Projeto de Programas (PPR)
Guião Padrões de Projeto

Universidade do Estado de Santa Catarina - UDESC
Centro de Ciências Tecnológicas - CCT

Prof. **Jeferson Souza** (thejefecomp)
jeferson.souza@udesc.br

Objetivo

Este guião tem como objetivo apresentar alguns dos principais padrões de projeto utilizados no desenvolvimento de software. Cada padrão de projeto está acompanhado de uma descrição, exemplo(s) de implementação na linguagem de programação Java, e exemplo(s) de utilização. Neste guião, os seguintes padrões de projeto serão abordados:

1. *Factory*;
2. *Abstract Factory*;
3. *Builder*;
4. *Prototype*;
5. *Singleton*;
6. *Facade*;
7. *Data Access Object (DAO)*;

8. *Model View Controller (MVC).*

Os códigos fonte dos exemplos utilizados nesse guião estão disponíveis no github, no seguinte repositório: `https://github.com/thejefecomp/ppr`.

1 Padrões de Projeto "Criacionais"

Nesta seção são apresentados os padrões de projeto "criacionais", os quais tem o objetivo de especificar uma forma padronizada de criar entidades de software. Os seguintes padrões de projeto são "criacionais": padrão *Factory*; padrão *Abstract Factory*; padrão *Builder*; padrão *Prototype*; e o padrão *Singleton*.

1.1 Padrão de Projeto Factory

O padrão de projeto *Factory* permite a criação de instâncias de entidades de software de um dado tipo, que estejam organizadas em uma hierarquia de classes. Analogamente, imaginem uma fábrica de refrigeradores, a qual produz (não fiquem espantados!) refrigeradores. A fábrica é capaz de produzir diferentes tipos de refrigeradores, a depender da requisição que é feita no momento da produção. O padrão de projeto *Factory* tem o mesmo princípio de funcionamento: a "fábrica" de objetos cria uma instância de uma das subclasses que a "fábrica" consegue produzir, a depender de um dado parâmetro que determina qual instância que será fabricada. Caso a "fábrica" seja capaz de criar instâncias de somente uma classe, então nenhum parâmetro precisa ser informado no momento da criação.

1.1.1 Exemplo de Implementação

A classe *UserFactory* ilustra um exemplo de implementação do padrão de projeto *Factory*. Na linha 9 está declarado o método *newInstance*; esse método retorna uma nova instância da classe *User*, a qual é uma superclasse que possui as seguintes subclasses: *Administrator*, *Client*, *Manager*, e *Seller*.

A classe *UserFactory2* ilustra um segundo exemplo do padrão de projeto *Factory*. No exemplo ilustrado pela classe *UserFactory2* o tipo de usuário é definido pelo tipo genérico T. Logo, a "fábrica" especificada pela classe *UserFactory2* cria instâncias específicas de somente uma das subclasses *Administrator*, *Client*, *Manager*, e *Seller*.

Implementação 1: UserFactory.java

```
1  package org.jefecomp.patterns.factory;
2  import org.jefecomp.entity.Administrator;
3  import org.jefecomp.entity.Client;
4  import org.jefecomp.entity.Manager;
5  import org.jefecomp.entity.Seller;
6  import org.jefecomp.entity.User;
7  import org.jefecomp.entity.UserTypeEnum;
8  public class UserFactory{
9    public User newInstance(UserTypeEnum userType){
10     switch(userType){
11       case MANAGER: return new Manager();
12       case SELLER: return new Seller();
13       case ADMINISTRATOR: return new Administrator();
14       case CLIENT: return new Client();
15       default: return null;
16     }
17   }
18 }
```

1.1.2 Exemplo de Utilização

Nesta seção serão apresentados dois exemplos de utilização do padrão de projeto *Factory*, um exemplo para cada implementação apresentada anteriormente.

```
UserFactory factory = new UserFactory();
User user = factory.newInstance(UserTypeEnum.MANAGER);
```

Exemplo 1 - Criar uma instância de usuário da subclasse *Manager*

```
UserFactory2<Manager> factory = new
UserFactory2<>(UserTypeEnum.MANAGER);

Manager manager = factory.newInstance();
```

Exemplo 2 - Criar uma instância de usuário da subclasse *Manager*, utilizando uma "fábrica" parametrizada

