

INSTITUTO FEDERAL
ESPIRITO SANTO

PADRÕES DE PROJETO

PADRÕES DE PROJETO

- O objetivo de um *design pattern* é registrar uma experiência no projeto de software OO, na forma de um padrão passível de ser efetivamente utilizado por projetistas.
- Um projetista familiarizado com padrões de projeto pode aplicá-los diretamente a problemas de projeto.
- Uma vez que um padrão é aplicado, muitas decisões de projeto decorrem automaticamente.

PADRÕES DE PROJETO

- Busca de um baixo acoplamento;
 - Grau de dependência entre dois artefatos;
 - Ex. classes, subsistemas
- Busca de uma alta coesão;
 - princípio da responsabilidade única;
 - uma classe deve ter apenas uma única responsabilidade e realizá-la de maneira satisfatória;

Problemas de Coesão!

```
public class Angu {  
  
    public String funcionario;  
    private float perimetro;  
    public float nota_bimestral;  
    private float valor_mensal;  
  
    public static int calculaDiametro(float raio) {  
        // ...  
    }  
  
    public static int calcularMedia(Vector numeros) {  
        // ...  
    }  
  
    public static outputStream abreArquivo(String nomeArquivo) {  
        // ...  
    }  
  
}
```

PADRÕES DE PROJETO

	Propósito		
Escopo	Criativo	Estrutural	Comportamental
Classe	Método-Fábrica	Adaptador (classe)	Interpretador Método Modelo
Objeto	Construtor Fábrica Abstrata Protótipo Singular	Adaptador (objeto) Composto Decorador Fachada Peso-Mosca Ponte Procurador	Cadeia de Responsabilidade Comando Iterador Mediador Memorial Observador Estado Estratégia Visitador

PADRÕES DE PROJETO

- **Padrão Criativo:** abstrai o processo de instanciação (criação) de objetos, ajudando a tornar um sistema independente de como seus objetos são criados, compostos e representados.

PADRÕES DE PROJETO

- **Padrão Estrutural:** diz respeito a como classes e objetos são compostos para formar estruturas maiores.

$H + H + O \rightarrow$



PADRÕES DE PROJETO

- **Padrão Comportamental:** diz respeito a algoritmos e a atribuição de responsabilidades e comportamentos entre objetos.



PADRÕES DE PROJETO

- Padrões Criativos:

- **Método Fabrica:** define uma interface para criar um objeto, mas “deixa” a decisão de instanciação para as subclasses;
- **Fábrica Abstrata:** prover uma interface para criar uma família de objetos sem especificar as classes concretas;
- **Builder:** permite criar diversos objetos semelhantes, utilizando o mesmo algoritmo;
- **Protótipo:** especifica os tipos de objetos que serão criados através de protótipos e cria um novo objetivo copiando um protótipo;
- **Singleton:** garante que apenas um objeto será criado e prover um único ponto de acesso para a esse objeto;

PADRÕES CRIATIVOS

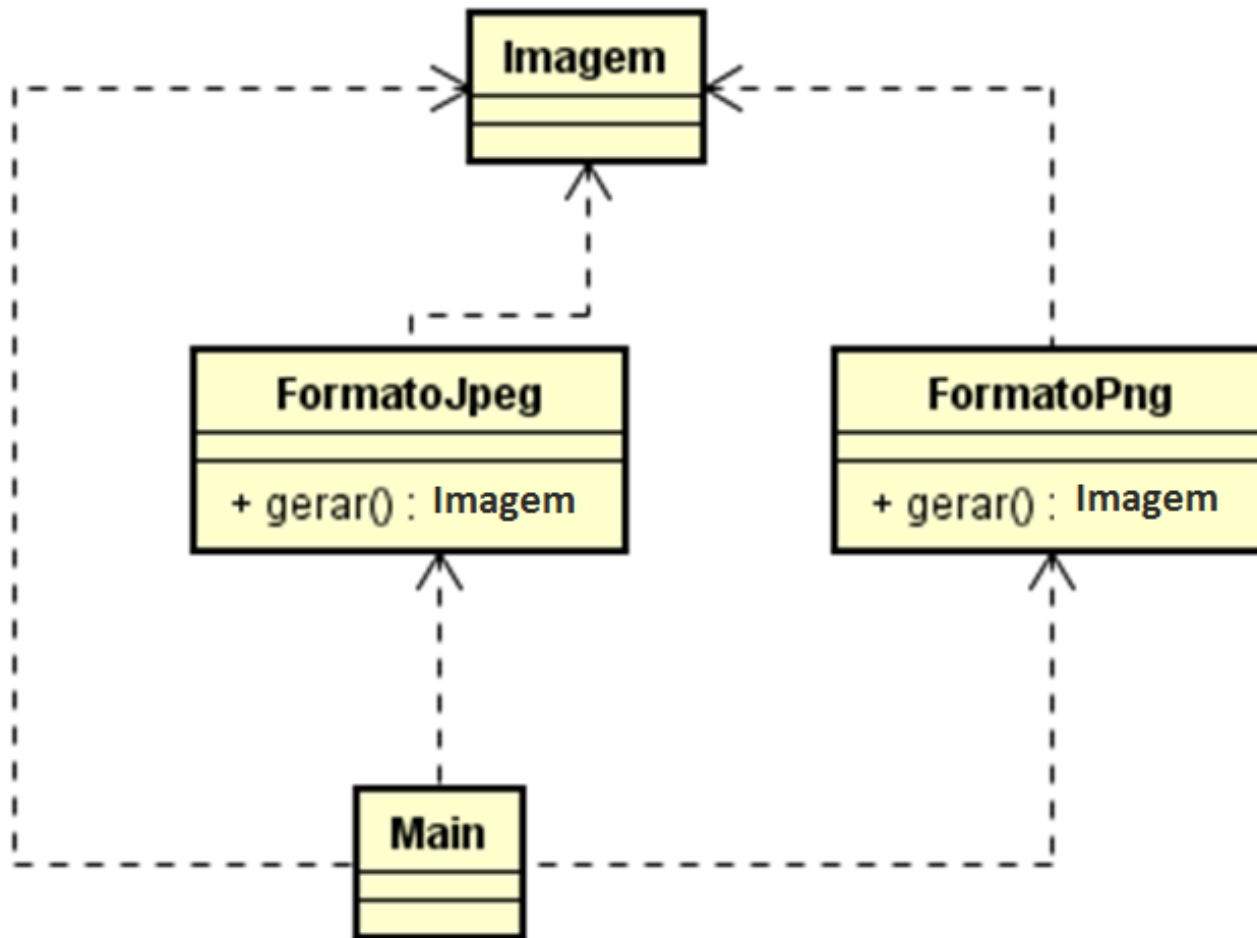
Método Fábrica

Exemplo - Sem padrão

```
public class FormatoJpeg {  
    public FormatoJpeg () {  
    }  
    public Imagem gerar(Imagem imagem) {  
        System.out.println("gera jpeg ");  
        return imagem;  
    }  
}
```

```
public class FormatoPng {  
  
    public FormatoPng() {  
    }  
    public Imagem gerar(Imagem imagem) {  
        System.out.println("gera png");  
        return imagem;  
    }  
}
```

Exemplo - Sem padrão



PADRÕES DE PROJETO

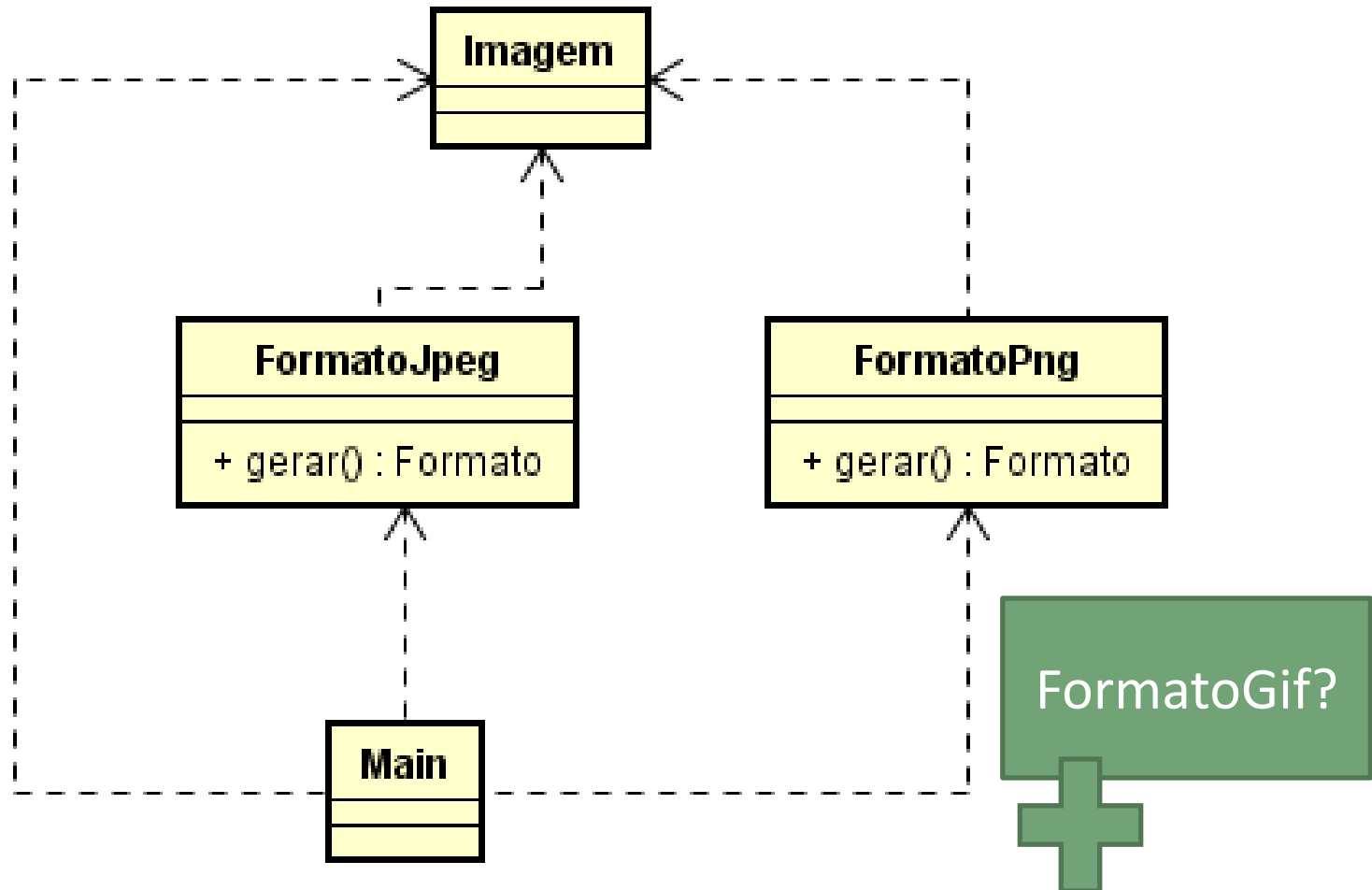
Método Fábrica:

- **Propósito:** definir uma interface para criar um objeto, mas deixar as subclasses decidirem que classe instanciar. O *método fábrica* permite adiar a instanciação para as subclasses.

Exemplo - Sem padrão

```
public class Main{  
    public void static main(String[] args){  
        Imagem img = Imgem.readImagem();  
        FormatoJpeg jpeg = new FormatoJpeg();  
        Imagem novalmagem = jpeg.gerar(img);  
  
        FormatoPng png = new FormatoPng();  
        Imagem novalmagem = png.gerar(img);  
    }  
}
```

Método Fábrica

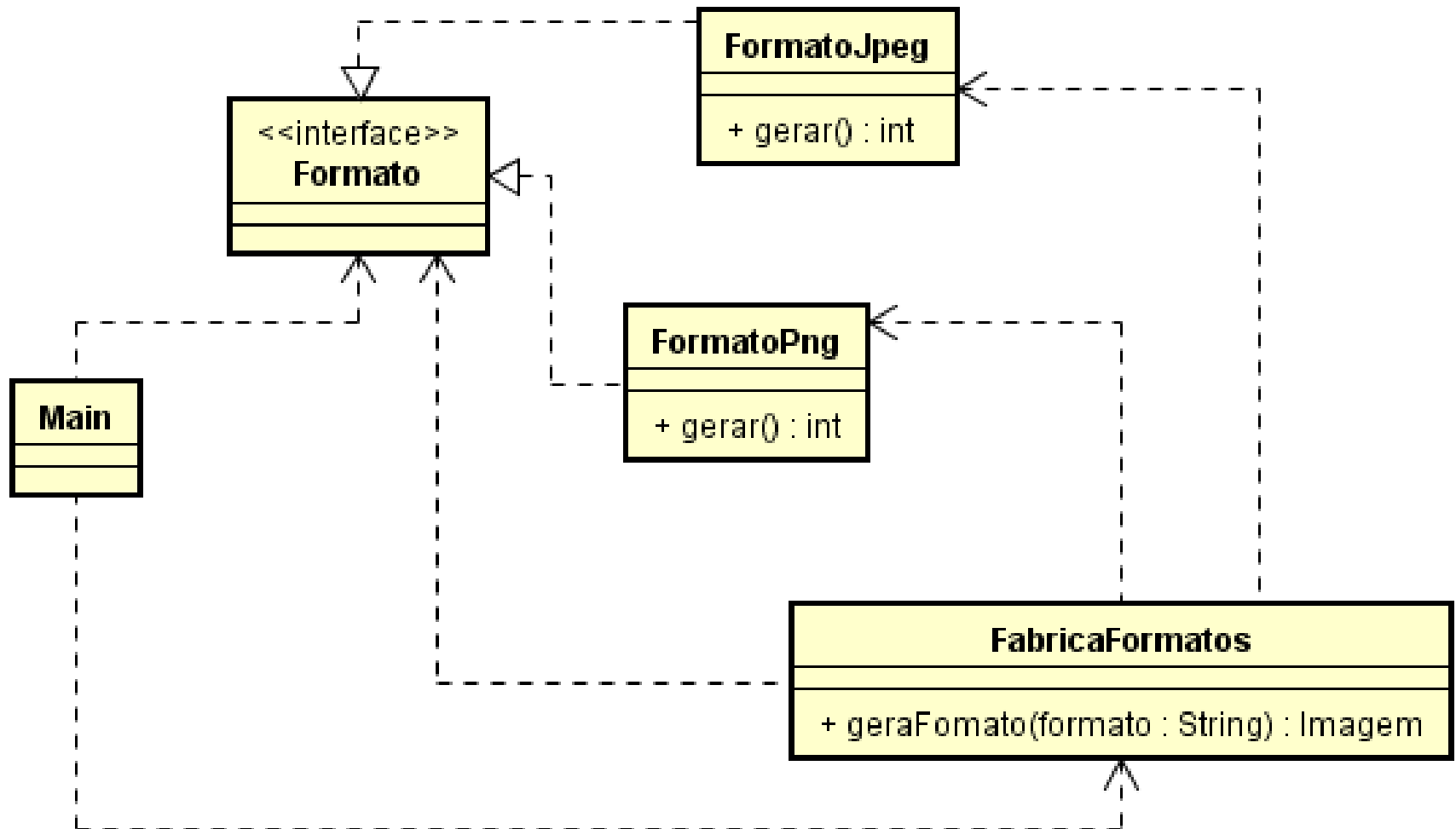


Exemplo - Sem padrão

```
public class Main{  
  
    public void static main(String[] args){  
        Imagem img = Imagem.readImagem();  
        FormatoJpeg jpeg = new FormatoJpeg();  
        Imagem novalmagem = jpeg.gerar(img);  
  
        FormatoPng png = new FormatoPng();  
        Imagem novalmagem = png.gerar(img);  
  
        FormatoGif gif = new FormatoGif();  
        Imagem novalmagem = gif.gerar(img);  
    }  
}
```

