANÇA

Na POO, há classes que são chamadas de “pai”, as superclasses*;* e outras chamadas de “filha”*,* as subclasses*.* As subclasses herdam características de suas respectivas superclasses*.* Essas características são os atributos e métodos, permitindo um alto reaproveitamento de código no desenvolvimento de sistemas.

Segundo Minetto (2008), com o mecanismo da herança podemos definir uma classe como sendo uma classe-filha de outra. Isto significa que a nova classe herda todas as características da classe-pai.

O maior benefício em utilizar esse conceito é o reaproveitamento de código. A seguir, serão descritas duas classes:

A superclasse “Pessoa” e a subclasse “Interprete”.

```
public abstract class Pessoa {

    private Integer id;
    private String nomeArtistico;
    private String descricao;

    public Pessoa(Integer id, String nomeArtistico, String descricao){
        this.setId(id);
        this.setNomeArtistico(nomeArtistico);
        this.setDescricao(descricao);
    }

    public Integer getId() {
        return id;
```

```
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getNomeArtistico() {
        return nomeArtistico;
    }

    public void setNomeArtistico(String nomeArtistico) {
        this.nomeArtistico = nomeArtistico;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public abstract void descreveClasse();

}
```

Listagem 2 – Código fonte da superclasse Pessoa, que compõe o DB Música.
Fonte: VIEIRA, 2009.

```
public class Interprete extends Pessoa{

    private String paisOrigem;

    public Interprete(Integer id, String nomeArtistico, String descricao, String
paisOrigem) {
        super(id, nomeArtistico, descricao);
        this.setPaisOrigem(paisOrigem);
    }

    public String getPaisOrigem() {
        return paisOrigem;
    }

    public void setPaisOrigem(String paisOrigem) {
        this.paisOrigem = paisOrigem;
    }

    @Override
```

```
        public void descreveClasse() {
            System.out.println("Classe (Intérprete) para criar objetos do mundo
real (intérpretes).");
        }

    }
```

Listagem 3 – Código fonte da subclasse Interprete, que compõe o DB Música.
Fonte: VIEIRA, 2009.

A subclasse Interprete, através do comando "*extends*" herda os atributos e os métodos da superclasse Pessoa.

Além do seu atributo específico chamado *paisOrigem* e de seus métodos chamados *setPaisOrigem* e *getPaisOrigem*, a subclasse passa a conter através da herança os atributos *id*, *nomeArtistico* e *descrição*, bem como os métodos: *getId, setId, getNomeArtistico, setNomeArtistico, getDescricao* e *setDescricao*.

1.4 POLIMORFISMO

O significado da palavra polimorfismo nos remete a "muitas formas". Na POO, permite que classes diferentes, filhas de uma mesma superclasse, tenham métodos com o mesmo nome e mesmos parâmetros, mas que tenham comportamentos diferentes, definidos em cada uma das classes filhas.

> O polimorfismo é a possibilidade de enviar um mesmo seletor de mensagem para diferentes objetos mesmo que estes sejam instâncias de classes diferentes, isso significa que um mesmo método pode atuar de modos diferentes em classes diferentes. (SINTES, 2002).

Com esse conceito, podemos criar códigos com maior facilidade de leitura e promover a extensibilidade em um *software*, como pode ser visto a seguir:

> Com o polimorfismo, podemos projetar e implementar sistemas que são facilmente extensíveis – novas classes podem ser adicionadas a partes gerais do programa com pouca ou nenhuma modificação, contando que as novas classes façam parte da hierarquia de herança que o programa processa genericamente. As únicas partes de um programa que devem ser alteradas para acomodar as novas classes são aquelas que exigem conhecimento direto das novas classes que o programador adiciona à hierarquia (DEITEL; DEITEL, 2005).