Implementação 2: UserFactory2.java

```
1  package org.jefecomp.patterns.factory;
2  import org.jefecomp.entity.Administrator;
3  import org.jefecomp.entity.Client;
4  import org.jefecomp.entity.Manager;
5  import org.jefecomp.entity.Seller;
6  import org.jefecomp.entity.UserTypeEnum;
7  public class UserFactory2<T>{
8    private Class<?> clazz;
9    public UserFactory2(UserTypeEnum userType){
10     switch(userType){
11       case ADMINISTRATOR: this.clazz = Administrator.class;
12       break;
13       case CLIENT: this.clazz = Client.class;
14       break;
15       case MANAGER: this.clazz = Manager.class;
16       break;
17       case SELLER: this.clazz = Seller.class;
18       break;
19       default: this.clazz = null;
20     }
21   }
22   public T newInstance(){
23     if(this.clazz == null)
24       return null;
25     try {
26       return (T) this.clazz.newInstance();
27     }
28     catch(InstantiationException | IllegalAccessException e){
29       e.printStackTrace();
30       return null;
31     }
32   }
33 }
```

1.1.3 Exercício de Fixação

1. Implemente o padrão de projeto *Factory* de forma a permitir a criação de instâncias de uma dada classe de clientes de um sistema bancário único, tal como uma fábrica de instâncias da classe Pessoa_Fisica;

2. Implemente o padrão de projeto *Factory* com tipo parametrizado, de forma a permitir a criação de um tipo de classe de cliente, de acordo com o tipo da classe que for passada como parâmetro para a fábrica.

1.2 Padrão de Projeto Abstract Factory

O padrão de projeto *Abstract Factory* especifica um nível mais abstrato de criação de instâncias de tipos de entidades de software. Com o padrão *Abstract Factory* é possível criar instâncias de diferentes classes, as quais não necessitam fazer parte da mesma hierarquia. Analogamente, imaginem uma fábrica que produz não somente refrigeradores, mas também exaustores, e aparelhos de ar-condicionado. Cada um desses equipamentos precisa de um processo de produção específico, o que significa que o seu processo de produção é independente. O padrão de projeto *Abstract Factory* tem o mesmo princípio de funcionamento; cada instância de uma dada classe possui uma forma de ser criada, o que significa que de uma forma bastante abstrata pode-se dizer que o padrão de projeto *Abstract Factory* é visto como uma "fábrica" com mini "fábricas" internas.

1.2.1 Exemplo de Implementação

Na classe *AbstractGenericFactory* está ilustrada a implementação de uma "fábrica" abstrata e genérica, o que significa que o nível de abstração desta fábrica é tão elevado que a mesma pode criar instâncias de qualquer classe definida pela linguagem Java que não seja abstrata, e que possa ser instanciada com a utilização do construtor sem argumentos.

1.2.2 Exemplos de Utilização

Nesta seção serão apresentados exemplos de utilização da implementação do padrão de projeto *Abstract Factory* apresentada anteriormente.

```
String s = AbstractGenericFactory.newInstance("java.lang.String");
```

Exemplo 1 - Criar uma instância da classe *String*

```
Client c = AbstractGenericFactory.newInstance("org.jefecomp.entity.Client");
```

Exemplo 2 - Criar uma instância da classe *Client*

Implementação 3: AbstractGenericFactory.java

```
1  package org.jefecomp.patterns.factory;
2  public abstract class AbstractGenericFactory{
3    public static <T> T newInstance(String className) throws ClassNotFoundException {
4      Class<T> clazz = (Class<T>) Class.forName(className);
5      return newInstance(clazz);
6    }
7    public static <T> T newInstance(Class<T> clazz){
8      try{
9        return clazz.newInstance();
10     }
11     catch(InstantiationException | IllegalAccessException e){
12       e.printStackTrace();
13     }
14     return null;
15   }
16 }
```

1.2.3 Exercício de Fixação

1. Implemente o padrão de projeto *Abstract Factory* de forma a permitir a criação de instâncias de diferentes classes que fazem parte do modelo de dados do sistema bancário único, tais como instâncias de classes de cliente e classes de conta.

1.3 Padrão de Projeto Builder

O padrão de projeto *Builder* permite a criação de instâncias de classes que podem ter o seu estado interno customizado a depender da necessidade de sua utilização. Isso significa que o padrão de projeto *Builder* fornece formas de atribuir valores aos atributos de uma dada classe, criando portanto instâncias com estado interno customizado.

