



# Padrão Factory Method

---

Prof<sup>a</sup>. Rachel Reis  
rachel@inf.ufpr.br



# Padrões – Factory Method

- Exemplos:

Criação	Estrutural	Comportamental
<ul style="list-style-type: none"><li>• Abstract factory</li><li>• Builder</li><li>• <b>Factory Method</b></li><li>• Prototype</li><li>• Singleton</li></ul>	<ul style="list-style-type: none"><li>• Adapter</li><li>• Bridge</li><li>• Composite</li><li>• Decorator</li><li>• Façade</li><li>• Flyweight</li><li>• Proxy</li></ul>	<ul style="list-style-type: none"><li>• Chain of responsibility</li><li>• Command</li><li>• Interpreter</li><li>• Iterator</li><li>• Mediator</li><li>• Memento</li><li>• Observer</li><li>• Etc.</li></ul>



# Sobre o Factory Method

---

- É um padrão de projeto de criação (lida com a criação de objetos)
- Oculta a lógica de instanciação do código cliente (desacopla o código que cria o objeto do código que utiliza o objeto).
- Utiliza os conceitos de interface, classe abstrata e herança.



# Sobre o Factory Method

---

- Oferece flexibilidade ao código permitindo a criação de novas *factories* sem a necessidade de alterar o código já escrito.
- Pode usar parâmetros para determinar o tipo dos objetos a serem criados ou receber esses parâmetros para repassar aos objetos que estão sendo criados.