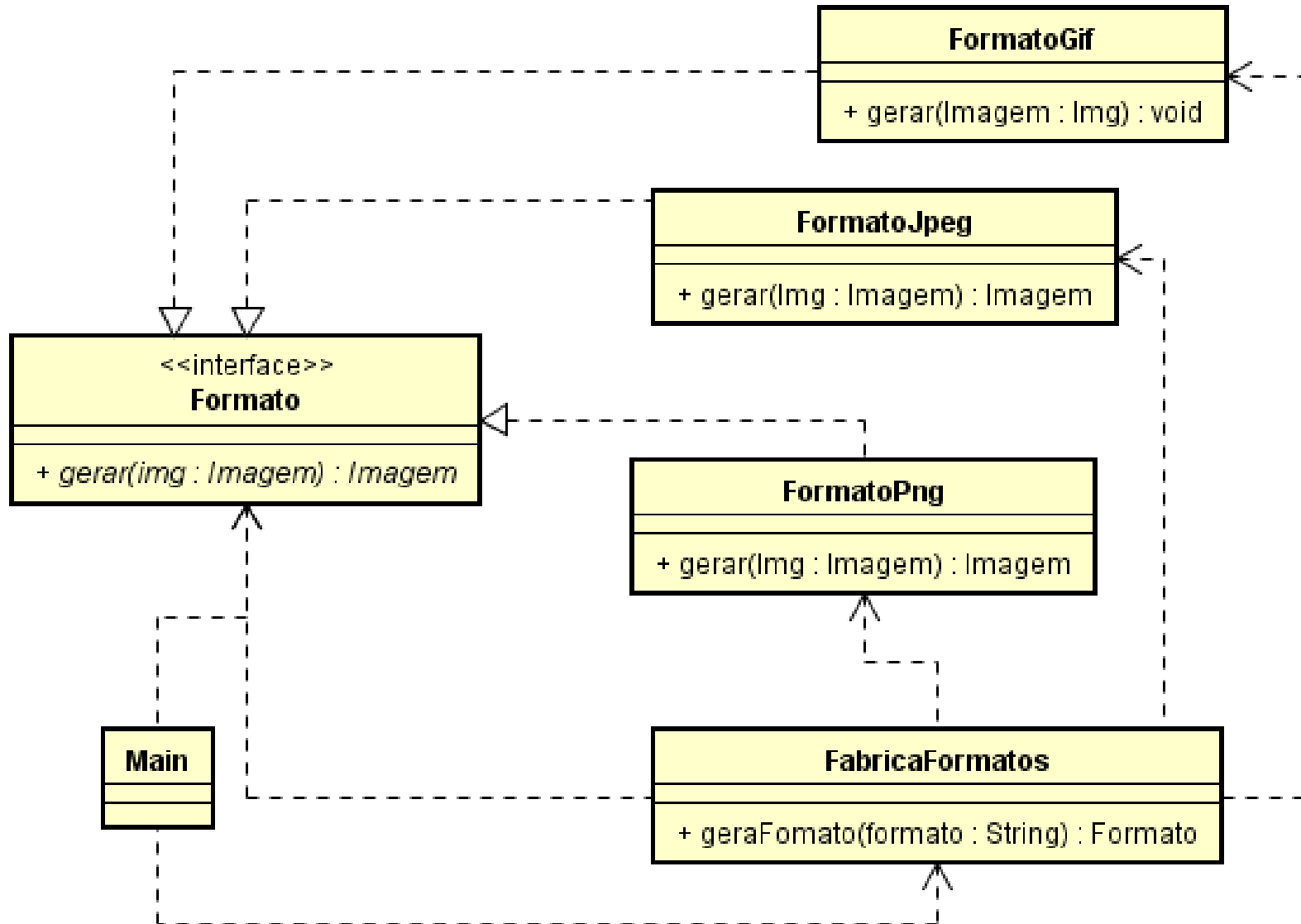
Preciso importar a classe FormatoGif

Método Fábrica



Método Fábrica

Adicionado!



Método Fábrica

```
public class FabricaFormatos{  
    public static Formato gerarFormato(String formato){  
        Formato gerador;  
        if(formato.equalsIgnoreCase("jpeg")){  
            gerador = new FormatoJpeg();  
        }else if(formato.equalsIgnoreCase("png")){  
            gerador = new FormatoPng();  
        }  
        return gerador;  
    }  
}
```

Método Fábrica

```
public class FabricaFormatos{  
    public static Formato gerarFormato(String formato){  
        Formato gerador;  
        if(formato.equalsIgnoreCase("jpeg"){  
            gerador = new FormatoJpeg();  
        }else if(formato.equalsIgnoreCase("png"){  
            gerador = new FormatoPng();  
        }  
        else if(formato.equalsIgnoreCase("gif"){  
            gerador = new FormatoGif();  
        }  
        return gerador;  
    }  
}
```

FormatoGif

Exemplo padrões Criativos

```
public class Main{  
    public void static main(String[] args){  
        Imagem img = Imagem.readImagem();  
        Formato formatador = FabricaFormatos. gerarFormato("jpeg");  
        Imagem img = formatador .gerar(img );  
    }  
}
```

Posso trocar por gif

Não preciso importar nada no caso de criar uma classe FormatoGif.

Não preciso saber como a classe FormatoGif, FormatoPng ou FormatoJpeg são criadas.

Método fábrica com Java Reflection

```
public class FabricaFormato {  
    public static Formato geraFormato(String classe){  
        Formato formato;  
        Object classeReflection = null;  
        try {  
            classeReflection = Class.forName(classe).newInstance();  
        } catch (InstantiationException e) {  
            e.printStackTrace();  
        } catch (IllegalAccessException e) {  
            e.printStackTrace();  
        } catch (ClassNotFoundException e) {  
            e.printStackTrace();  
        }  
        formato = (Formato) classeReflection;  
        return formato;  
    }  
}
```

Método fábrica com Java Reflection

```
package metodoestatico.fabrica;
```

```
public class MetodoFabrica {
```

```
    public static void main(String[] args) {
```

```
        Formato formato;
```

```
        formato = FabricaFormato.geraFormato("metodoestatico.fabrica.FormatoPng");
```

```
        formato.gera();
```

```
        formato = FabricaFormato.geraFormato("metodoestatico.fabrica.FormatoJpeg");
```

```
        formato.gera();
```

```
    }
```

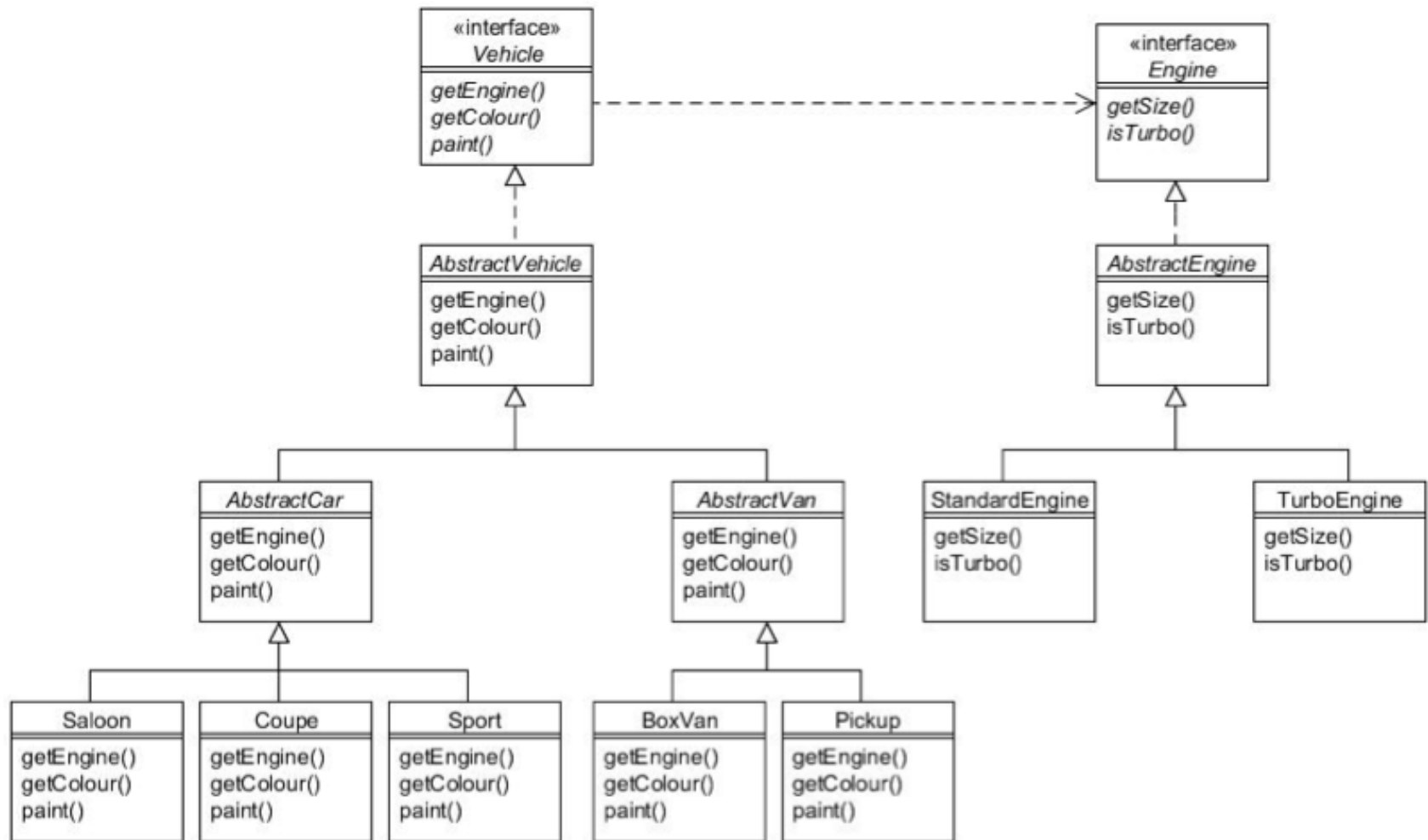
```
}
```

Exercício

CONTEXTO UTILIZADO - Exemplo

- Desenvolver um sistema para a Motores SA;
- A Motores SA fabrica:
 - Carros;
 - Vans;
 - Motores dos veículos;

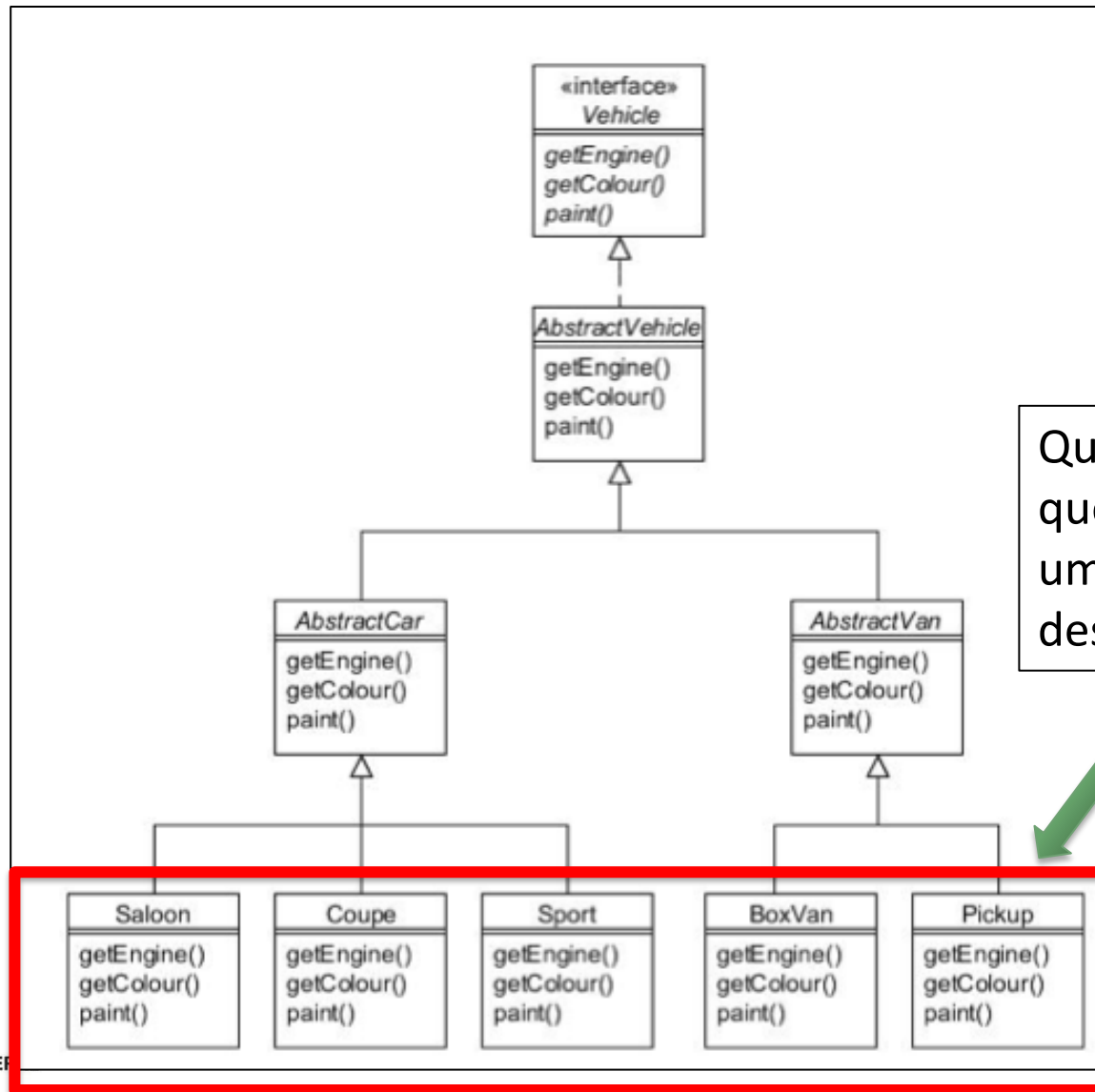
DIAGRAMA DE CLASSES



PADRÕES DE PROJETO

- Método Fábrica:
 - Problema:
 - Precisamos instanciar um tipo particular de veículo, por exemplo Coupe com um certa frequência;
 - Quero que uma classe seja responsável por gerenciar a instanciação;
 - Como faremos isso?