A implementação do padrão de projeto *Builder* pode ser realizada através da definição de uma classe *Builder*, a qual possui métodos para modificar os diferentes atributos cujo valor possa ser customizado. Além disso, um método chamado *build()* também é definido, o qual serve para criar uma instância de uma dada classe, logo após terem sido feitas todas as customizações requeridas para a sua criação.

1.3.1 Exemplo de Implementação

O exemplo de implementação do padrão de projeto *Builder* está dividido em duas classes: *AbstractUserBuilder* e *ClientBuilder*. A classe abstrata *AbstractUserBuilder* possui métodos responsáveis pela atribuição de valores dos atributos definidos na superclasse *User*, enquanto a classe *ClientBuilder* possui um método responsável pela atribuição do valor do atributo específico da classe *Client*. Além disso, como a classe abstrata *AbstractUserBuilder* é a superclasse dos "builders" associados as subclasses da superclasse *User*, essa classe também tem definido o método abstrato *build()*, o qual retorna um tipo genérico T associado ao tipo de usuário que será criado.

A classe *ClientBuilder* é um subclasse de *AbstractUserBuilder* e, então, define uma implementação específica do método *build()* para criar uma nova instância da classe *Client*.

1.3.2 Exemplo de Utilização

Nesta seção será apresentado um exemplo de utilização da implementação do padrão de projeto *Builder* apresentada anteriormente.

```
ClientBuilder builder = new ClientBuilder();
Client newClient = builder.addCreditPoints(0)
                          .addId(1L)
                          .addEmail("jefecomp.official@gmail.com")
                          .addCpf("00100023401")
                          .addName("Jeferson Souza")
                          .build();
```

Exemplo 1 - Criar uma instância da classe *Client*

1.3.3 Exercício de Fixação

1. Implemente o padrão de projeto *Builder* de forma a permitir a criação de instâncias de contas de um sistema bancário único, os quais são criados com atributos customizados, tal como um "builder" para criar instâncias da classe Conta, com o tipo poupança, e um saldo inicial de R$ 200,00.

Implementação 4: AbstractUserBuilder.java

```
1  package org.jefecomp.patterns.builder;
2  public abstract class AbstractUserBuilder<T>{
3    protected Long id;
4    protected String email;
5    protected String cpf;
6    protected String name;
7    protected char[] password;
8    public AbstractUserBuilder<T> addId(Long id){
9      this.id = id;
10     return this;
11   }
12   public AbstractUserBuilder<T> addEmail(String email){
13     this.email = email;
14     return this;
15   }
16   public AbstractUserBuilder<T> addCpf(String cpf){
17     this.cpf = cpf;
18     return this;
19   }
20   public AbstractUserBuilder<T> addName(String name){
21     this.name = name;
22     return this;
23   }
24   public AbstractUserBuilder<T> addPassword(char[] password){
25     this.password = password;
26     return this;
27   }
28   public abstract T build();
29 }
```

1.4 Padrão de Projeto Prototype

Em alguns casos não é necessário criar instâncias customizadas de classes, mas sim a mesma instância muitas vezes, repetidamente. Para realizar tal criação de forma padronizada existe o padrão de projeto *Prototype*. O padrão de projeto *Prototype* permite a clonagem de instâncias, diminuindo bastante o tempo de criação de novas instâncias de uma dada classe, e com o mesmo estado interno.

A implementação do padrão de projeto *Prototype* pode ser definida através da especificação de um método *clone()*, o qual permite fazer cópias de uma

Implementação 5: ClientBuilder.java

```
1  package org.jefecomp.patterns.builder;
2  import org.jefecomp.entity.Client;
3  public class ClientBuilder extends AbstractUserBuilder<Client>{
4    private Integer creditPoints;
5    public ClientBuilder addCreditPoints(Integer creditPoints){
6      this.creditPoints = creditPoints;
7      return this;
8    }
9    @Override
10   public Client build(){
11     Client client = new Client();
12     client.setId(this.id);
13     client.setEmail(this.email);
14     client.setCpf(this.cpf);
15     client.setName(this.name);
16     client.setPassword(this.password);
17     client.setCreditPoints(this.creditPoints);
18     this.id = null;
19     this.email = null;
20     this.cpf = null;
21     this.name = null;
22     this.password = null;
23     this.creditPoints = null;
24     return client;
25   }
26 }
```

mesma instância que podem, caso necessário e permitido, serem customizadas de acordo com a necessidade de utilização.