Observe que na classe Pessoa existe um método abstrato chamado *descreveClasse.* A declaração desse método como abstrato obriga sua implementação nas classes filhas. Para exemplificar o polimorfismo, esse método assumirá duas formas diferentes, ou seja, imprimirá uma sentença na classe Interprete e outra na classe Compositor, como descrito abaixo.

```
public class Compositor extends Pessoa{

    private String nome;
    private String dataNascimento;
    private String naturalidade;
    private String nacionalidade;

    public Compositor(Integer id, String nomeArtistico, String descricao,
String nome, String dataNascimento, String naturalidade, String nacionalidade) {
        super(id, nomeArtistico, descricao);
        this.setNome(nome);
        this.setDataNascimento(dataNascimento);
        this.setNaturalidade(naturalidade);
        this.setNacionalidade(nacionalidade);
    }

    public String getDataNascimento() {
        return dataNascimento;
    }

    public void setDataNascimento(String dataNascimento) {
        this.dataNascimento = dataNascimento;
```

```
        }

        public String getNacionalidade() {
            return nacionalidade;
        }

        public void setNacionalidade(String nacionalidade) {
            this.nacionalidade = nacionalidade;
        }

        public String getNaturalidade() {
            return naturalidade;
        }

        public void setNaturalidade(String naturalidade) {
            this.naturalidade = naturalidade;
        }

        public String getNome() {
            return nome;
        }

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

        @Override
        public void descreveClasse() {
            System.out.println("Classe (Compositor) para criar objetos do mundo real (compositores).");
        }

    }
```

Listagem 4 – Código fonte da subclasse Compositor, que compõe o DB Música.
Fonte: VIEIRA, 2009.

Na classe Interprete, o método *descreveClasse* exibe a seguinte mensagem: “Classe (Intérprete) para criar objetos do mundo real (intérpretes)”. Já na classe Compositor, esse método é redefinido exibindo a seguinte mensagem: “Classe (Compositor) para criar objetos do mundo real (compositores)”.

1.5 ABSTRAÇÃO

Segundo Dall'Oglio (2008), para construir um bom sistema utilizando a programação orientada a objetos, não se deve projetá-lo com sendo um grande bloco monolítico. Ele deve ser separado em partes, para podermos construir códigos bem definidos que possam ser reaproveitados em outro momento, formando uma estrutura hierárquica.

Nesse contexto, encontraremos classes abstratas, que estão na hierarquia para servirem de base a outras classes. Dessas classes não há possibilidade de instanciar objetos, tendo um tratamento diferente pela linguagem de programação. Apenas nas classes filhas são criados os objetos.

Na listagem 2, superclasse Pessoa, há uma abstração de classe. Com isso não há possibilidade de instanciar objetos.

Pode-se, também, criar métodos abstratos, que são definidos e não implementados nas superclasses, tornando obrigatório a implementação nas classes filhas, como vimos no exemplo de polimorfismo. O método *imprimeNomeClasse* é definido na classe Pessoa e implementado nas classes *Interprete* e *Compositor*.

2 A LINGUAGEM DE PROGRAMAÇÂO JAVA

De acordo com Horstmann e Cornell (2010), Java é uma linguagem de programação orientada a objetos, integrante da plataforma Java, livre e gratuita, que começou a ser desenvolvida em 1991 por um grupo de engenheiros da *Sun Microsystems*, liderados por Patrick Naughton e James Gosling.

O código fonte do Java é compilado e, diferente de outras linguagens, é convertido em *bytecodes.* Em seguida, esses *bytecodes* são interpretados por uma máquina virtual Java (JVM), que simula um computador, ocultando o *hardware* e o sistema operacional, onde ela está instalada. Esse processo faz com que a linguagem seja portável.

Para Deitel (2005), os *bytecodes* são instruções independentes de plataforma, podendo ser executados em qualquer sistema contendo uma JVM que estende a versão do Java em que eles foram compilados.

Segundo Horstmann e Cornell (2010), Java possui, em resumo, onze características chaves:

- **sintaxe simples:** a sintaxe do Java é, de fato, uma versão limpa da sintaxe do C++. Não há necessidade de arquivos de cabeçalho, aritmética de ponteiro (ou mesmo uma sintaxe de ponteiro), estruturas, uniões, sobrecarga de operadores, classes básicas virtuais e assim por diante;

- **suporte a orientação a objetos:** a orientação a objetos provou seu valor nos últimos anos, e é inconcebível que uma linguagem de programação moderna não a utilize. Os recursos OO do Java são semelhantes aos do C++. A principal diferença é a herança múltipla, que o Java substitui pelo conceito de interface;

- **compatível com redes:** o Java possui uma extensa biblioteca de métodos para lidar com protocolos TCP/IP. As capacidades de rede do Java são fortes e fáceis de usar;