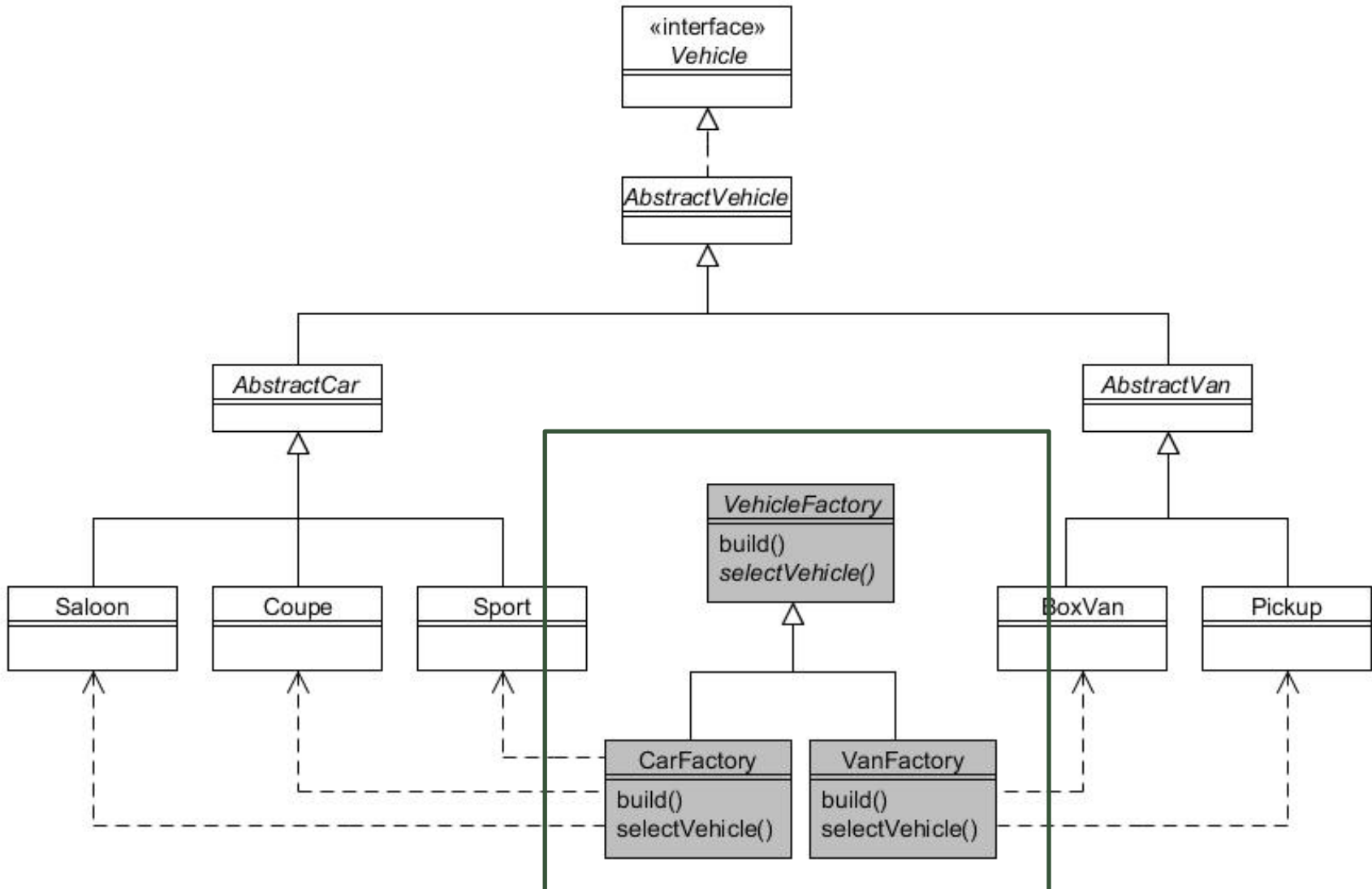
PADRÕES DE PROJETO



Quero um método
que retorne
uma instância
dessas classes!!



PADRÕES DE PROJETO



PADRÕES DE PROJETO

Método Fábrica:

```
public abstract class VehicleFactory {  
    public enum DrivingStyle {ECONOMICAL, MIDRANGE, POWERFUL};  
  
    public Vehicle build(DrivingStyle style, Vehicle.Colour colour) {  
        Vehicle v = selectVehicle(style);  
        v.paint(colour);  
        return v;  
    }  
  
    // This is the "factory method"  
    protected abstract Vehicle selectVehicle(DrivingStyle style);  
}
```

PADRÕES DE PROJETO

Fábrica de Van!

- Método Fábrica:



```
public class VanFactory extends VehicleFactory {  
    protected Vehicle selectVehicle(DrivingStyle style) {  
        if ((style == DrivingStyle.ECONOMICAL) ||  
            (style == DrivingStyle.MIDRANGE)) {  
            return new Pickup(new StandardEngine(2200));  
        }  
        else {  
            return new BoxVan(new TurboEngine(2500));  
        }  
    }  
}
```

PADRÕES DE PROJETO

Fábrica de Carros!

- Método Fábrica:

```
public class CarFactory extends VehicleFactory {  
    protected Vehicle selectVehicle(DrivingStyle style) {  
        if (style == DrivingStyle.ECONOMICAL) {  
            return new Saloon(new StandardEngine(1300));  
        } else if (style == DrivingStyle.MIDRANGE) {  
            return new Coupe(new StandardEngine(1600));  
        } else {  
            return new Sport(new TurboEngine(2000));  
        }  
    }  
}
```


PADRÕES DE PROJETO

- Método Fábrica:

```
// I want an economical car, coloured blue...
VehicleFactory carFactory = new CarFactory();
Vehicle car = carFactory.build(VehicleFactory.DrivingStyle.ECONOMICAL, Vehicle.Colour.BLUE);
System.out.println(car);

// I am a "white van man"...
VehicleFactory vanFactory = new VanFactory();
Vehicle van = vanFactory.build(VehicleFactory.DrivingStyle.POWERFUL, Vehicle.Colour.WHITE);
System.out.println(van);
```

PADRÕES DE PROJETO

- Método Fábrica:
 - Existe uma variação do método fábrica chamado: **método fábrica estático.**
 - No uso dela não é necessário instanciar um classe do tipo fábrica;

PADRÕES DE PROJETO

- Método Fábrica:

```
public enum Category {CAR, VAN};
```

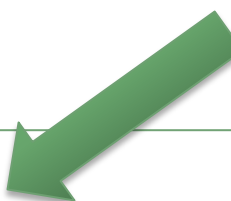
```
public static Vehicle make(Category category, DrivingStyle style, Vehicle.Colour colour) {  
    VehicleFactory factory = null;  
  
    if (category == Category.CAR) {  
        factory = new CarFactory();  
  
    } else {  
        factory = new VanFactory();  
    }  
  
    return factory.build(style, colour);  
}
```



PADRÕES DE PROJETO

Utilizando método fábrica estático!


- Método Fábrica Estático:



```
// Create a red sports car...
Vehicle sporty = VehicleFactory.make(VehicleFactory.Category.CAR, VehicleFactory.DrivingStyle.POWERFUL, Colour.RED);
System.out.println(sporty);
```

```
// I want an economical car, coloured blue...
VehicleFactory carFactory = new CarFactory();
Vehicle car = carFactory.build(VehicleFactory.DrivingStyle.ECONOMICAL, Vehicle.Colour.BLUE);
System.out.println(car);

// I am a "white van man"...
VehicleFactory vanFactory = new VanFactory();
Vehicle van = vanFactory.build(VehicleFactory.DrivingStyle.POWERFUL, Vehicle.Colour.WHITE);
System.out.println(van);
```

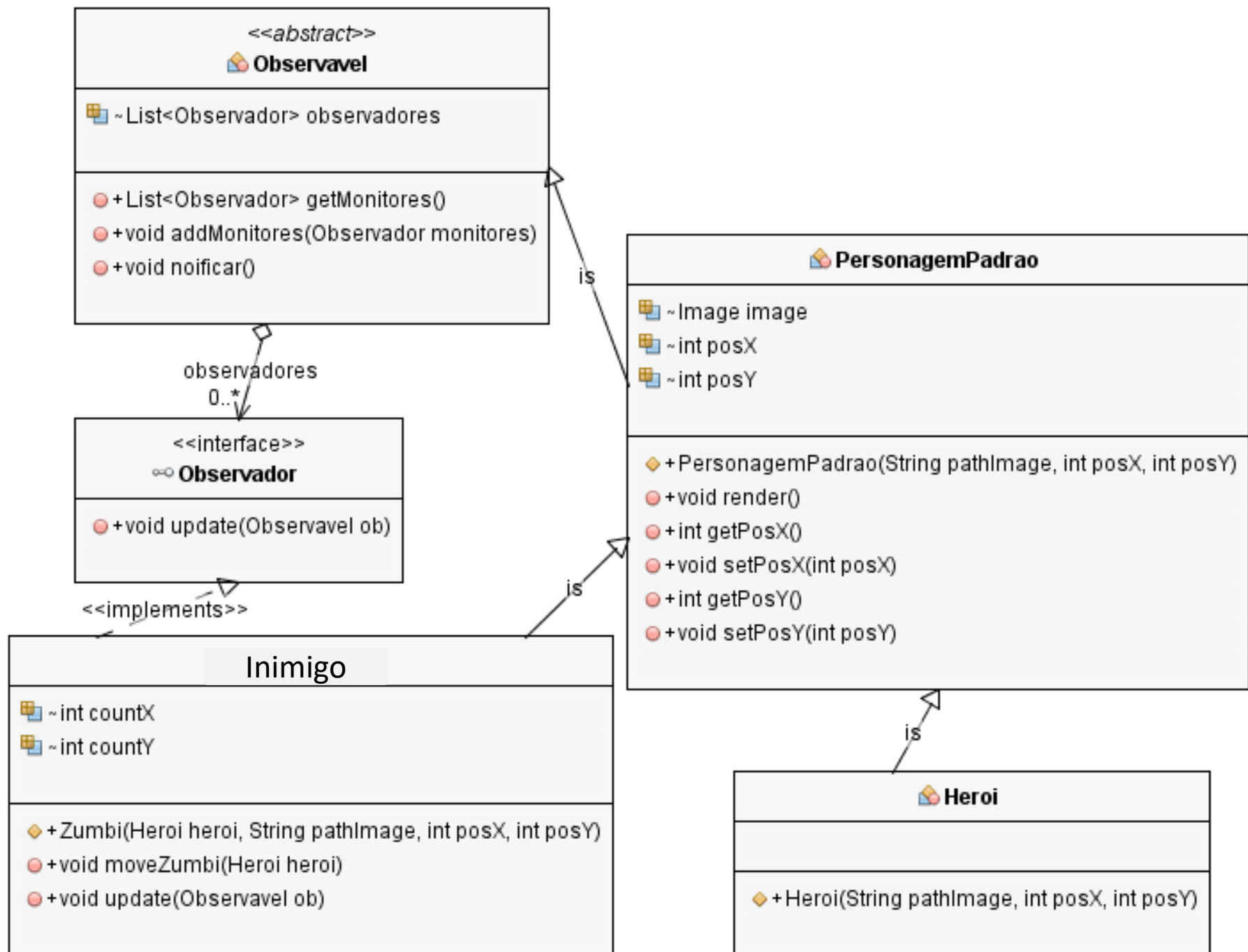


Utilizando método fábrica normal!

FPS: 2551

Time : 1





Classe SimpleSlickGame

```
public class SimpleSlickGame extends BasicGame {  
    public SimpleSlickGame(String gamename) {  
        super(gamename);  
    }
```

SimpleSlickGame
herda os
comportamentos de
BasicGame

```
    public void init(GameContainer gc) throws SlickException { ... }
```

```
    public void render(GameContainer gc, Graphics g) throws SlickException { ... }
```

```
    public void update(GameContainer gc, int i) throws SlickException { ... }
```

```
    public static void main(String[] args) { ... }
```

Esse método é
chamado em
intervalos de tempos
muito curtos como
0.01 segundos

Interfaces e Abstrações

Interface: Contrato que uma classe deve assinar quando quer utilizar

```
public interface Observador {  
  
    public void update(Observavel ob);  
  
}
```

A interface não implementa nada ela só organiza como deve ser implementado


```
public class Inimigo extends PersonagemPadrao implements Observador{
```

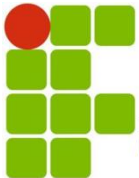
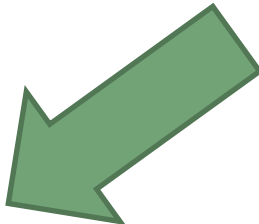
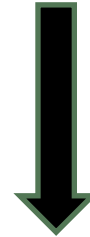
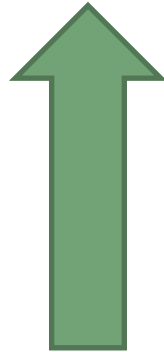
```
    public Inimigo(String pathImage, int posX, int posY) {  
        super(pathImage, posX, posY);
```

```
    }  
  
    @Override
```

```
    public void update(Observavel ob) {  
        moveInimigo((Herói) ob);
```

```
    }  
}
```

Preciso implementar pois na declaração da classe eu coloque implements Observador



INSTITUTO FEDERAL
ESPIRITO SANTO



GAME
OVER



```
public class Inimigo extends PersonagemPadrao implements Observador{
```

```
    public Inimigo(String pathImage, int posX, int posY) {  
        super(pathImage, posX, posY);  
    }
```

```
@Override
```

```
public void update(Observavel ob) {  
    moveInimigo((Heroi) ob);  
}
```

```
public void moveInimigo(Heroi heroi){
```

```
    if(this.posY >= 400){  
        this.posY = -20;  
        this.posX = ThreadLocalRandom.current().nextInt(0, 640);  
    }  
    else{  
        this.posY = this.posY + 10;  
    }
```

```
}
```

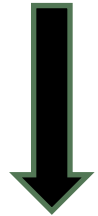
```
}
```

Adicionar

Quando a posição do mostro chegar ao final da tela reiniciar o y e criar um novo X

X randômico entre 0 e 640

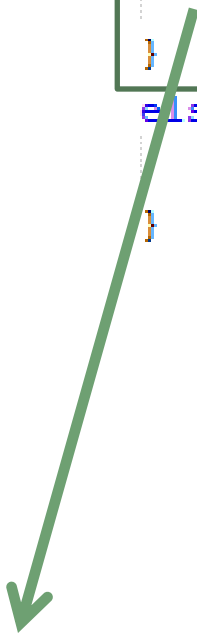
Nova posição Y



```
if(this.posY >= 400){  
    this.posY = -20;  
    this.posX = ThreadLocalRandom.current().nextInt(0, 640);  
}  
else{  
    this.posY = this.posY + 10;  
}
```



```
if(this.posY >= 400){  
    this.posY = -20;  
    this.posX = ThreadLocalRandom.current().nextInt(0, 640);  
}  
else{  
    this.posY = this.posY + 10;  
}
```

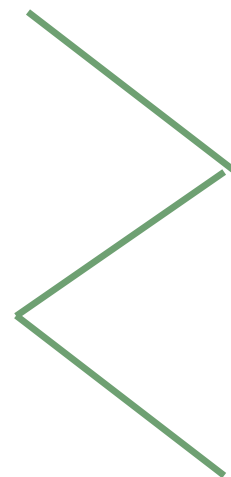


```
public class InimigoEsperto extends PersonagemPadrao implements Observavel {

    int somaX = 5;

    public InimigoEsperto(String pathImage, int posX, int posY) {
        super(pathImage, posX, posY);
    }

    @Override
    public void update(Observavel ob) {
        int minX = this.posX - 30;
        int maxX = this.posX + 30;
        if(this.posY >= 400){
            this.posY = -20;
            this.posX = ThreadLocalRandom.current().nextInt(0, 640);
        }
        else{
            this.posY = this.posY + 10;
            if(this.posX > maxX){
                somaX = 5;
            } else if (this.posX < minX) {
                somaX = -5;
            }
            this.posX = this.posX + somaX;
        }
    }
}
```



```
@Override
public void init(GameContainer gc) throws SlickException {

    try {
        File file = new File(".");
        String filePath = file.getCanonicalPath();

        land = new Image(filePath + "\\src\\main\\java\\bg.jpg");
        heroi = new Heroi(filePath + "\\src\\main\\java\\heroi1.png", 200, 400);
        inimigo1 = new Inimigo(filePath + "\\src\\main\\java\\devil1.png", 500, -100);
        inimigo2 = new Inimigo(filePath + "\\src\\main\\java\\devil2.png", 300, -00);
        inimigo3 = new Inimigo(filePath + "\\src\\main\\java\\devil3.png", 400, -250);
        inimigoEsperto = new InimigoEsperto(filePath + "\\src\\main\\java\\devil3.png", 200, -250);
        gameover = new Image(filePath + "\\src\\main\\java\\gameover.png");

        heroi.addMonitores(inimigo1);
        heroi.addMonitores(inimigo2);
```


pacote default>

ame.observer

ControleColisao.java

FabricaDeInimigos.java

FabricaDeNotificacoes.java

Heroi.java

Inimigo.java

InimigoEsperto.java

Observador.java

Observavel.java

PersonagemPadrao.java

SimpleSlickGame.java

es de Teste

os Códigos-fonte

ndências

ndências de Runtime

ndências Java

vos do Projeto

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

```
import org.newdawn.slick.Image;

public class FabricaDeInimigos {
    String filePath;
    FabricaDeInimigos() {
        try {
            File file = new File(".");
            filePath = file.getCanonicalPath();
        } catch (IOException ex) {
            Logger.getLogger(FabricaDeInimigos.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    public Inimigo criar(tipoInimigo) {
        return new Inimigo(filePath + "\\src\\m");
    }
}
```