1.4.1 Exemplo de Implementação

O exemplo de implementação do padrão de projeto *Prototype* é ilustrado pela classe *CarPrototype*. O método *clone()* é o responsável por criar uma nova instância da classe *CarPrototype*, e copiar os valores dos atributos da instância corrente para essa nova instância, retornando-a como resultado da execução do método.

Implementação 6: CarPrototype.java

```
1  package org.jefecomp.patterns.prototype;
2  public class CarPrototype implements Cloneable{
3    private String type;
4    private String colour;
5    public final String getType() {
6      return type;
7    }
8    public final void setType(final String type) {
9      this.type = type;
10   }
11   public final String getColour() {
12     return colour;
13   }
14   public final void setColour(final String colour) {
15     this.colour = colour;
16   }
17   public CarPrototype clone(){
18     CarPrototype newInstance = new CarPrototype();
19     newInstance.type = type != null ? new String(type) : null;
20     newInstance.colour = colour != null ? new String(colour) : null;
21     return newInstance;
22   }
23 }
```

1.4.2 Exemplo de Utilização

Nesta seção será apresentado um exemplo de utilização da implementação do padrão de projeto *Prototype* apresentada anteriormente.

```
CarPrototype car = new CarPrototype();
car.setType("Sport");
car.setColour("Red");
CarPrototype carCloned = car.clone();
```

Exemplo 1 - Clonar uma instância da classe *CarPrototype*

1.4.3 Exercício de Fixação

1. Implemente o padrão de projeto *Prototype* de forma a permitir a clonagem de instâncias da classe Conta.

1.5 Padrão de Projeto Singleton

Quando é necessário permitir a criação de somente uma instância de uma dada classe, o padrão de projeto *Singleton* entra em ação. Com o padrão de projeto *Singleton* somente uma instância de uma dada classe é criada durante toda a execução de um dado software/componente. A partir do momento que a primeira instância de uma dada classe é criada, toda vez que é necessário utilizar uma instância dessa classe, a mesma instância criada anteriormente é utilizada.

1.5.1 Exemplo de Implementação

O exemplo de implementação do padrão de projeto *Singleton* tem alguns aspectos importantes. O primeiro está na definição do construtor "default" (sem argumentos) com visibilidade privada, ou seja, visível somente dentro da classe *UserFactorySingleton*. Como mais nenhum construtor é definido na classe *UserFactorySingleton*, definir o construtor "default" como privado evita que instâncias da classe *UserFactorySingleton* sejam criadas fora da própria classe. Isso garante que somente uma única instância da classe *UserFactorySingleton* será criada durante a execução de um dado software. O segundo está na definição do método *getInstance()*, o qual é thread-safe (pode ser utilizado com softwares que possuam linhas de execução concorrentes), e garante que somente uma instância da classe *UserFactoySingleton* é criada e armazenada dentro da própria classe, utiizando o atributo estático *instance* para essa finalidade.

1.5.2 Exemplos de Utilização

Nesta seção será apresentado um exemplo de utilização da implementação do padrão de projeto *Singleton* apresentada anteriormente.

```
UserFactorySingleton factory = UserFactorySingleton.getInstance();
```

Exemplo 1 - Criar uma única instância da classe *UserFactorySingleton*

1.5.3 Exercício de Fixação

1. Implemente o padrão de projeto *Singleton* de forma a permitir a criação de somente uma instância da classe BaseDeDados.