- **robusta:** o compilador Java detecta vários problemas que, em outras linguagens, só viriam à tona em tempo de execução. O Java dá ênfase à verificação preliminar de possíveis problemas e tenta eliminar situações propensas a erros;

- **segura:** o Java foi concebido para ser utilizado em ambientes distribuídos. Com isso, muita ênfase foi dada à segurança. Ele foi projetado para tornar certos tipos de ataques impossíveis, entre eles o estouro de pilha em tempo de execução, a corrupção de memória fora do seu próprio espaço de processo e a leitura ou escrita de arquivos sem permissão;

- **arquitetura neutra:** o compilado gera *bytecodes*, que são independentes de uma arquitetura de computador específica. Eles são interpretados por uma máquina virtual Java que, além interpretá-los, melhora a segurança, verificando o comportamento das seqüências de instruções;

- **portável:** diferentemente do C e C++, não há nenhum aspecto dependente de arquitetura ou sistema operacional. Por exemplo, os tamanhos dos tipos de dados primitivos estão especificados, assim como o comportamento da aritmética neles;

- **interpretada:** o interpretador Java pode executar *bytecodes* em qualquer máquina em que o foi instalado. A “grosso modo”, onde um interpretador Java pode ser instalado, executa-se um sistema Java;

- **alto desempenho:** a qualidade dos compiladores *just-in-time* é tão boa que eles concorrem com os tradicionais e, em alguns casos, tem melhor desempenho por possuírem mais informações;

- **suporte a múltiplos *threads*:** *threads* no Java podem tirar vantagem de sistemas com múltiplos processadores se o sistema operacional base suportar isso.

- **dinâmica:** Java é uma linguagem mais dinâmica do que o C e C++. Ele foi projetado para adaptar-se a um ambiente em evolução. As bibliotecas podem adicionar livremente novos métodos e objetos sem nenhum efeito sobre seus clientes.

Diante dessas características, essa forte linguagem de programação ganha destaque no desenvolvimento de sistemas robustos, utilizando orientação a objetos e padrões de projeto.

3 PADRÕES DE PROJETO

O termo Padrão de Projeto (*Design Pattern*) tem sido muito utilizado na programação orientada a objetos com o objetivo de construir formas de relacionamento entre as classes e os objetos, a fim de solucionar problemas específicos.

> Padrões de Projeto ou *Design Pattern* descrevem problemas que ocorrem várias vezes no ambiente de desenvolvimento, logo descrevem o núcleo de soluções para aqueles problemas, num caminho em que se possa utilizar várias vezes a mesma solução, sem ter que repetir o mesmo caminho. (GAMMA; HELM; JOHNSON; VLISSIDES, 2005).

A programação utilizando padrões muitas vezes não é a forma mais rápida de construir códigos, mas, sem dúvida, é a forma que trás maior flexibilidade e capacidade de reutilização da solução. Com isto, se tem uma enorme vantagem quando se trata de manutenção de softwares.

Segundo Fowler (2004), um sistema projetado de forma equivocada geralmente precisa de mais código para realizar as mesmas tarefas, pois o mesmo código foi escrito em vários lugares diferentes. Quanto maior a

dificuldade em visualizar um projeto, a partir de seu código, mais difícil será a sua manutenção e mais rapidamente ele pode se desestruturar. A eliminação da duplicidade é um ponto importante no desenvolvimento de software, trazendo uma diferença enorme na manutenção.

Além dessas vantagens, os padrões formam um vocabulário comum onde qualquer desenvolvedor pode entender como as rotinas foram implementadas, bem como, os módulos do software se interagem.

Segundo Freeman (2005), padrões são experiências orientadas a objeto comprovadas. Eles são compostos basicamente de sua nomenclatura, que é utilizada para comunicação entre desenvolvedores; do problema, que descreve quando utilizar o padrão; da solução, que podem ser utilizadas como gabaritos em futuros problemas semelhantes; e das consequencias, que são os resultados e as análises das vantagens e desvantagens da aplicação do padrão.

De acordo com Gamma (2005) e Pimentel (2010), os padrões de projetos são divididos em três categorias, descritas a seguir:

- **Padrões criacionais:** procuram separar as funções de uma aplicação de como os seus objetos são criados;

- **Padrões estruturais:** provêem generalidade para que a estrutura do projeto possa ser estendida em outro momento;

- **Padrões comportamentais:** utilizam herança para distribuir o comportamento entre as classes filhas, ou agregação e composição para construir um comportamento complexo a partir de componentes mais simples.