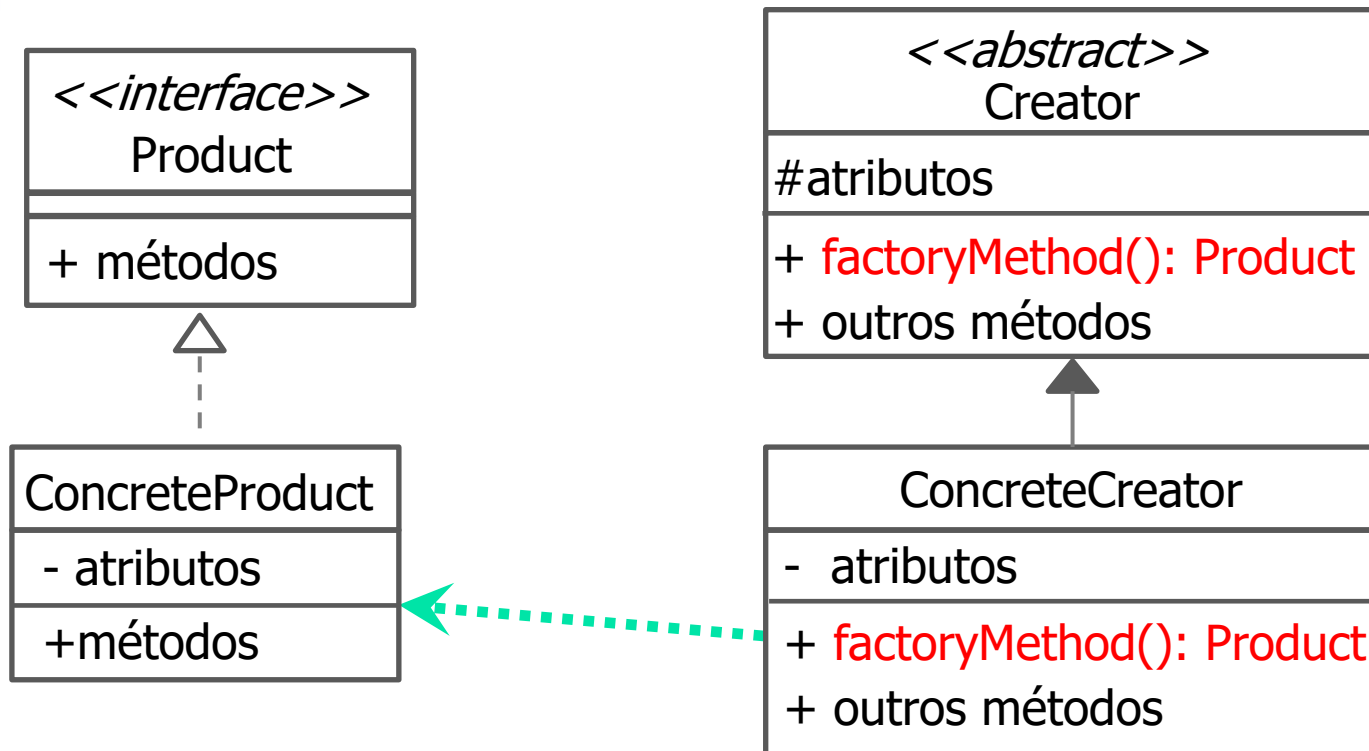


# Factory Method - Intenção

---

- Definir uma interface para criar um objeto, mas deixar as subclasses decidirem que classe instanciar.
- Permite a uma classe adiar a instanciação para as subclasses.

# Factory Method - Estrutura

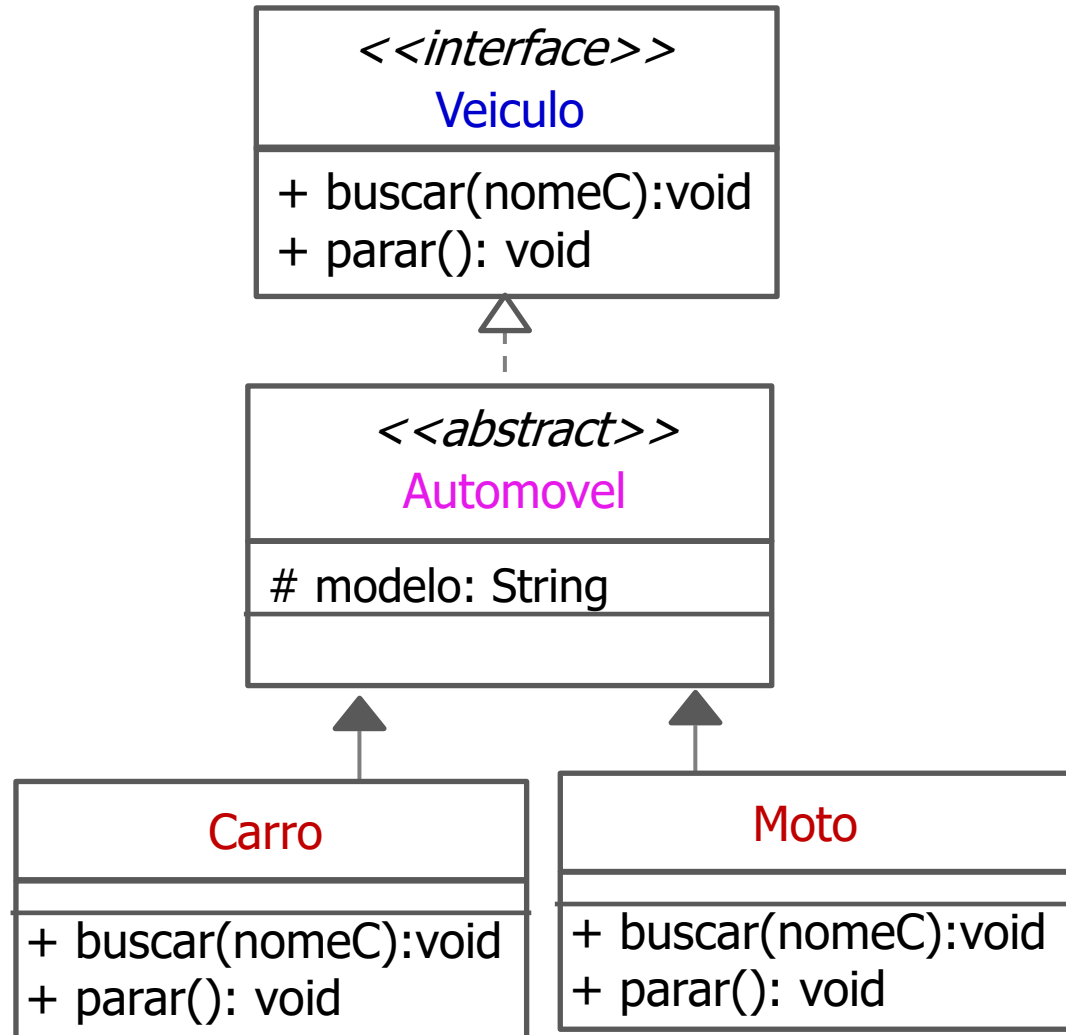
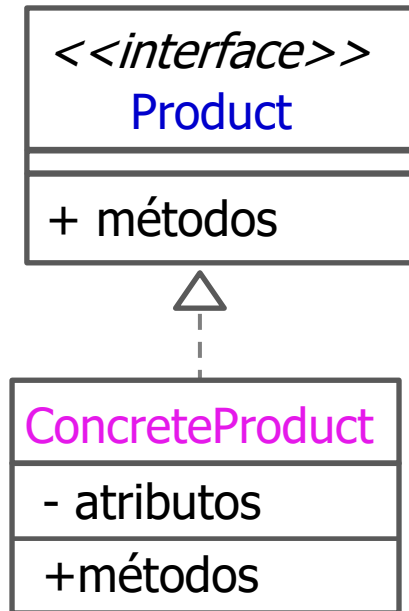