Adicionando a classe
que vai criar os objetos

Método Fábrica

```
public Inimigo criarInimigo(int tipoInimigo) {  
    Inimigo inimigo = null;  
    if (tipoInimigo == 1) {  
        inimigo = new Inimigo(filePath + "\\src\\main\\java\\devil1.png", 500, -100);  
    } else if (tipoInimigo == 2) {  
        inimigo = new Inimigo(filePath + "\\src\\main\\java\\devil2.png", 300, -00);  
    } else if (tipoInimigo == 3) {  
        inimigo = new Inimigo(filePath + "\\src\\main\\java\\devil3.png", 400, -250);  
    }  
    else if (tipoInimigo == 4) {  
        inimigo = new InimigoEsperto(filePath + "\\src\\main\\java\\devil3.png", 200, -250);  
    }  
    return inimigo;  
}
```

Usando na classe principal

FabricaDeInimigos fabricaDeInimigos =
new FabricaDeInimigos();

```
inimigo1 = fabricaDeInimigos.criarInimigo(1);  
inimigo2 = fabricaDeInimigos.criarInimigo(2);  
inimigo3 = fabricaDeInimigos.criarInimigo(3);  
inimigoEsperto = fabricaDeInimigos.criarInimigo(4);
```

PADRÕES CRIATIVOS

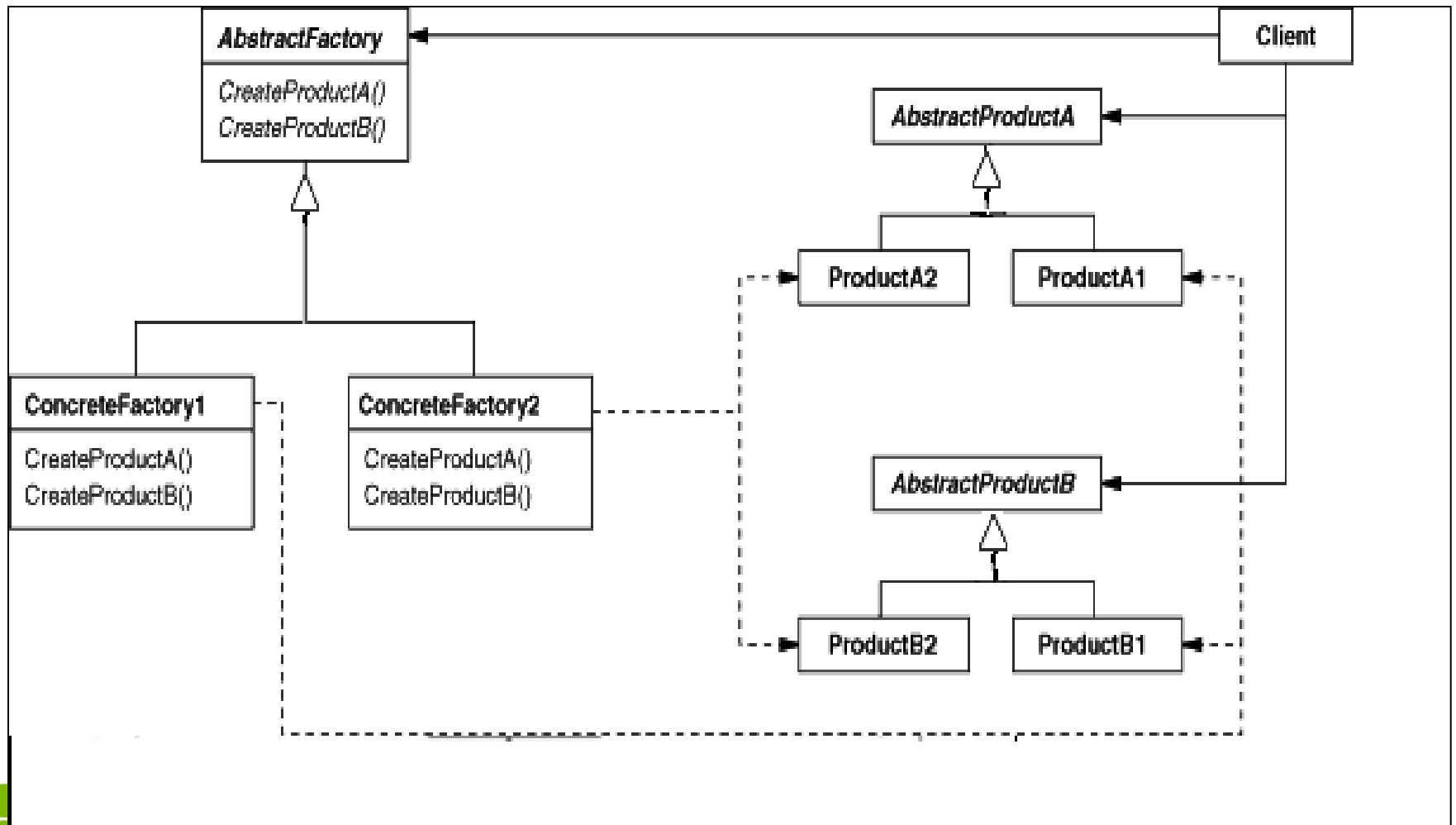
Fábrica Abstrata

PADRÕES DE PROJETO

Fábrica Abstrata:

- **Propósito:** Nesse padrão, em vez de termos uma fábrica para a criação de um objeto, ela é destinada à criação de uma família de objetos relacionados. Dessa forma, se todos os objetos relacionados forem obtidos a partir da mesma fábrica, não haverá inconsistência entre eles.

Fábrica Abstrata



Fábrica Abstrata

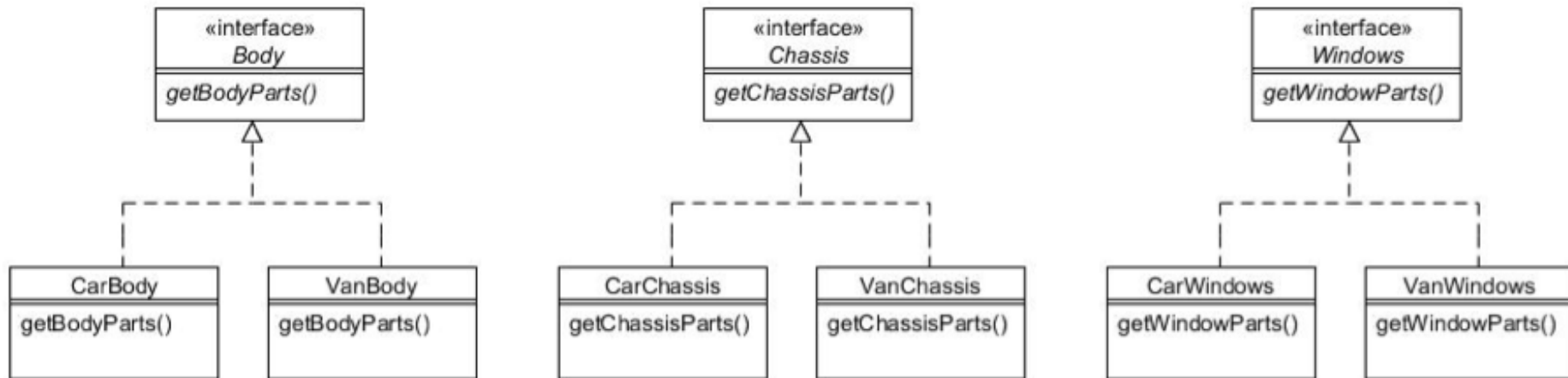
- **Problema:**

- A Motores SA fabrica os carros e vans com a lataria, chassi e janelas.
- Embora os carros e vans precisem dos mesmo tipos de componentes, a especificação de cada tipo difere do tipo de veículo;

PADRÕES DE PROJETO

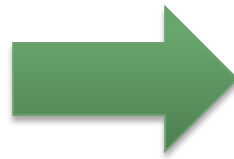
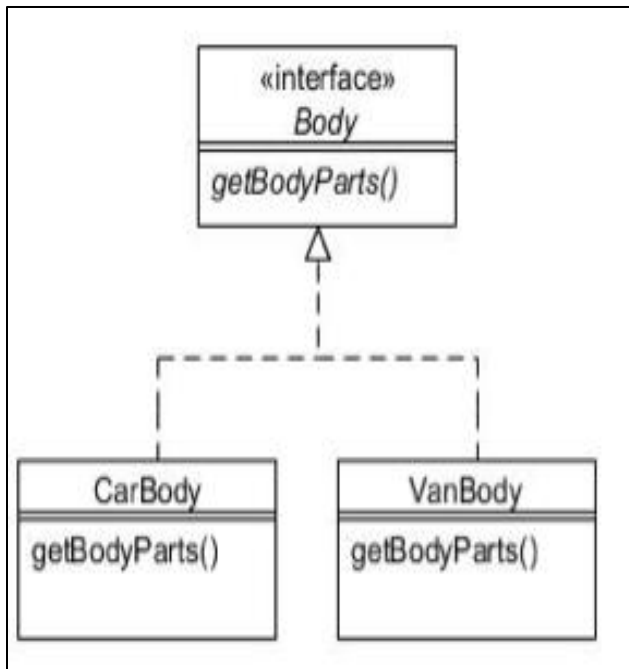
- Fábrica Abstrata:

- Para construirmos um veículo devemos pensar que os componentes possuem diferentes “famílias”!



PADRÕES DE PROJETO

- Fábrica Abstrata:

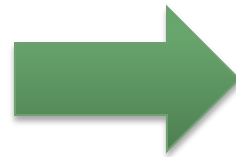
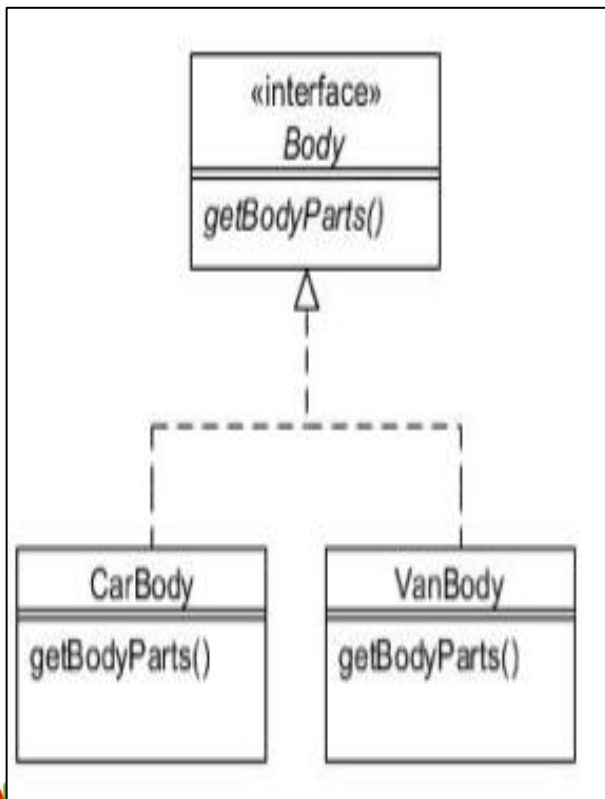


```
public interface Body {  
    public String getBodyParts();  
}
```

```
public class CarBody implements Body {  
    public String getBodyParts() {  
        return "Body shell parts for a car";  
    }  
}
```

PADRÕES DE PROJETO

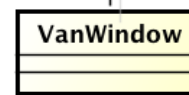
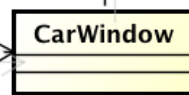
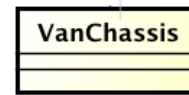
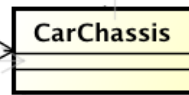
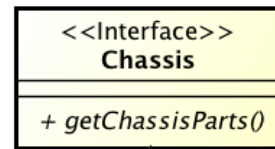
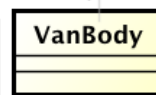
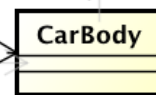
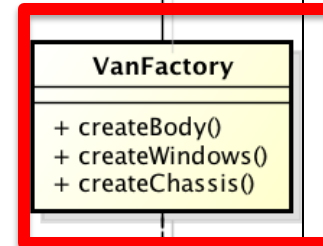
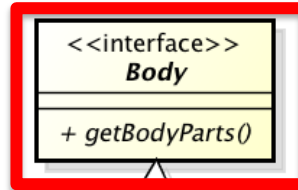
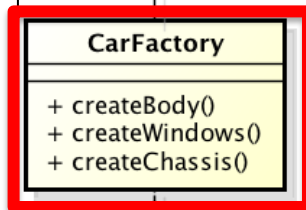
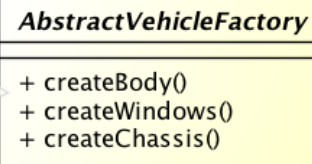
- Fábrica Abstrata:



```
public interface Body {  
    public String getBodyParts();  
}
```

```
public class CarBody implements Body {  
    public String getBodyParts() {  
        return "Body shell parts for a car";  
    }  
}
```

Fábrica Abstrata



Fábrica de Carros!

Fábrica dos Van!

Por que utilizar interfaces?

PADRÕES DE PROJETO

- Fábrica Abstrata:

- A Classe *AbstractVehicleFactory* é a responsável por definir os métodos: `createBody()`, `createChassis()` e `createWindows()`;

```
public abstract class AbstractVehicleFactory {  
    public abstract Body createBody();  
    public abstract Chassis createChassis();  
    public abstract Windows createWindows();  
}
```

PADRÕES DE PROJETO

- Fábrica Abstrata:

- A Classe CarFactory é a responsável por retornar os objetos da família do tipo carro.

```
public class CarFactory extends AbstractVehicleFactory {  
    public Body createBody() {  
        return new CarBody();  
    }  
  
    public Chassis createChassis() {  
        return new CarChassis();  
    }  
  
    public Windows createWindows() {  
        return new CarWindows();  
    }  
}
```

PADRÕES DE PROJETO

- Fábrica Abstrata:
 - A Classe VanFactory é a responsável por retornar os objetos da família do tipo Van.

```
public class VanFactory extends AbstractVehicleFactory {  
    public Body createBody() {  
        return new VanBody();  
    }  
  
    public Chassis createChassis() {  
        return new VanChassis();  
    }  
  
    public Windows createWindows() {  
        return new VanWindows();  
    }  
}
```

Como eu uso isso em código?

```
String whatToMake = "car"; // or "van"
AbstractVehicleFactory factory = null;

// Create the correct 'factory'...
if (whatToMake.equals("car")) {
    factory = new CarFactory();
} else {
    factory = new VanFactory();
}

// Create the vehicle's component parts...
// These will either be all car parts or all van parts.
Body vehicleBody = factory.createBody();
Chassis vehicleChassis = factory.createChassis();
Windows vehicleWindows = factory.createWindows();

// Show what we've created...
System.out.println(vehicleBody.getBodyParts());
System.out.println(vehicleChassis.getChassisParts());
System.out.println(vehicleWindows.getWindowParts());
```