A seguir são apresentados alguns padrões de projeto, suas descrições e exemplos práticos em Java.

3.1 *MODEL-VIEW-CONTROLLER*

O *Model-View-Controller* (MVC) é um padrão estrutural que separa os dados do aplicativo (contidos no modelo), os componentes gráficos de apresentação (a visão) e lógica de processamento de entrada (o controlador).

A Ilustração 1 mostra os relacionamentos entre esses componentes.

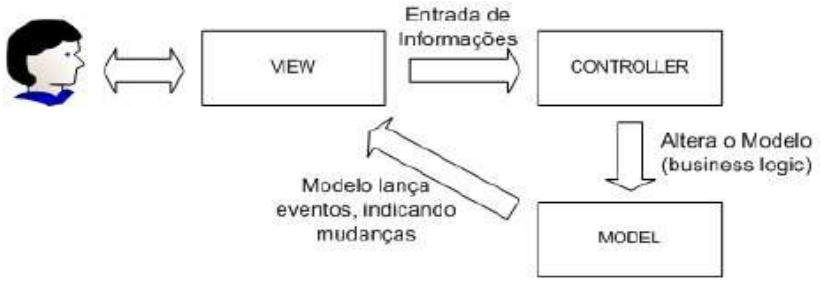

Ilustração 1: Padrão MVC
Fonte: ALMEIDA, [200-?].

A *View* é a interface com o usuário. Fornece meios para entrada de dados ou solicitações, além de formatar a saída das informações a serem apresentadas como resposta. É a parte do sistema visível ao usuário.

O *Controller* é composto de classes que fazem o controle de acesso à aplicação. Comanda a visão e o modelo para se alterarem de forma apropriada de acordo com as necessidades. Ele implementa a lógica para processar entradas do usuário.

Por fim, temos a camada *Model,* que são as classes de modelo da aplicação. Gerenciam o comportamento dos dados. Elas fazem os acessos na base de dados, de acordo com a necessidade, a fim de atender à solicitação do usuário**.**

> A grande vantagem de se utilizar o padrão MVC é a separação de lógica e apresentação, sendo que isso favorece o trabalho em equipe. Um *designer* poderia trabalhar na apresentação, definindo o HTML, CSS, Flash, enquanto um *Database Administrator* (DBA), administrador de banco de dados, poderia trabalhar com o modelo e o outro programador poderia se concentrar nas regras de negócio inseridas no controlador. Dessa forma, qualquer mudança, por exemplo, na apresentação, teria pouco ou nenhum impacto nas demais camadas da aplicação. (MINETTO, 2008).

3.2 *FACADE*

O *Facade* é um padrão estrutural que tem como objetivo fornecer uma interface unificada para um conjunto de interfaces em um subsistema. Com isso, há um aumento no controle de acesso aos recursos internos da aplicação.

A ilustração 2, mostrada abaixo, ilustra duas estruturas de um sistema. A primeira, uma aplicação onde todas as solicitações são destinadas diretamente às classes modelos, sem a implementação do padrão *Facade.* A segunda, uma aplicação que implementa esse padrão. Todas as solicitações passam pela "fachada", que por sua vez distribui para a classe modelo correspondente à resposta.

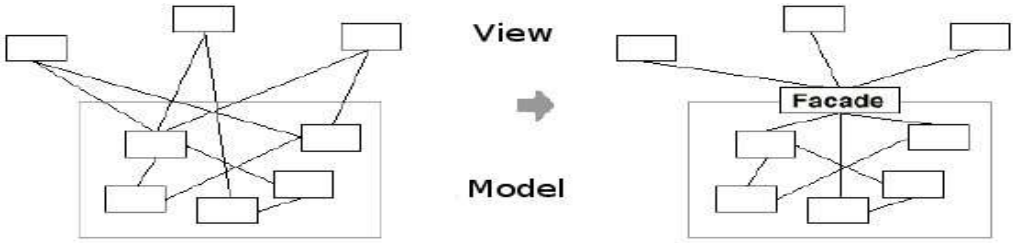

Ilustração 2: Padrão *Facade.*
Fonte: VIEIRA, 2003

Esse padrão faz parte do *Controller*, apresentado anteriormente junto ao MVC. Observe que a *View* continua a não saber como as classes atrás da fachada cumprem com suas responsabilidades.