Implementação 7: UserFactorySingleton.java

```
1  package org.jefecomp.patterns.singleton;
2  import org.jefecomp.entity.Administrator;
3  import org.jefecomp.entity.Client;
4  import org.jefecomp.entity.Manager;
5  import org.jefecomp.entity.Seller;
6  import org.jefecomp.entity.User;
7  import org.jefecomp.entity.UserTypeEnum;
8  public class UserFactorySingleton{
9    private static UserFactorySingleton instance;
10   private UserFactorySingleton(){}
11   public static UserFactorySingleton getInstance(){
12     synchronized(UserFactorySingleton.class){
13       if(instance == null){
14         instance = new UserFactorySingleton();
15       }
16     }
17     return instance;
18     }
19   public User newInstance(UserTypeEnum userType){
20     switch(userType){
21       case MANAGER: return new Manager();
22       case SELLER: return new Seller();
23       case ADMINISTRATOR: return new Administrator();
24       case CLIENT: return new Client();
25       default: return null;
26     }
27   }
28 }
```

2 Padrões de Projeto “Estruturais”

Os padrões de projeto ditos “Estruturais” foram especificados para permitir que diferentes partes de um dado software possam ser combinadas para formar estruturas maiores. Nesta seção é apresentado o padrão de projeto *Façade*, que compõe os padrões de projeto ditos “Estruturais”.

2.1 Padrão Facade

O padrão de projeto *Facade* permite retirar a complexidade de uso de interfaces de diferentes partes do sistema, criando uma interface simple e única. Um exemplo claro de uso do padrão de projeto *Facade* pode ser encontrado na camada de acesso aos dados de um dado sistema, onde esses dados podem estar armazenados com o auxílio de diferentes tecnologias (tais como Bancos de Dados Relacionais, Arquivos, Nuvem, etc), mas o acesso a esses dados é realizado através da mesma interface, e da mesma forma (i.e. transparência de localização). Outro exemplo oriundo da camada de acesso aos dados é a especificação de uma interface para fazer operações *CRUD* em diferentes entidades do sistema, utilizando a mesma interface. O termo *CRUD* vem do inglês *Create, Retrieve, Update, and Delete* que são operações para criar, obter, atualizar e remover dados.

2.1.1 Exemplo de Implementação

Nesta seção é apresentado um exemplo de implementação do padrão de projeto *Facade*. Este exemplo apresenta uma implementação de uma camada de persistência de dados, a qual pode ser acessada através da fachada definida pela interface *DataFacade*, ilustrada na implementação 8. A interface *DataFacade* define operações genéricas do tipo *CRUD* para a manipulação dos dados. Acompanhando a interface *DataFacade* está a sua implementação *DataFacadeImpl*, a qual especifica uma implementação de uma camada de persistência de dados que utiliza internamente o padrão de projeto *DAO* para realizar manipulações nas diferentes entidades do modelo de dados.

Implementação 8: DataFacade.java

```
1 package org.jefecomp.patterns.facade;
2 public interface DataFacade{
3     <T> boolean delete(Long key, Class<T> clazz);
4     <T> T find(Long key, Class<T> clazz);
5     <T> boolean insert(Long key, T entity);
6     <T> boolean update(Long key, T entity);
7 }
```

Implementação 9: DataFacadeImpl.java

```
1  package org.jefecomp.patterns.facade;
2  import java.util.HashMap;
3  import java.util.Map;
4  import org.jefecomp.entity.*;
5  import org.jefecomp.patterns.dao.*;
6  public class DataFacadeImpl implements DataFacade{
7    private Map<Class<?>, Dao> daoMap;
8    private Dao getDao(Class<?> clazz){
9      return this.daoMap.get(clazz);
10   }
11   public DataFacadeImpl(){
12     this.daoMap = new HashMap();
13     this.daoMap.put(Client.class,ClientDao.getInstance());
14   }
15   public <T> boolean delete(Long key, Class<T> clazz){
16     return this.getDao(clazz).delete(key);
17   }
18   public <T> T find(Long key, Class<T> clazz){
19     return (T) this.getDao(clazz).find(key);
20   }
21   public <T> boolean insert(Long key, T entity){
22     return this.getDao(entity.getClass()).insert(key,entity);
23   }
24   public <T> boolean update(Long key, T entity){
25    return this.getDao(entity.getClass()).update(key,entity);
26   }
27 }
```

2.1.2 Exercício de Fixação

Implemente o padrão de projeto *Facade* de forma a permitir que o acesso aos dados do sistema seja realizado através de uma mesma interface, e de forma que operações *CRUD* de diferentes tipos de dados sejam realizadas com a mesma interface.