# Exemplo 1: Aplicação Uber

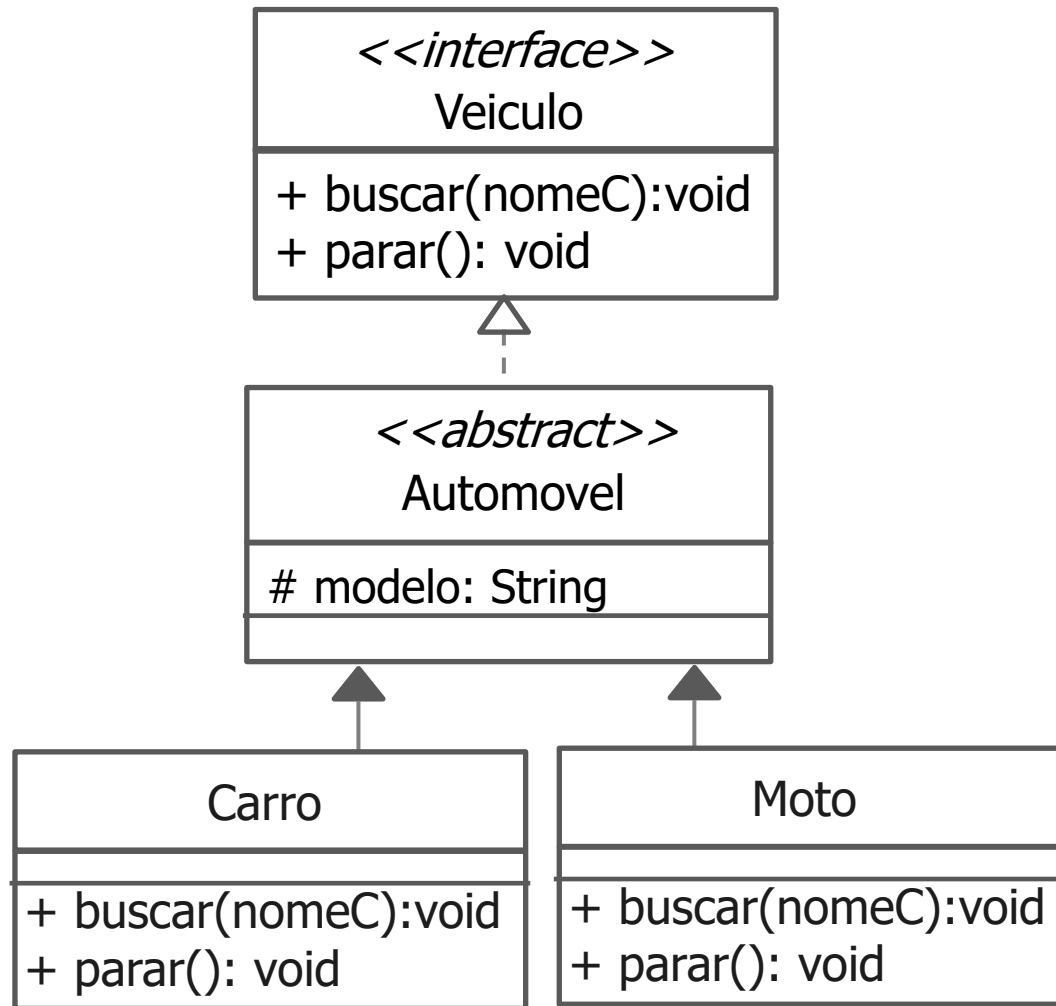
---

- Desenvolva uma aplicação Uber que ofereça o serviço de transporte de pessoas (similar ao taxi) em carros e motos.