Segundo Gamma (2005), seus principais benefícios são:

- Isolar os clientes dos componentes do subsistema, reduzindo o número de objetos com os quais o cliente tem que lidar;

- Promover um acoplamento fraco entre o subsistema e seus clientes. Isso permite variar os componentes do subsistema sem afetar os seus clientes;

- Não impedir as aplicações de utilizarem as classes do subsistema, caso necessitem.

Na listagem 5 é apresentado um *facade* implementado em Java.

```
public class FrontController extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
                    throws ServletException, IOException {

        String action = request.getParameter("action");
        Action actionObject=null;

        if(action==null || action.equals(""))
            response.sendRedirect("index.jsp");

        actionObject=ActionFactory.create(action);
        actionObject.execute(request, response);

    }

    protected void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

    protected void doPost(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
```

```
            processRequest(request, response);
        }
    }

    public interface Action {
        public void execute(HttpServletRequest request,
HttpServletResponse response) throws IOException;
    }

    public class ActionFactory {

        public static Action create(String action){

            Action actionObject= null;
            String nomeClasse = "action." + action + "Action";
            Class classe = null;
            Object object = null;

            try {
               classe = Class.forName(nomeClasse);
               object = classe.newInstance();
            } catch (Exception ex) {
               return null;
            }

            if(!(object instanceof Action))
                return null;

            actionObject = (Action) object;
            return actionObject;

        }
    }
```

Listagem 5 – Classes *FrontController* e *ActionFactory*, e interface *Action,* que compõem o DB Música.
Fonte: VIEIRA, 2009.

No primeiro instante, a requisição do usuário é recuperada da url e atribuída a uma variável chamada *action.* A partir daí, começa a essência da classe controladora, onde o código verifica qual a ação requisitada pela *view* a redireciona para a classe modelo que irá responder pela solicitação. A fachada provém uma interface única para acesso às classes de modelo do sistema, fornecendo segurança e organização da aplicação.

3.3 *DATA ACCESS OBJECT*

Data Access Object (DAO) é um padrão de projeto usado para realizar todas as operações que envolvem banco de dados. Ele faz a separação entre as regras de acesso aos dados e as regras de negócio.

Em uma aplicação que utiliza MVC, todas as funcionalidades que envolvem comandos de acesso a dados, tais como mapear objetos para tipos de dados SQL, executar comandos SQL ou obter conexões devem ser feitas por classes de DAO.

> Sempre que você precisa acessar um banco de dados que está mantendo seu modelo de objetos, é melhor empregar o padrão DAO. O padrão DAO fornece uma interface independente, no qual você pode usar para persistir objetos de dados. A ideia é colocar todas as funcionalidades encontradas no desenvolvimento de acesso e trabalho com dados em um só local, tornando simples sua manutenção. (GONÇALVES, 2007).

Com isso, os objetos de negócio podem usar a conexão e acesso aos dados sem conhecer os detalhes específicos de implementação. Além disso, como toda implementação relacionada a instruções SQL está contida na camada DAO e não no objeto, ficando mais fácil a manutenção de códigos.

A seguir há um exemplo desse padrão, onde os acessos à base de dados para manipulação das informações de compositores são feitas pela classe *CompositorDAO*.

```
public class CompositorDAO {

    private static CompositorDAO instance = new CompositorDAO();
    private CompositorDAO() {
    }

    public static CompositorDAO getInstance() {
        return instance;
    }

    public void save(Compositor compositor) throws SQLException,
ClassNotFoundException {
        Connection conn = null;
        Statement st  = null;

        try {
            conn = DatabaseLocator.getInstance().getConnection();
            st = conn.createStatement();
            st.execute("INSERT INTO "
                    + " compositor (nome, "
                    + "data_nascimento, "
                    + "naturalidade, "
                    + "nacionalidade)"
                    + " values ('" + compositor.getNome()
                    + "', '" + compositor.getDataNascimento()
                    + "', '" + compositor.getNaturalidade()
                    + "', '" + compositor.getNacionalidade() + "')");
        } catch(SQLException e) {
            throw e;
        } finally {
            closeResources(conn, st);
        }
    }

    public void closeResources(Connection conn, Statement st) {
        try {

            if(st!=null)
                st.close();
            if(conn!=null)
                conn.close();
            } catch(SQLException e) {
            }
    }
```

```
}
```

Listagem 6 – Trecho do código fonte da classe CompositorDAO, que compõe o DB Música.