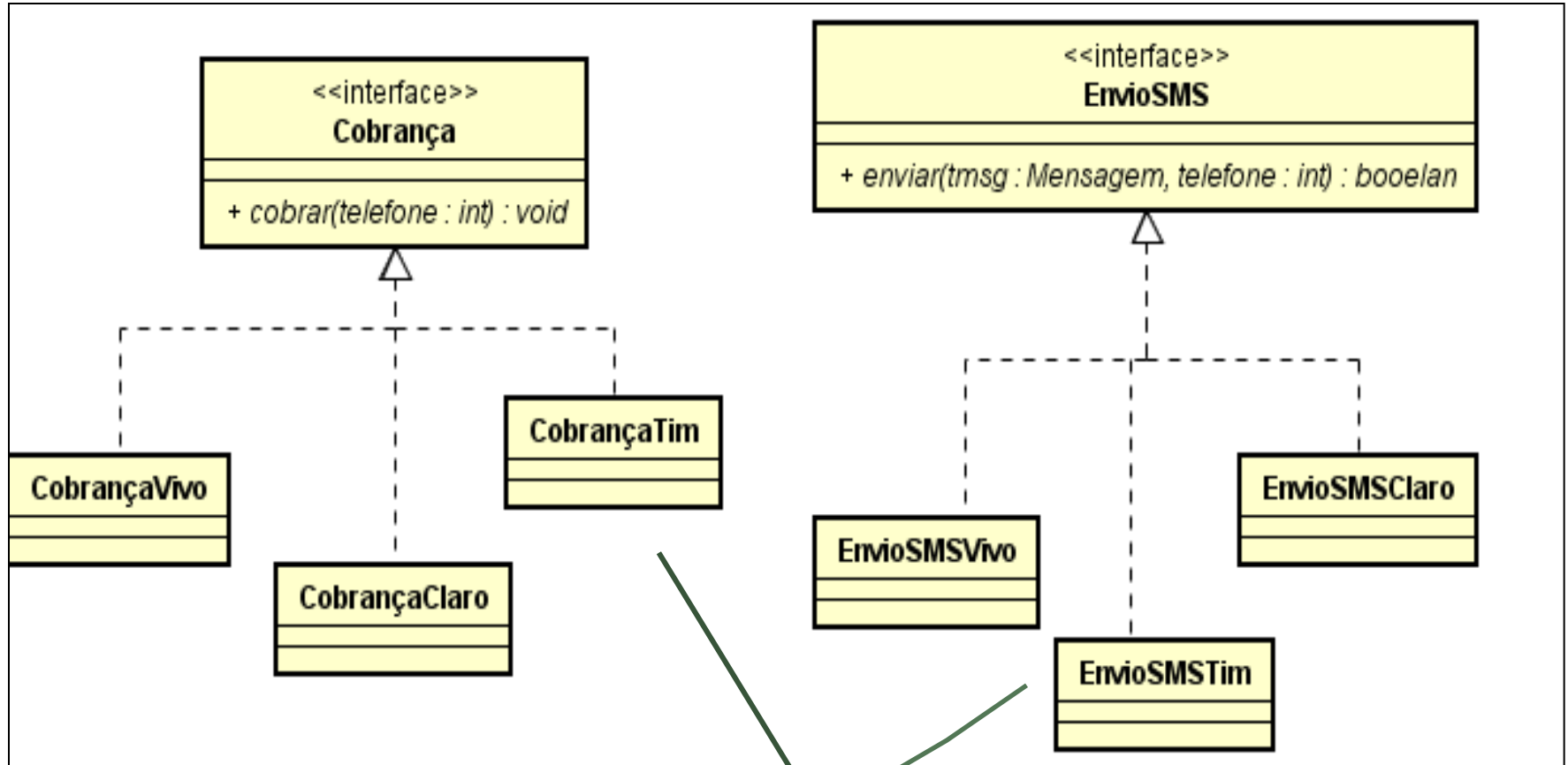
Criando objetos para o envio de SMS

Vamos imaginar que uma aplicação precise interagir com o sistema de SMS de diversas operadoras de telefonia móvel.

Também vamos supor que existam diversos objetos que precisem ser criados para interagir com esse serviço como:

- **Cobrança:** interface que define a cobrança do envio do sms.
- **EnvioSMS:** interface que define o envio do SMS.

Envio SMS



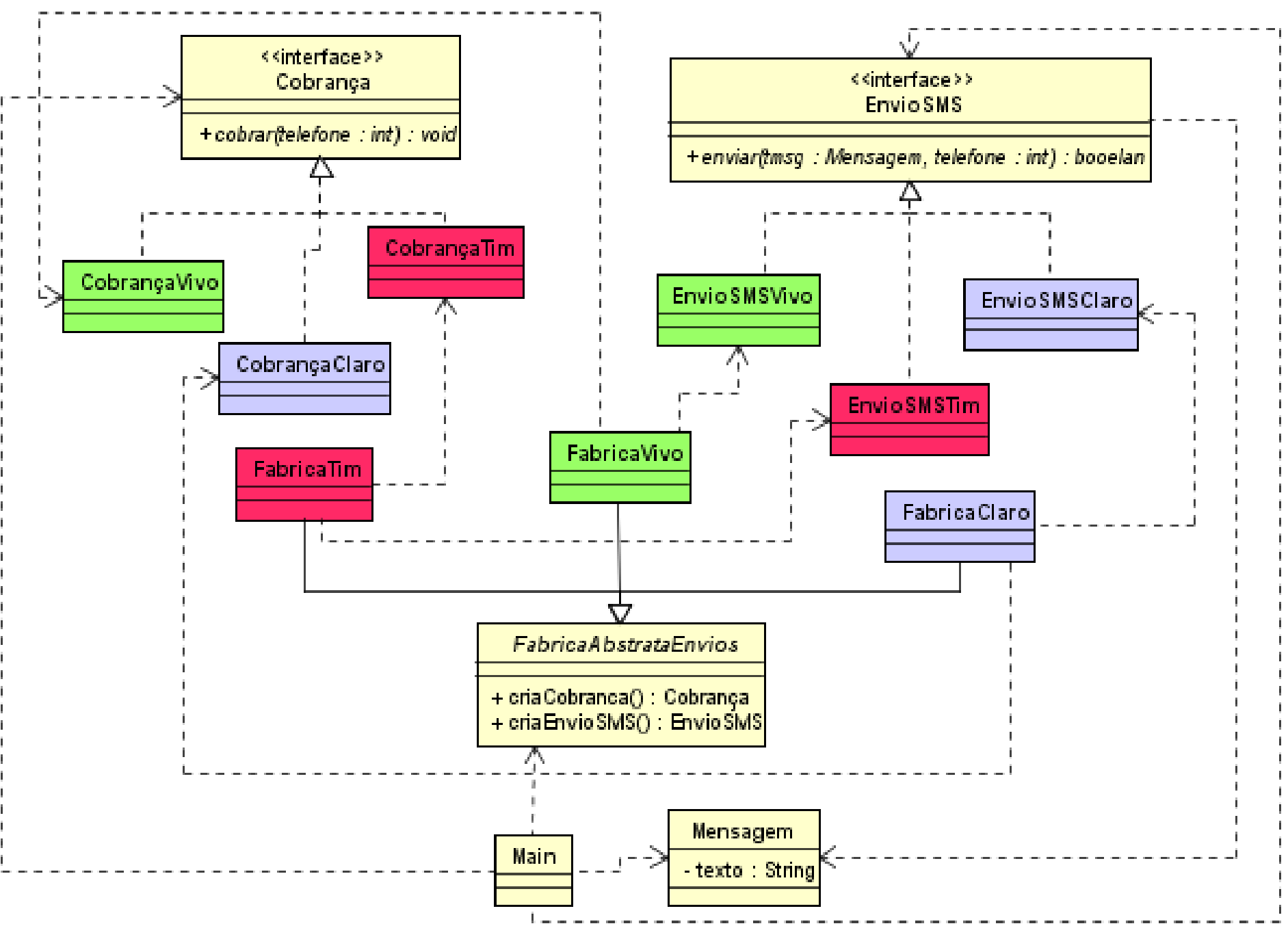
Preciso garantir que ao Instanciar um EnvioSMSTim seja instanciado uma CobrançaTIM

Implementação sem Fábrica Abstrata

```
public static void main(String[] args) {

    String escolha = JOptionPane.showInputDialog("Tim ou Vivo?");
    int telefone = Integer.parseInt(JOptionPane.showInputDialog("Telefone?"));
    String texto = JOptionPane.showInputDialog("Mensagem");
    Mensagem msg = new Mensagem(texto);
    EnvioSMS envio = null;
    Cobranca cobranca = null;
    if (escolha.equalsIgnoreCase("tim")) {
        envio = new EnvioSMSTim("assincrono");
        if (envio.enviar(msg, telefone)) {
            cobranca = new CobrancaTim();
            cobranca.cobrar(telefone);
        } else if (escolha.equalsIgnoreCase("vivo")) {
            envio = new EnvioSMSVivo();
            if (envio.enviar(msg, telefone)) {
                cobranca = new CobrancaTim();
                cobranca.cobrar(telefone);
            }
        }
    }
}
```





Implementação

```
public interface Cobranca {  
    public void cobrar(int telefone);  
}
```

```
public interface EnvioSMS {  
    public boolean enviar(Mensagem msg, int telefone);  
}
```

Implementação

```
public class CobrancaVivo implements Cobranca {  
  
    @Override  
    public void cobrar(int telefone) {  
        System.out.println("Cobrar via vivo");  
    }  
}
```

```
public class CobrancaTim implements Cobranca {  
  
    @Override  
    public void cobrar(int telefone) {  
        System.out.println("Cobrar via tim");  
    }  
}
```



```
public class EnvioSMSTim implements EnvioSMS{

    public EnvioSMSTim(String config){
        //realiza configuracao especifica para tim.
    }

    @Override
    public boolean enviar(Mensagem msg, int telefone) {
        System.out.println("Enviando mensagem:" + msg.texto +
            " para o telefone tim:" + telefone );
        return true;
    }

}
```

```
public class EnvioSMSVivo implements EnvioSMS{

    @Override
    public boolean enviar(Mensagem msg, int telefone) {
        System.out.println("Enviando mensagem:" + msg.texto +
            " para o telefone vivo:" + telefone );
        return true;
    }

}
```



Implementação

```
public abstract class FabricaAbstrataEnvios {  
    public abstract Cobranca criaCobranca();  
    public abstract EnvioSMS criaEnvioSMS();  
}
```

```
public class FabricaTim extends FabricaAbstrataEnvios{  
  
    @Override  
    public EnvioSMS criaEnvioSMS () {  
        return new EnvioSMSTim("Assincrono");  
    }  
  
    @Override  
    public Cobranca criaCobranca() {  
        return new CobrancaTim();  
    }  
}
```



Implementação

```
public class FabricaVivo extends FabricaAbstrataEnvios{

    @Override
    public Cobranca criaCobranca() {
        return new CobrancaVivo();
    }

    @Override
    public EnvioSMS criaEnvioSMS() {
        return new EnvioSMSVivo();
    }

}
```


Utilização da Fábrica Abstrata

```
public static void main(String[] args) {  
  
    String escolha = JOptionPane.showInputDialog("Tim ou Vivo?");  
    int telefone = Integer.parseInt(JOptionPane.showInputDialog("Telefone?"));  
    String texto = JOptionPane.showInputDialog("Mensagem");  
    Mensagem msg = new Mensagem(texto);  
    FabricaAbstrataEnvios fabrica = null;  
    if(escolha.equalsIgnoreCase("tim")){  
        fabrica = new FabricaTim();  
    }else if(escolha.equalsIgnoreCase("vivo")){  
        fabrica = new FabricaVivo();  
    }  
    EnvioSMS envio = fabrica.criaEnvioSMS();  
    if(envio.enviar(msg, telefone)){  
        Cobranca cobranca = fabrica.criaCobranca();  
        cobranca.cobrar(telefone);  
    }  
}
```

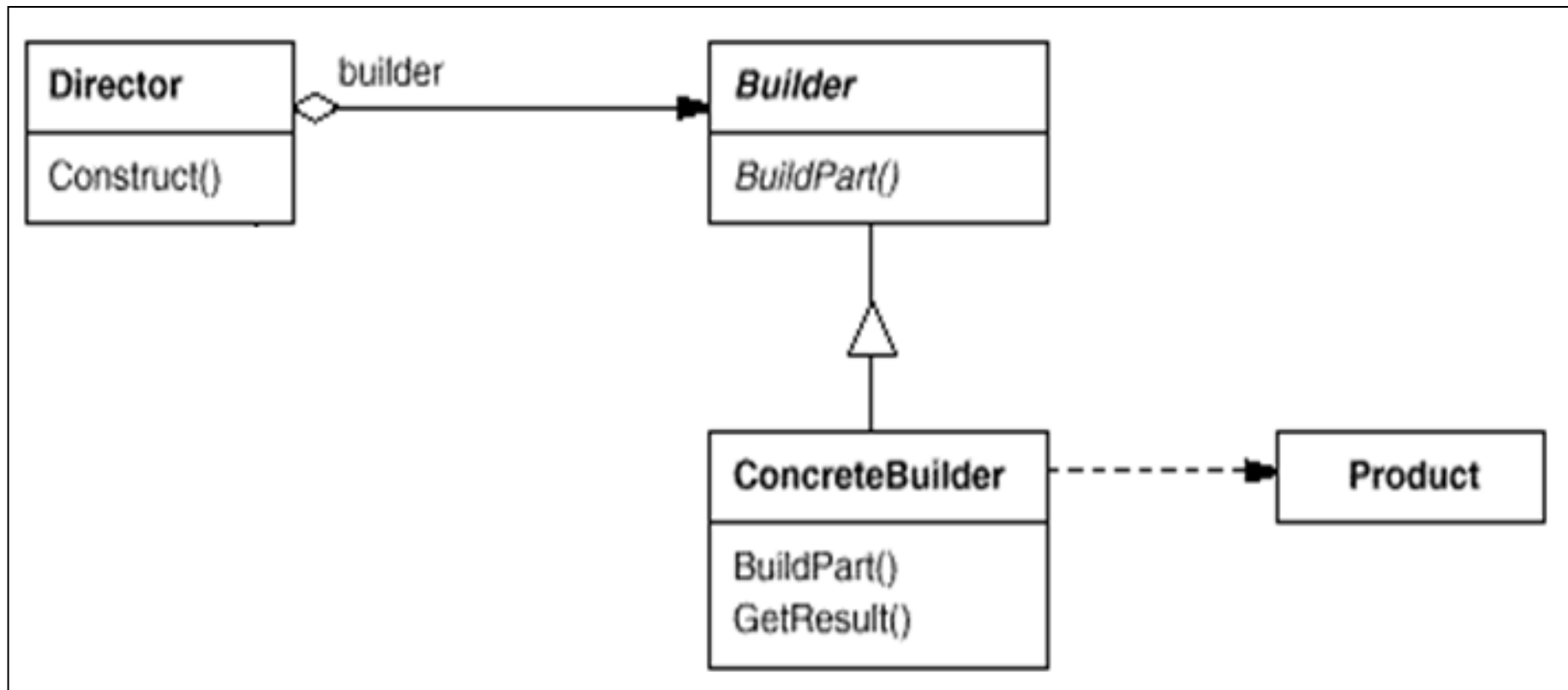
Código - Padrão Fábrica

PADRÕES DE PROJETO

- **Aplicabilidade (Quando usar?)**
 - Um sistema deve ser independente de como seus produtos são criados, compostos ou representados;
 - Uma família de objetos for projetada para ser usada em conjunto, e você necessita garantir esta restrição;
 - Você quer fornecer uma biblioteca de classes e quer revelar somente suas interfaces, não suas implementações.