# Implementar a estrutura abaixo



```
public interface Veiculo
{
    public abstract void buscar(String nomeC);
    public abstract void parar();
}
```

Veiculo.java

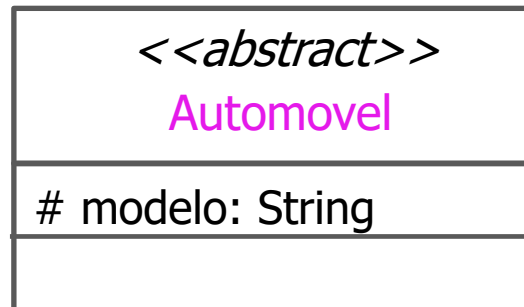
<<interface>>

**Veiculo**

+ buscar(nomeC):void  
+ parar(): void

```
public abstract class Automovel implements Veiculo{  
    protected String modelo;  
  
    public Automovel(String modelo){  
        this.setModelo(modelo);  
    }  
  
    // Implementar métodos get/set  
  
}
```

Automovel.java



```
public class Carro extends Automovel  
{
```

```
    public Carro(String modelo){  
        super(modelo);  
    }
```

```
    public void buscar(String nomeC){  
        System.out.println(this.modelo + " buscando " + nomeC);  
    }  
    public void parar(){  
        System.out.println(this.modelo + " parado");  
    }  
}
```

<<abstract>>  
Automovel



Carro

+buscar(nomeC):void  
+parar(): void

Carro.java

```
public class Moto extends Automovel  
{
```

```
    public Moto(String modelo){  
        super(modelo);  
    }
```

```
    public void buscar(String nomeC){  
        System.out.println(this.modelo + " buscando " + nomeC);  
    }  
    public void parar(){  
        System.out.println(this.modelo + " parada");  
    }  
}
```