3 Padrões de Projeto Associados à Persistência de Dados

3.1 Padrão Data Access Object (DAO)

O padrão de projeto *Data Access Object (DAO)* permite a criação de classes para acesso aos dados, de forma a que os dados sejam manipulados através de uma interface de serviço com operações que permitem a inserção, atualização, remoção, e busca de entidades do modelo de dados. No padrão de projeto DAO cada entidade, e/ou hierarquia de entidades, possui uma classe associada para acesso aos dados. A seguir é apresentado um exemplo de implementação do padrão de projeto DAO.

3.1.1 Exemplo de Implementação

No exemplo apresentado, a interface DAO, apresentada na implementação 10, define as operações de inserção, aualização, remoção, e busca de entidades de uma dada classe de entidades. É possível perceber que a interface DAO é parametrizada, o que permite a criação de implementações de DAO com tipos específicos. Neste exemplo foi criado um DAO para a entidade Cliente, o qual é especificado pela classe ClienteDAO apresentada na implementação 11. Neste exemplo a base de dados é acessada por meio da interface Database e da classe DatabaseImpl,a qual é uma implementação super simples de uma base de dados em memória que utiliza mapas para armazenar os dados.

Implementação 10: Dao.java

```
1  package org.jefecomp.patterns.dao;
2  import java.util.List;
3  import java.util.Map;
4  public interface Dao<T> {
5     boolean delete(Long key);
6     boolean delete(Map<String,Object> attributeMap);
7     <K> T find(Long key);
8     List<T> find(Map<String,Object> attributeMap);
9     boolean insert(Long key, T entity);
10    boolean update(Long key, T entity);
11 }
```

Implementação 11: ClientDao.java

```
1  package org.jefecomp.patterns.dao;
2  import java.util.List;
3  import java.util.Map;
4  import org.jefecomp.database.Database;
5  import org.jefecomp.database.DatabaseImpl;
6  import org.jefecomp.entity.Client;
7  public class ClientDao implements Dao<Client>{
8    private static ClientDao instance;
9    private Database database;
10   private ClientDao(){
11     this.database = DatabaseImpl.getInstance();
12   }
13   public static ClientDao getInstance(){
14     synchronized(ClientDao.class){
15       if(instance == null){
16         instance = new ClientDao();
17       }
18     }
19     return instance;
20   }
21   public boolean delete(Long key){
22     return this.database.delete(key);
23   }
24   public boolean delete(Map<String,Object> attributeMap){
25     return false;
26   }
27   public Client find(Long key){
28     return this.database.find(key);
29   }
30   public List<Client> find(Map<String,Object> attributeMap){
31     return null;
32   }
33   public boolean insert(Long key, Client entity){
34     return this.database.insert(key,entity);
35   }
36   public boolean update(Long key, Client entity){
37     return this.database.update(key,entity);
38   }
39 }
```

3.1.2 Exercício de Fixação

Implemente o padrão de projeto *DAO* de forma a permitir que o acesso aos dados seja realizado através de "DAOs".

4 Padrões de Projeto Associados à Arquitetura de Software

4.1 Padrão Model View Controller (MVC)

O padrão *Model View Controller (MVC)* é uma padrão de projeto que permite a separação clara entre a interface de utilização do usuário e o modelo de dados. No padrão de projeto *MVC* um software é divido em três camadas: modelo (*Model*, controle (*Controller*, e visão (*View*). A camada do modelo é responsável por fornecer uma interface responsável pela manipulação dos dos dados, de forma independente de como esses dados serão apresentados. Para tal, a implementação da camada modelo não só pode como deve utilizar outros padrões de projeto na sua implementação, tais como os padrões de projeto *Facade* e *DAO*. A camada controle é um elo intermediário entre a camada modelo e a camada visão, estabelecendo a comunicação entre essas duas camadas. É na camada controle que o mapeamento entre os dados fornecidos pelo usuário e as entidades do modelo de dados. Já a camada visão é responsável por fornecer classes que permitam a utilização do software por um dado usuário. É na camada de visão que toda a interface gráfica é definida, seja ela implementada com componentes visuais apelativos ou somente dentro de um console.

Esse padrão é fundamental para atingir uma alta modularidade e um alto desacoplamento do sistema, já que poermite uma separação clara dos diferentes domínios a qual uma aplicação deve ser segmentada.

4.1.1 Exemplo de Implementação

Esta seção apresenta um exemplo de implementação do padrão de projeto *MVC*, o qual define uma camada de visão para cadastrar um cliente, e as suas respectivas camadas de controle e modelo.