PADRÕES CRIATIVOS Builder

PADRÕES DE PROJETO



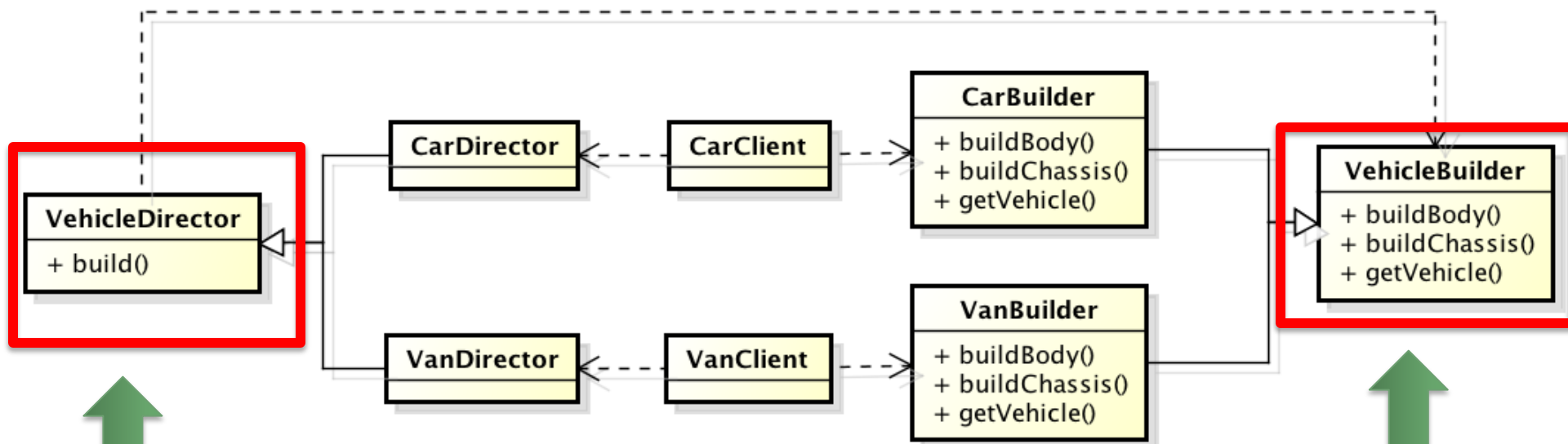
PADRÕES DE PROJETO

- **Builder:**
 - **Propósito:** permite criar diversos objetos semelhantes, utilizando o mesmo algoritmo;
 - **Problema:**
 - A Motores SA fabrica os carros e vans. Porém, o processo de construção difere em detalhes para cada veículo:
 - Van: possui uma grande área de carga reforçada;
 - Saloon: possui uma área dos passageiros e;
 - **Como construir diferentes carros com o mesmo material e usando o mesmo processo?**

PADRÕES DE PROJETO

- Builder:
 - O padrão *Builder* facilita a construção de objetos complexos da seguinte forma:
 1. Separando os métodos utilizados em hierarquia de Builder;
 2. Usando um Director (diretor) que especifica quais etapas são requeridas e a ordem;
 3. Builder retorna o objeto construído;

PADRÕES DE PROJETO



Diretor!

Builder: tem os métodos(tarefas)
Director: sabe como organizar as tarefas(ordem)

Builder!

PADRÕES DE PROJETO

- *VehicleBuilder* possui todos os métodos necessários para construir os veículos;
- *VehicleBuilder* também possui os método *getVehicle()* que retorna o veículo construído;

```
public abstract class VehicleBuilder {  
    public void buildBody() {}  
    public void buildBoot() {}  
    public void buildChassis() {}  
    public void buildPassengerArea() {}  
    public void buildReinforcedStorageArea() {}  
    public void buildWindows() {}  
  
    public abstract Vehicle getVehicle();  
}
```



PADRÕES DE PROJETO

CarBuilder responsável por construir as partes do carro;

```
public class CarBuilder extends VehicleBuilder {
    private AbstractCar carInProgress;

    public CarBuilder(AbstractCar car) {
        carInProgress = car;
    }

    public void buildBody() {
        // Add body to carInProgress
        System.out.println("building car body");
    }

    public void buildBoot() {
        // Add boot to carInProgress
        System.out.println("building car boot");
    }

    public void buildChassis() {
        // Add chassis to carInProgress
        System.out.println("building car chassis");
    }
}
```

```
    public void buildPassengerArea() {
        // Add passenger area to carInProgress
        System.out.println("building car passenger area");
    }

    public void buildWindows() {
        // Add windows to carInProgress
        System.out.println("building car windows");
    }

    public Vehicle getVehicle() {
        return carInProgress;
    }
}
```

PADRÕES DE PROJETO

- *CarBuilder* responsável por construir as partes do carro;

```
public class CarBuilder extends VehicleBuilder {
    private AbstractCar carInProgress;

    public CarBuilder(AbstractCar car) {
        carInProgress = car;
    }

    public void buildBody() {
        // Add body to carInProgress
        System.out.println("building car body");
    }

    public void buildBoot() {
        // Add boot to carInProgress
        System.out.println("building car boot");
    }

    public void buildChassis() {
        // Add chassis to carInProgress
        System.out.println("building car chassis");
    }
}
```

```
    public void buildPassengerArea() {
        // Add passenger area to carInProgress
        System.out.println("building car passenger area");
    }

    public void buildWindows() {
        // Add windows to carInProgress
        System.out.println("building car windows");
    }

    public Vehicle getVehicle() {
        return carInProgress;
    }
}
```

PADRÕES DE PROJETO

CarDirector responsável por definir o passo-a-passo do processo da construção de um Carro;

```
public class CarDirector Extends VehicleDirector {  
  
    public Vehicle build(VehicleBuilder builder){  
        builder.buildChassis();  
        builder.buildBody();  
        builder.buildPassengerArea();  
        builder.buildWindows();  
        return builder.getVehicle();  
    }  
}
```

PADRÕES DE PROJETO

- *VanDirector* responsável por definir o passo-a-passo do processo da construção de uma Van;

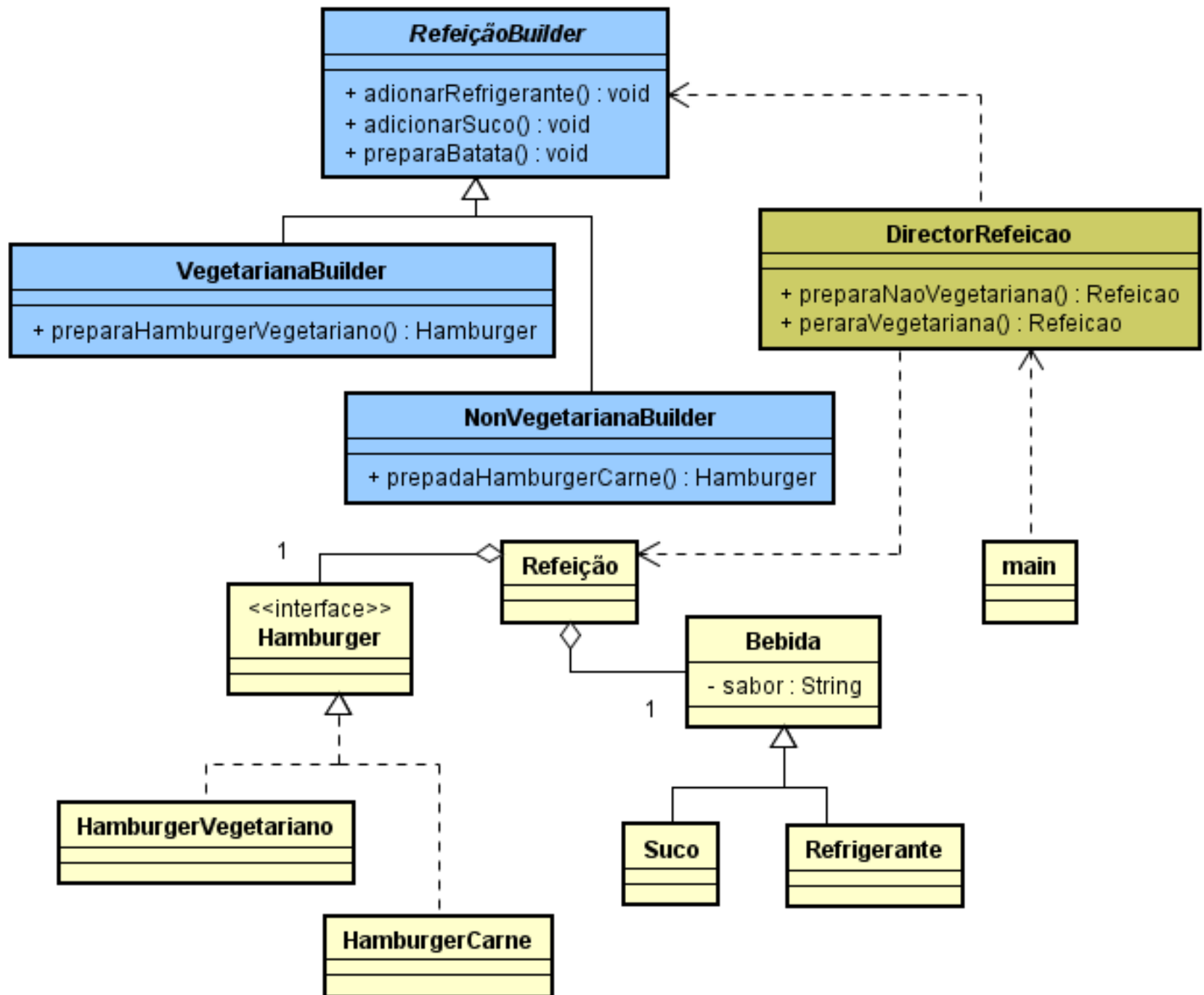
```
public class VanDirector extends VehicleDirector {  
    public Vehicle build(VehicleBuilder builder) {  
        builder.buildChassis();  
        builder.buildBody();  
        builder.buildReinforcedStorageArea();  
        builder.buildWindows();  
        return builder.getVehicle();  
    }  
}
```

Como eu uso isso em código?

```
AbstractCar car = new Saloon(new StandardEngine(1300));  
VehicleBuilder builder = new CarBuilder(car);  
VehicleDirector director = new CarDirector();  
Vehicle v = director.build(builder);  
System.out.println(v);
```

Padrão Builder

Uma lanchonete vende combinados de hambúrguer com bebida. Vamos utilizar o padrão builder para organizar a construção dos **lanches/refeição**.




```
public interface Hamburger {  
  
}
```

```
public class Bebida {  
  
    private String sabor;  
    public String getSabor() {  
        return sabor;  
    }  
    public void setSabor(String sabor) {  
        this.sabor = sabor;  
    }  
}
```

```
public class Refeicao {  
  
    private Hamburger hamburger;  
    private Bebida bebida;  
  
    public Hamburger getHamburger() {  
        return hamburger;  
    }  
    public void setHamburger(Hamburger hamburger) {  
        this.hamburger = hamburger;  
    }  
    public Bebida getBebida() {  
        return bebida;  
    }  
    public void setBebida(Bebida bebida) {  
        this.bebida = bebida;  
    }  
}
```



```
public abstract class RefeicaoBuilder {  
    Refeicao refeicao;  
  
    public void adicionarSuco(String sabor) {  
        refeicao.setBebida(new Suco(sabor));  
    }  
  
    public void adicionarRefrigerante(String sabor) {  
        refeicao.setBebida(new Refrigerante(sabor));  
    }  
}
```

```
public class NonVegetarianoBuilder extends RefeicaoBuilder{

    public NonVegetarianoBuilder(Refeicao novaRefeicao){
        this.refeicao = novaRefeicao;
    }

    public void preparaHamburgerCarne() {
        System.out.println("cria hamburger carne");
        this.refeicao.setHamburger(new HamburgerCarne());
    }

}
```

```
public class VegetarianoBuilder extends RefeicaoBuilder{

    public VegetarianoBuilder(Refeicao refeicao){
        this.refeicao = refeicao;
    }

    public void preparaHamburgerVegetariano() {
        System.out.println("cria hamburger vegetariano");
        this.refeicao.setHamburger(new HamburgerVegetariano());
    }

}
```



```
public class DiretorRefeicao {
```

```
    public Refeicao preparaNaoVegetariana(String sabor, String tipoBebida){
```

```
        Refeicao novaRefeicao = new Refeicao();
```

```
        NonVegetarianoBuilder refeicaoBuilder = new NonVegetarianoBuilder(novaRefeicao);
```

```
        if(tipoBebida.equalsIgnoreCase("refrigerante"))
```

```
            refeicaoBuilder.adicionarRefrigerante(sabor);
```

```
        else
```

```
            refeicaoBuilder.adicionarSuco(sabor);
```

```
        refeicaoBuilder.preparaHamburgerCarne();
```

```
        return novaRefeicao;
```

```
    }
```

```
    public Refeicao preparaVegetariana(String sabor, String tipoBebida){
```

```
        Refeicao novaRefeicao = new Refeicao();
```

```
        VegetarianoBuilder refeicaoBuilder = new VegetarianoBuilder(novaRefeicao);
```

```
        if(tipoBebida.equalsIgnoreCase("refrigerante"))
```

```
            refeicaoBuilder.adicionarRefrigerante(sabor);
```

```
        else
```

```
            refeicaoBuilder.adicionarSuco(sabor);
```

```
        refeicaoBuilder.preparaHamburgerVegetariano();
```

```
        return novaRefeicao;
```

```
    }
```

```
}
```

Podemos usar algum
outro padrão de
projeto aqui?

Chamando na Main para construir uma Refeição

```
public class Main {  
  
    public static void main(String[] args) {  
  
        DiretorRefeicao criaRefeicao = new DiretorRefeicao();  
        //controi uma refeicao.  
        Refeicao refeicao1 = criaRefeicao.preparaNaoVegetariana("laranja", "suco");  
        Refeicao refeicao2 = criaRefeicao.preparaVegetariana("guarana", "refrigerante");  
    }  
}
```

PADRÕES DE PROJETO

- **Oferece um controle fino sobre o processo de construção:**
 - o Builder constrói o produto passo a passo sob o controle do diretor;
 - Somente quando o produto está terminado o diretor recupera o construtor;

PADRÕES CRIATIVOS

Protótipo

PADRÕES DE PROJETO

- Protótipo:
 - **Propósito:** especifica os tipos de objetos que serão criados através de protótipos e cria um novo objetivo copiando um protótipo;
 - **Problema:**
 - Imagine que seja necessário criar um carro e uma van em tempo de execução e da maneira mais rápida possível. Como faremos isso ?
 - No Java a técnica que mais se adequa ao padrão Protótipo é o clone();

PADRÕES DE PROJETO

- Protótipo:

```
public interface Vehicle extends Cloneable {  
    public enum Colour {UNPAINTED, BLUE, BLACK, GREEN, RED, SILVER, WHITE, YELLOW};  
  
    public Engine getEngine();  
    public Vehicle.Colour getColour();  
    public void paint(Vehicle.Colour colour);  
  
    public Object clone();  
}
```

PADRÕES DE PROJETO

Modificação na Classe AbstractVehicle

```
public Object clone() {  
    Object obj = null;  
    try {  
        obj = super.clone();  
    } catch (CloneNotSupportedException x) {  
        // Should not happen...  
    }  
    return obj;  
}
```

Método responsável por clonar.
Clone é protegido por isso preciso sobrescrever....

PADRÕES DE PROJETO

- Protótipo:
 - Agora devemos criar uma classe que é responsável por gerenciar as instâncias: **VechileManager**.

```

public class VehicleManager {
    private Vehicle saloon, coupe, sport, boxVan, pickup;

    public VehicleManager() {
        // For simplicity all vehicles use same engine type...
        saloon = new Saloon(new StandardEngine(1300));
        coupe = new Coupe(new StandardEngine(1300));
        sport = new Sport(new StandardEngine(1300));
        boxVan = new BoxVan(new StandardEngine(1300));
        pickup = new Pickup(new StandardEngine(1300));
    }

    public Vehicle createSaloon() {
        return (Vehicle) saloon.clone();
    }

    public Vehicle createCoupe() {
        return (Vehicle) coupe.clone();
    }

    public Vehicle createSport() {
        return (Vehicle) sport.clone();
    }

    public Vehicle createBoxVan() {
        return (Vehicle) boxVan.clone();
    }

    public Vehicle createPickup() {
        return (Vehicle) pickup.clone();
    }
}

```

```
public class VehicleManager {  
    private Vehicle saloon, coupe, sport, boxVan, pickup;  
  
    public VehicleManager() {  
        // For simplicity all vehicles use same engine type...  
        saloon = new Saloon(new StandardEngine(1300));  
        coupe = new Coupe(new StandardEngine(1300));  
        sport = new Sport(new StandardEngine(1300));  
        boxVan = new BoxVan(new StandardEngine(1300));  
        pickup = new Pickup(new StandardEngine(1300));  
    }  
  
    public Vehicle createSaloon() {  
        return (Vehicle) saloon.clone();  
    }  
  
    public Vehicle createCoupe() {  
        return (Vehicle) coupe.clone();  
    }  
  
    public Vehicle createSport() {  
        return (Vehicle) sport.clone();  
    }  
  
    public Vehicle createBoxVan() {  
        return (Vehicle) boxVan.clone();  
    }  
  
    public Vehicle createPickup() {  
        return (Vehicle) pickup.clone();  
    }  
}
```

Criando os clones!