Fonte: VIEIRA, 2009.

A fim de simplificar o código, entre os métodos de manipulação de dados, foi apresentado apenas o *save*, para incluir um compositor no banco de dados.

CONCLUSÃO

A utilização da programação orientada a objetos para o desenvolvimento de sistemas traz uma excelente oportunidade para a reutilização de trechos de códigos ou até módulos em outros sistemas. Levando em consideração que os prazos são cada vez menores para a conclusão do desenvolvimento de softwares, ela oferece uma vantagem para o desenvolvedor.

A linguagem Java traz diversos benefícios, como segurança, portabilidade, robustez, oferecendo suporte para o desenvolvimento de sistemas orientado a objetos, além de ser uma plataforma livre e gratuita.

Além dos benefícios dessas técnicas e tecnologias, pode-se melhorar ainda mais o desenvolvimento de sistemas concatenando Java, POO e o uso de padrões de projeto. Essa integração promove um vocabulário comum entre desenvolvedores e o uso de soluções comprovadas para determinados problemas de desenvolvimento.

A adoção desses conceitos facilita a organização e a reutilização de diversas partes de um sistema, evitando assim o retrabalho por parte dos analistas e desenvolvedores.

REFERÊNCIAS BIBLIOGRÁFICAS

ALMEIDA, Rodrigo Rebouças. de. ***Model-View-Controller*** **(MVC)**. Campina Grande: Universidade Federal de Campina Grande, [200-?]. Disponível em: <http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/arqu/mvc/mvc.htm>. Acesso em: 18 jul. 2011.

BEZERRA, Eduardo. **Princípios de Análise e Projetos de Sistemas com UML**. Rio de Janeiro: Campus, 2002.

CADENHEAD, Rogers.; LEMAY, Laura. **Aprenda Java 2 em 21 dias: *Professional Reference***. 3. ed. Rio de Janeiro: Campus, 2003.

DALL'OGLIO, Pablo**. PHP: Programando com Orientação a Objetos**. São Paulo: Novatec, 2008.

DEITEL, Harvey M.; DEITEL, Paul J.. **Java: como programar**. 6 ed. São Paulo: Pearson Prentice Hall, 2005

FOWLER, Martin. **Refatoração: aperfeiçoando o projeto de código existente**. Bookman, Porto Alegre, 2004.

FREEMAN, Eric. **Use a Cabeça: Padrões de Projeto**. São Paulo: Alta Books, 2005.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Padrões de Projeto: soluções reutilizáveis de softwares orientado a objetos**. São Paulo: Bookman, 2005.

GONÇALVES, Edson. **JSP, *Servlets*, *Java Serve Faces*, *Hibernate*, EJB3 *Persistence* e *Ajax***. Rio de Janeiro: Ciência Moderna, 2007.

HORSTMANN, Cay S.; CORNELL, Gary. ***Core* Java – Volume 1 - Fundamentos**. 8 ed. São Paulo: Pearson Prentice Hall, 2010.

MINETTO, Elton Luís. ***Frameworks* para desenvolvimento PHP**. São Paulo: Novatec, 2008.

PIMENTEL, A. R. **Modelos, Especificações, Modelos de Projeto**. Curitiba, 2010. Disponível em: <http://www.inf.ufpr.br/andrey/ci221/SOFTua11.pdf> Acesso em: 27 jun de 2011.

SINTES, Anthony. **Aprenda a Programação Orientada a Objetos em 21 dias**. São Paulo: Makron Books, 2002.

VIEIRA, Rafael Francelino Ferreira Mendes. **DB Música: Base de Dados para Aprendizado de Harmonia**. 47 f. Trabalho de Conclusão de Curso (Documentação de Software - Graduação em Sistemas de Informação). Centro de Ensino Superior de Juiz de Fora, Juiz de Fora, 2009.

VIEIRA, Maurício. **Anotações sobre o Padrão de Projeto Estrutural "*Facade*" (Fachada).** Bahia: Universidade Federal da Bahia, 2003. Disponível em: <http://wiki.dcc.ufba.br/Aside/NotasFacade>. Acesso em: 17 jul. 2011.