Implementação 12: OperationEnum.java

```
1 package org.jefecomp.patterns.mvc.controller;
2 public enum OperationEnum{
3     ADD_CLIENT,
4     ADD_SHOES,
5     DEL_CLIENT,
6     DEL_SHOES,
7     FIND_CLIENT,
8     SELL_PRODUCT;
9 }
```

Implementação 13: Controller.java

```
1 package org.jefecomp.patterns.mvc.controller;
2 public interface Controller{
3     Object execute(OperationEnum operation, Object ... data);
4 }
```

4.1.2 Exercício de Fixação

Implemente o padrão de projeto *MVC* de forma a separar a visão (como os dados são visualizados) dos dados propriamente ditos, implementando as três camadas (modelo, visão, e controle) de forma a permitir que o controle realize a comunicação entre a visão e o modelo de dados.

Implementação 14: ControllerImpl.java

```
1  package org.jefecomp.patterns.mvc.controller;
2  import java.util.HashMap;
3  import java.util.List;
4  import java.util.Map;
5  import org.jefecomp.entity.Client;
6  import org.jefecomp.entity.vo.ClientVO;
7  import org.jefecomp.patterns.facade.DataFacade;
8  import org.jefecomp.patterns.facade.DataFacadeImpl;
9  public class ControllerImpl implements Controller{
10   private Map<Class<?>, Class<?» dataBindingMap;
11   private DataFacade dataFacade;
12   private Client map(ClientVO source){
13     Client destination = new Client();
14     destination.setCreditPoints(source.getCreditPoints());
15     destination.setCpf(source.getCpf());
16     destination.setEmail(source.getEmail());
17     destination.setId(source.getId());
18     destination.setName(source.getName());
19     destination.setPassword(source.getPassword());
20     return destination;
21   }
22   public ControllerImpl(){
23     this.dataFacade = new DataFacadeImpl();
24     this.dataBindingMap = new HashMap<>();
25     this.dataBindingMap.put(ClientVO.class,Client.class);
26   }
27   public Object execute(OperationEnum operation, Object ... data){
28     switch(operation){
29       case ADD_CLIENT:
30         for(Object entity : data){
31           Client client = this.map((ClientVO) entity);
32           this.dataFacade.insert(client.getId(), client);
33         }
34         return true;
35         case ADD_SHOES:
36         case DEL_CLIENT:
37         case DEL_SHOES:
38         case FIND_CLIENT:
39         default:
40       }
41     return null;
42   }
43 }
```

Implementação 15: AppUI.java

```
1  package org.jefecomp.patterns.mvc.view;
2  import java.util.Scanner;
3  import org.jefecomp.entity.vo.ClientVO;
4  import org.jefecomp.patterns.mvc.controller.Controller;
5  import org.jefecomp.patterns.mvc.controller.ControllerImpl;
6  import org.jefecomp.patterns.mvc.controller.OperationEnum;
7  public class AppUI{
8    private Controller controller;
9  2mm]
10   private Scanner scanner;
11   private Long clientId_Generated;
12   public AppUI(){
13     this.controller = new ControllerImpl();
14     this.scanner = new Scanner(System.in);
15     this.clientId_Generated = 1L;
16   }
17   public boolean addClient(){
18     ClientVO client = new ClientVO();
19     client.setId(this.clientId_Generated++);
20     String input = null;
21     System.out.println("Digite o nome do cliente: ");
22     client.setName(this.scanner.next());
23     System.out.println("Digite o email do cliente: ");
24     client.setCpf(this.scanner.next());
25     System.out.println("Digite o cpf do cliente: ");
26     client.setCpf(this.scanner.next());
27     return (Boolean) this.controller.execute(OperationEnum.ADD_CLIENT, client);
28   }
29   public boolean menu(){
30     System.out.println("***********Examplo de APP MVC *************");
31     System.out.println("Digite a opcao desejada:");
32     System.out.println("1 - Cadastrar Cliente");
33     System.out.println("2 - Sair");
34     System.out.println("***********Examplo de APP MVC *************");
35     int option = scanner.nextInt();
36     if(option == 1){
37       Boolean result = this.addClient();
38       if(result){
39         System.out.println("Cliente cadastrado com sucesso!");
40       }
41       return true;
42     }
43     this.scanner.close();
44     return false;
45   }
46 }
```