PADRÕES DE PROJETO


- Utilizando a Classe **VehicleManager**:

```
VehicleManager manager = new VehicleManager();  
Vehicle saloon1 = manager.createSaloon();  
Vehicle saloon2 = manager.createSaloon();  
Vehicle pickup1 = manager.createPickup();
```

PADRÕES DE PROJETO

- Protótipo:
 - É necessário instanciar todas as classes ao mesmo tempo para realizar o clone ?

```
public class VehicleManager {  
    private Vehicle saloon, coupe, sport, boxVan, pickup;  
  
    public VehicleManager() {  
        // For simplicity all vehicles use same engine type...  
        saloon = new Saloon(new StandardEngine(1300));  
        coupe = new Coupe(new StandardEngine(1300));  
        sport = new Sport(new StandardEngine(1300));  
        boxVan = new BoxVan(new StandardEngine(1300));  
        pickup = new Pickup(new StandardEngine(1300));  
    }  
}
```



PADRÕES DE PROJETO

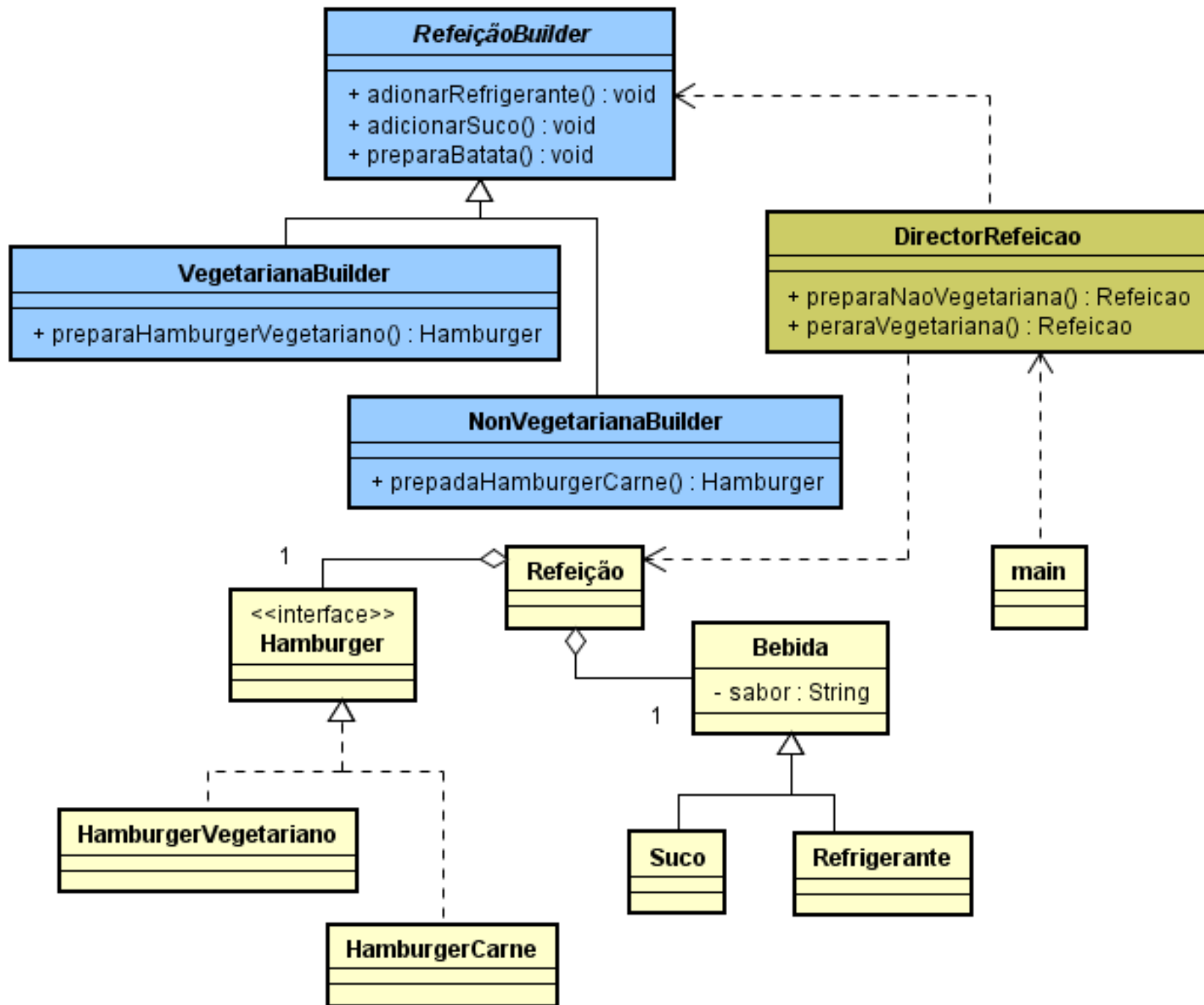
- Protótipo:
 - É necessário instanciar todas as classes ao mesmo tempo para realizar o clone ?
 - R.: Não. Podemos fazer uma instanciação tardia.

```
public class VehicleManagerLazy {  
    private Vehicle saloon, coupe, sport, boxVan, pickup;  
  
    public VehicleManagerLazy() {  
    }  
  
    public Vehicle createSaloon() {  
        if (saloon == null) {  
            saloon = new Saloon(new StandardEngine(1300));  
            return saloon;  
        } else {  
            return (Vehicle) saloon.clone();  
        }  
    }  
}
```

Prototype

No exemplo da lanchonete vamos economizar tempo criando refeições de exemplo e clonando esses exemplos quando precisarmos de novos objetos.

https://github.com/felipefo/poo2/tree/master/Padroes_de_Projeto/Cria%C3%A7%C3%A3o/prototype/Prototype_BuilderRefeicao



Implements
Cloneable

Como ele é um
objeto compostos
de outros objetos
preciso de clonar
os objetos
compostos
também

Shallow copy (Cópia Superficial)

```
public class Refeicao implements Cloneable {  
    private Hamburger hamburger;  
    private Bebida bebida;  
    public Hamburger getHamburger() {  
        return hamburger;  
    }  
    public void setHamburger(Hamburger hamburger) {  
        this.hamburger = hamburger;  
    }  
    public Bebida getBebida() {  
        return bebida;  
    }  
    public void setBebida(Bebida bebida) {  
        this.bebida = bebida;  
    }  
    public Object clone() {  
        Object obj = null;  
        try {  
            obj = super.clone();  
        } catch (CloneNotSupportedException ex) {  
            ex.printStackTrace();  
        }  
        return obj;  
    }  
}
```

```

public class Refeicao implements Cloneable {
    private Hamburger hamburger;
    private Bebida bebida;
    public Hamburger getHamburger() {
        return hamburger;
    }
    public void setHamburger(Hamburger hamburger) {
        this.hamburger = hamburger;
    }
    public Bebida getBebida() {
        return bebida;
    }
    public void setBebida(Bebida bebida) {
        this.bebida = bebida;
    }
    public Object clone() {
        Object obj = null;
        try{
            obj = super.clone();
            bebida = (Bebida) bebida.clone();
            hamburger = (Hamburger) hamburger.clone();
        } catch (CloneNotSupportedException ex) {
            ex.printStackTrace();
        }
        return obj;
    }
}

```

Deep Cloning – Cópia Profunda

```
public class Refeicao implements Cloneable, Serializable{
    private Hamburger hamburger;
    private Bebida bebida;
    public Hamburger getHamburger() {
        return hamburger;
    }
    public void setHamburger(Hamburger hamburger) {
        this.hamburger = hamburger;
    }
    public Bebida getBebida() {
        return bebida;
    }
    public void setBebida(Bebida bebida) {
        this.bebida = bebida;
    }
    public Object clone(){
        Refeicao obj = null;
        try{
            obj = (Refeicao)super.clone();
            obj.setBebida((Bebida) bebida.clone());
            obj.setHamburger((Hamburger) hamburger.clone());
        }catch (CloneNotSupportedException ex){
            ex.printStackTrace();
        }
    }
}
```

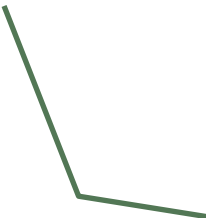


Usando na Main

```
public class Main {  
  
    static HashMap<String, Refeicao> refeicoes = new HashMap<>();  
  
    public static void main(String[] args) {  
        DiretorRefeicao criaRefeicao = new DiretorRefeicao();  
        //controi uma refeicao.  
        Refeicao refeicao1 = criaRefeicao.preparaNaoVegetariana("laranja", "suco");  
        refeicoes.put("laranja/suco" , refeicao1 );  
        Refeicao refeicao2 = criaRefeicao.preparaVegetariana("guarana", "refrigerante");  
        refeicoes.put("guarana/refrigerante" , refeicao2 );  
        Refeicao  refeicaoClone1 = (Refeicao) refeicoes.get("laranja/suco").clone();  
        Refeicao  refeicaoClone2 = (Refeicao) refeicoes.get("guarana/refrigerante").clone();  
        refeicao1.getBebida().setSabor("morango");  
        System.out.println("clone refeicao 1 " + refeicaoClone1.getBebida().getSabor());  
        System.out.println("refeicao1 " + refeicao1.getBebida().getSabor());  
    }  
}
```

Com serialização

```
/*
 * This method makes a "deep clone" of any Java object it is given.
 */
public static Object deepClone(Object object) {
    try {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(baos);
        oos.writeObject(object);
        ByteArrayInputStream bais = new ByteArrayInputStream(baos.toByteArray());
        ObjectInputStream ois = new ObjectInputStream(bais);
        return ois.readObject();
    }
    catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```



Deep Cloning – Cópia Profunda

Implements Serializable

```
public class Refeicao implements Cloneable, Serializable{  
    private Hamburger hamburger; _____  
    private Bebida bebida;  
    public Hamburger getHamburger() {  
        return hamburger;  
    }  
    public void setHamburger(Hamburger hamburger) {  
        this.hamburger = hamburger;  
    }  
    public Bebida getBebida() {  
        return bebida;  
    }  
    public void setBebida(Bebida bebida) {  
        this.bebida = bebida;  
    }  
}
```

Também
precisam
implementar

Implements Serializable

```
public class Hamburger implements Cloneable, Serializable {
```

```
public class Bebida implements Cloneable, Serializable{  
  
    private String sabor;  
    public String getSabor() {  
        return sabor;  
    }  
    public void setSabor(String sabor) {  
        this.sabor = sabor;  
    }  
}
```

Utilização com serialização

```
public static void main(String[] args) {  
    DiretorRefeicao criaRefeicao = new DiretorRefeicao();  
    //controi uma refeicao.  
    Refeicao refeicao1 = criaRefeicao.preparaNaoVegetariana("laranja", "suco");  
    refeicoes.put("laranja/suco", refeicao1 );  
    Refeicao refeicao2 = criaRefeicao.preparaVegetariana("guarana", "refrigerante");  
    refeicoes.put("guarana/refrigerante", refeicao2 );  
    Refeicao refeicaoClone1 = (Refeicao) refeicoes.get("laranja/suco").clone();  
    Refeicao refeicaoClone2 = (Refeicao) refeicoes.get("guarana/refrigerante").clone();  
    refeicao1.getBebida().setSabor("morango");  
    System.out.println("clone refeicao 1 " + refeicaoClone1.getBebida().getSabor());  
    System.out.println("refeicao1 " + refeicao1.getBebida().getSabor());  
  
    Refeicao novaRefeicao = (Refeicao) deepClone(refeicao1);  
    novaRefeicao.getBebida().setSabor("abacaxi");  
    System.out.println("nova Refeicao1: " + novaRefeicao.getBebida().getSabor());  
    System.out.println("refeicao1 " + refeicao1.getBebida().getSabor());  
}
```



PADRÕES DE PROJETO

- **Aplicabilidade (Quando usar?)**

- Use o padrão Prototype quando o seu sistema tiver que ser independente de como os seus produtos são criados, compostos e representados; e:
 - Quando as classes a instanciar forem especificadas em tempo de execução, por exemplo, por carga dinâmica;
 - Para evitar a construção de uma hierarquia de classes de fábricas paralela à hierarquia de classes de produto;
 - Quando as instâncias de uma classe puderem ter uma dentre poucas combinações diferentes de estados. Pode ser mais conveniente instalar um número correspondente de protótipos e cloná-los, ao invés de instanciar a classe manualmente, cada vez com um estado apropriado

PADRÕES DE PROJETO

- **Vantagens:**

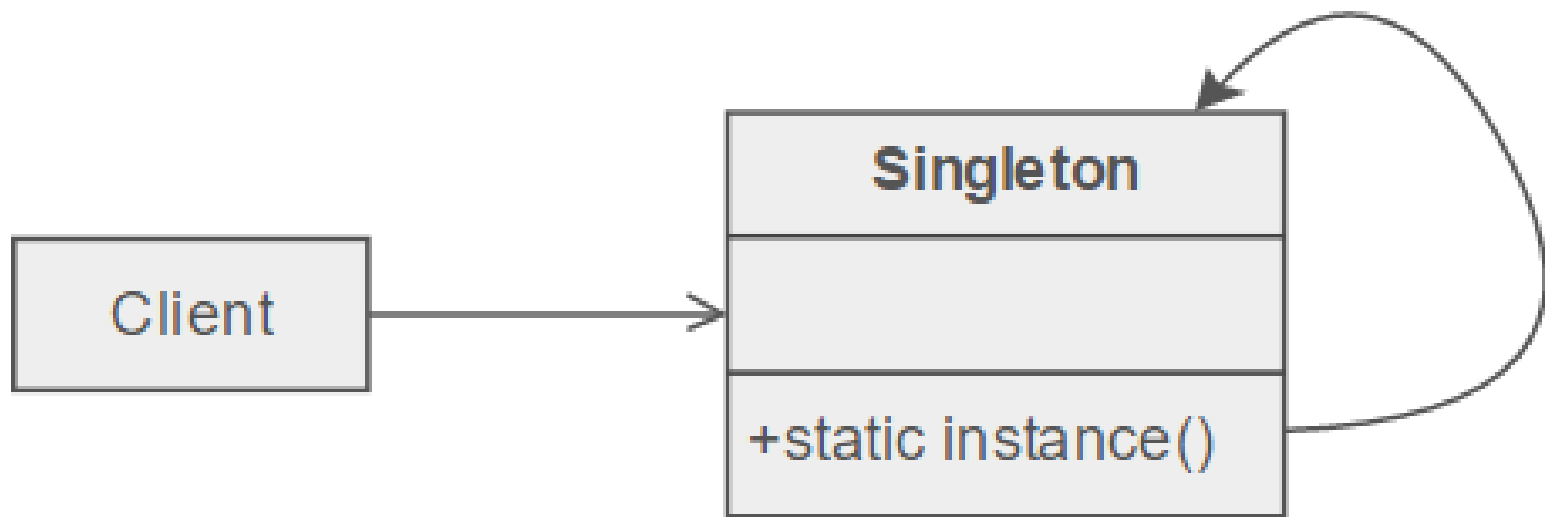
- Acrescentar e remover produtos em tempo de execução:
 - Permite adicionar uma nova classe concreta de produto a um sistema, simplesmente registrando uma instância protótipo com o cliente. Assim, o cliente pode adicionar e remover protótipo em tempo de execução;
- Especifica novos objetos pela variação de valores:
 - Sistemas altamente dinâmicos permitem definir novos comportamentos através da composição de objetos – por exemplo, pela especialização de valores para as variáveis de um objeto – e não pela definição de novas classes. Esse tipo de projeto permite ao usuário definir novas “Classes” sem ter que programar. Por exemplo, no editor musical, uma classe GraphicTool pode criar uma variedade ilimitada de objetos músicas.
- Reduzir o número de subclasses:
 - O padrão Prototype permite clonar um protótipo em vez de pedir um método fábrica para construir um novo objeto. Daí, não necessitar-se de nenhuma hierarquia de classes;

PADRÕES DE PROJETO

- **Desvantagens:**
 - O principal ponto fraco do padrão Prototype é que cada subclasse de Prototype deve implementar a operação clone, o que pode ser difícil.
Problemas:
 - Quando a estrutura interna de uma classe é complexa;
 - Ou quando existe referência circular;

PADRÕES CRIATIVOS

Singleton



PADRÕES DE PROJETO

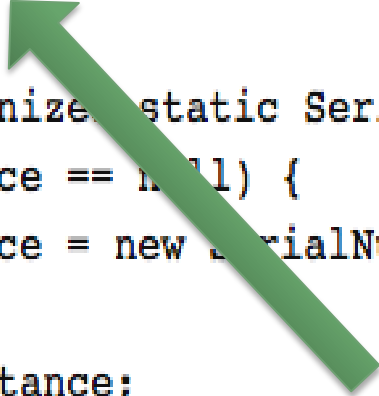
- Singleton:
 - **Propósito:** garante que apenas um objeto será criado e prover um único ponto de acesso para a esse objeto;
 - **Problema:**
 - Os carros somente podem ter um único serial;
 - Querem se assegurar que existe apenas um local onde esse serial é obtido;

PADRÕES DE PROJETO

- Singleton:
 - Como garantir que exista apenas uma instância de uma classe?
 - R: Na abordagem “tradicional”:
 - criar um método construtor “private”;
 - Prover um método *public static* para retornar um **objeto estático** da classe;

```
public class SerialNumberGeneratorTraditional {  
    // static members  
    private static SerialNumberGeneratorTraditional instance;  
  
    public synchronized static SerialNumberGeneratorTraditional getInstance() {  
        if (instance == null) {  
            instance = new SerialNumberGeneratorTraditional();  
        }  
        return instance;  
    }  
  
    // instance variables  
    private int count;  
  
    // private constructor  
    private SerialNumberGeneratorTraditional() {}  
  
    // instance methods  
    public synchronized int getNextSerial() {  
        return ++count;  
    }  
}
```

```
public class SerialNumberGeneratorTraditional {  
    // static members  
    private static SerialNumberGeneratorTraditional instance;  
  
    public synchronized static SerialNumberGeneratorTraditional getInstance() {  
        if (instance == null) {  
            instance = new SerialNumberGeneratorTraditional();  
        }  
        return instance;  
    }  
}
```



Instância estática do objeto

```
// instance variables  
private int count;
```

```
// private constructor  
private SerialNumberGeneratorTraditional() {}
```

```
// instance methods  
public synchronized int getNextSerial() {  
    return ++count;  
}
```

```
public class SerialNumberGeneratorTraditional {  
    // static members  
    private static SerialNumberGeneratorTraditional instance;  
  
    public synchronized static SerialNumberGeneratorTraditional getInstance() {  
        if (instance == null) {  
            instance = new SerialNumberGeneratorTraditional();  
        }  
        return instance;  
    }  
}
```

```
// instance variables  
private int count;
```

```
// private constructor  
private SerialNumberGeneratorTraditional() {}
```

```
// instance methods  
public synchronized int getNextSerial() {  
    return ++count;  
}
```



**Utilizou o synchronized para evitar
concorrência entre threads!**

```
public class SerialNumberGeneratorTraditional {  
    // static members  
    private static SerialNumberGeneratorTraditional instance;  
  
    public synchronized static SerialNumberGeneratorTraditional getInstance() {  
        if (instance == null) {  
            instance = new SerialNumberGeneratorTraditional();  
        }  
        return instance;  
    }  
  
    // instance variables  
    private int count;  
  
    // private constructor  
    private SerialNumberGeneratorTraditional() {}  
  
    // instance methods  
    public synchronized int getNextSerial() {  
        return ++count;  
    }  
}
```

**Constructor
privado!**



PADRÕES DE PROJETO

- Singleton:
 - “Executando” o singleton;

```
System.out.println("Using traditional singleton");  
SerialNumberGeneratorTraditional generator = SerialNumberGeneratorTraditional.getInstance();  
System.out.println("next serial: " + generator.getNextSerial());  
System.out.println("next serial: " + generator.getNextSerial());  
System.out.println("next serial: " + generator.getNextSerial());
```

PADRÕES DE PROJETO

- Singleton:
 - Será que existe um modo mais elegante de aplicar o padrão singleton?

PADRÕES DE PROJETO

- Singleton:
 - Será que existe um modo mais elegante de aplicar o padrão singleton?
 - R.: No Java 1.5 é feito através de ENUM.
 - COMO??



PADRÕES DE PROJETO

- Singleton:

```
public enum SerialNumberGenerator {  
    INSTANCE;  
  
    private int count;  
  
    public synchronized int getNextSerial() {  
        return ++count;  
    }  
}
```



Como eu uso
isso?

PADRÕES DE PROJETO

- Singleton:

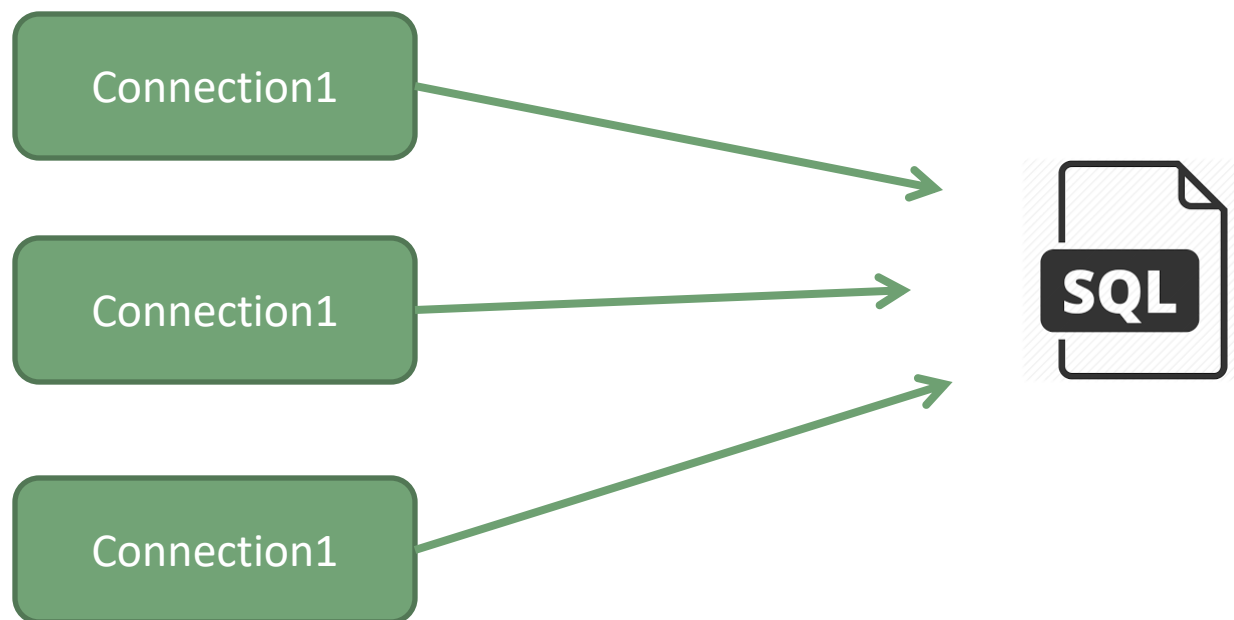
```
System.out.println("Using enum singleton");  
System.out.println("next vehicle: " + SerialNumberGenerator.INSTANCE.getNextSerial());  
System.out.println("next vehicle: " + SerialNumberGenerator.INSTANCE.getNextSerial());  
System.out.println("next engine: " + SerialNumberGenerator.INSTANCE.getNextSerial());
```

Singleton SQLite

Singleton com Sql Lite. Múltiplos acessos através do Connection como inserir, remover e recuperar.

SQLite é um é um banco de dados em arquivo.

Como o banco de dados é em arquivo se eu tentar acessar ele simultaneamente ele vai dar problema de concorrência.



Código Java - Database Example



```

public class MainSqlLite
{
    public static void main( String args[] )
    {
        usingThreads();
        usingThreads();
        usingThreads();
        usingThreads();
        usingThreads();
        usingThreads();
        usingThreads();
        usingThreads();
    }

    public static void usingThreads() {

        Runnable hello = new ThreadSql();
        Thread thread1 = new Thread(hello);
        thread1.start();

    }
}

```



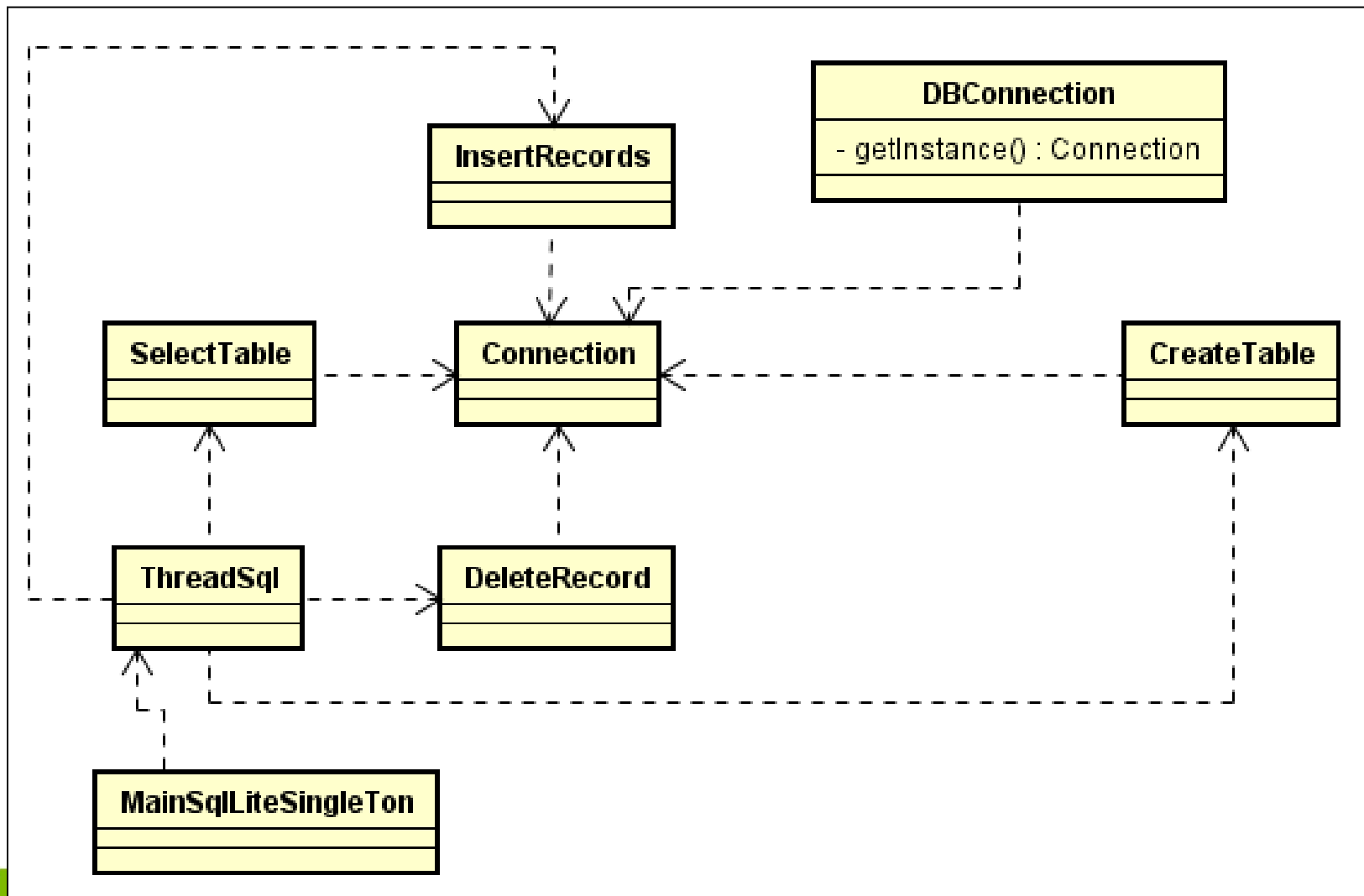
```
public class ThreadSql implements Runnable
{
    public void run()
    {
        InsertRecords insert = new InsertRecords();
        insert.insertRecords();
        SelectTable select = new SelectTable();
        select.selectTable();
        DeleteRecord delete = new DeleteRecord();
        delete.deleteRecord();
    }
}
```



```
public class DeleteRecord {  
  
    public synchronized void deleteRecord() {  
        Connection c = null;  
        Statement stmt = null;  
  
        try {  
            Class.forName("org.sqlite.JDBC");  
            c = DriverManager.getConnection("jdbc:sqlite:test.db");  
            //c.setAutoCommit(false);  
            System.out.println("Opened database successfully");  
  
            stmt = c.createStatement();  
            String sql = "DELETE from COMPANY where ID=2;";  
            stmt.executeUpdate(sql);  
            c.commit();  
  
            ResultSet rs = stmt.executeQuery("SELECT * FROM COMPANY;");  
            while (rs.next()) {  
                int id = rs.getInt("id");  
                String name = rs.getString("name");  
                int age = rs.getInt("age");  
            }  
        }  
    }  
}
```

GetConnection!





```
public class DeleteRecord {
```

```
    public void deleteRecord() {
```

```
        Connection c = null;
```

```
        Statement stmt = null;
```

```
        try {
```

```
            c = DBConnection.getInstance();
```

```
            System.out.println("Remove");
```

```
            stmt = c.createStatement();
```

```
            String sql = "DELETE from COMPANY where ID=2;";
```

```
            stmt.executeUpdate(sql);
```

```
            //c.commit();
```

```
            ResultSet rs = stmt.executeQuery("SELECT * FROM COMPANY;");
```

```
            while (rs.next()) {
```

```
                int id = rs.getInt("id");
```

Usando Singleton pattern



SingleTon

```
public class DBConnection {  
  
    private static Connection instance = null;  
  
    private DBConnection() {  
    }  
  
    public static synchronized Connection getInstance() {  
        if (instance == null) {  
            try {  
                Connection c = null;  
                Class.forName("org.sqlite.JDBC");  
                c = DriverManager.getConnection("jdbc:sqlite:test.db");  
                c.setAutoCommit(true);  
                instance = c;  
            } catch (Exception e) {  
                System.err.println(e.getClass().getName() + ": " + e.getMessage());  
            }  
        }  
        return instance;  
    }  
}
```

Licença para Uso e Distribuição

- Este material está disponível para uso não-comercial e pode ser derivado e/ou distribuído, desde que utilizando uma licença equivalente.

- Maiores informações: <http://creativecommons.org/licenses/by-nc-sa/2.5/deed.pt>

- Você pode copiar, distribuir, exibir e executar a obra, além de criar obras derivadas, sob as seguintes condições: (a) você deve dar crédito ao autor original, da forma especificada pelo autor ou licenciante; (b) você não pode utilizar esta obra com finalidades comerciais; (c) Se você alterar, transformar, ou criar outra obra com base nesta, você somente poderá distribuir a obra resultante sob uma licença idêntica a esta.

Referências

- Freeman, Freeman. **Use a Cabeça! Padrões de Projeto. Segunda Edição.**2009. Capítulo 4 e 5
- Gamma, E., Helm, R. JonhSon , Ralph e Vlissides, J. **Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos.** Capitulo 3: Padrões de Criação.
- Fox, Armando; Patterson, David. **Engineering Software as a Service: An Agile Approach Using Cloud Computing.** Strawberry Canyon LLC. 2014
- Bevis, Tony. **Java Design Pattern Essentials.** Ability First Limited. 2012.
- SourceMaking, acessado:
<https://sourcemaking.com/refactoring/bad-smells-in-code>