<<abstract>>  
Automovel



Moto

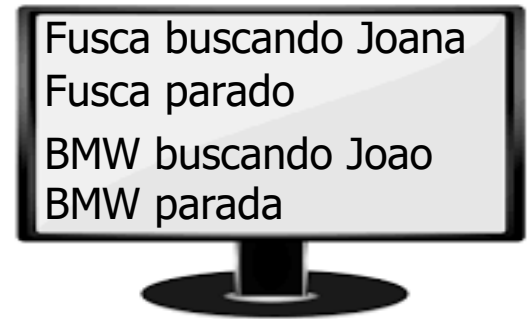
+buscar(nomeC):void  
+parar(): void

Moto.java

```
public class Principal
{
    public static void main(String []args)
    {
        // Código sem usar o padrão Method Factory
        Veiculo fusca = new Carro("Fusca");
        fusca.buscar("Joana");
        fusca.parar();

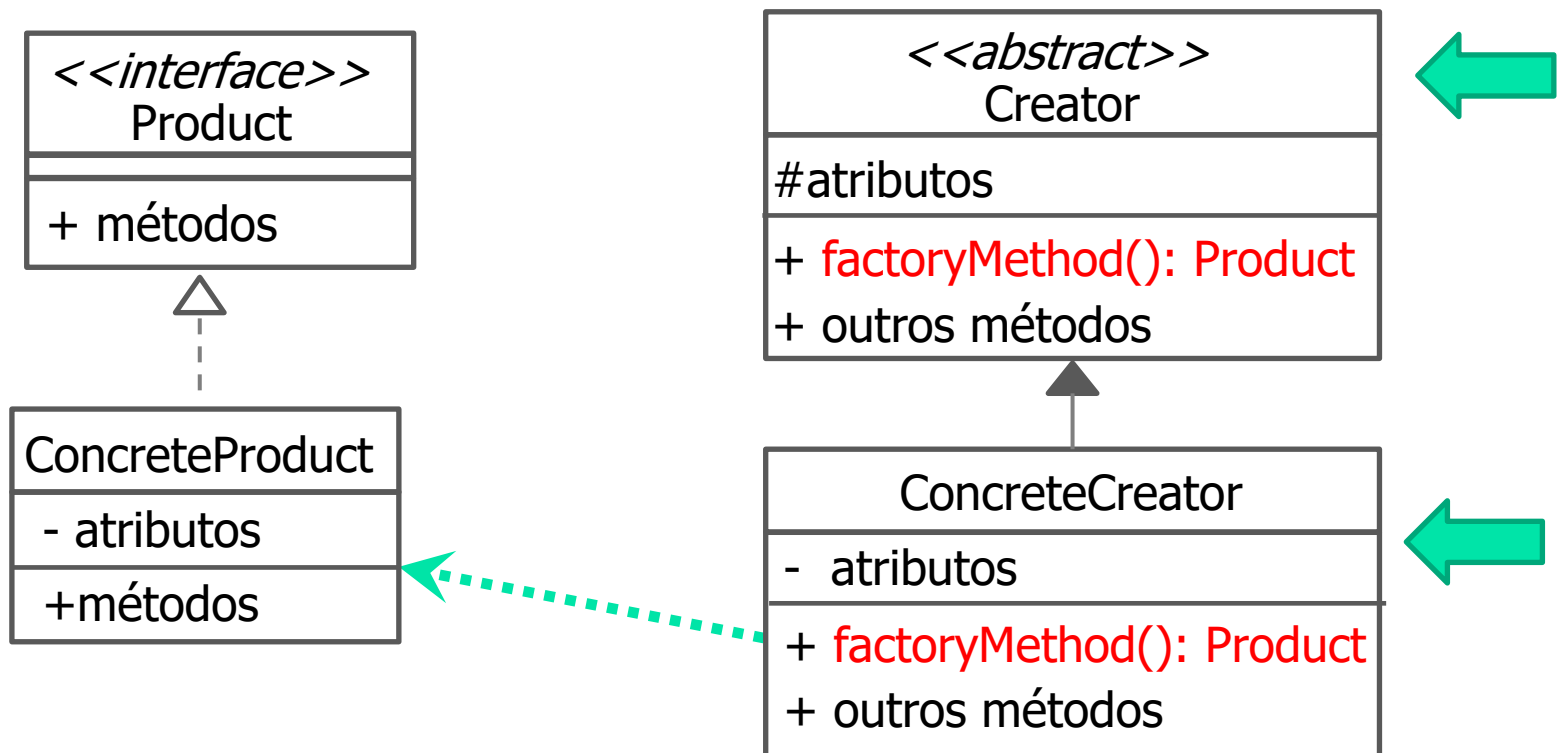
        Veiculo honda = new Moto("BMW");
        honda.buscar("João");
        honda.parar();

    }
}
```



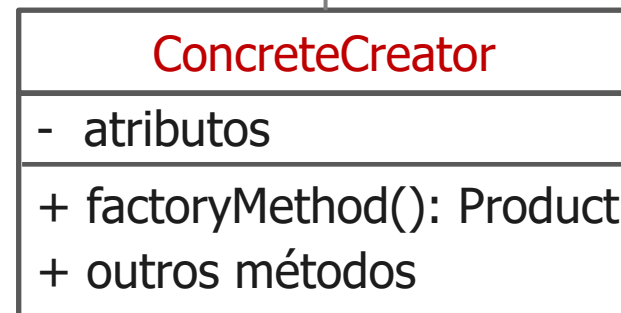
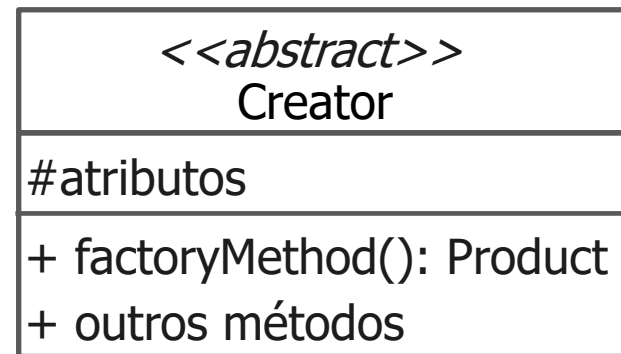
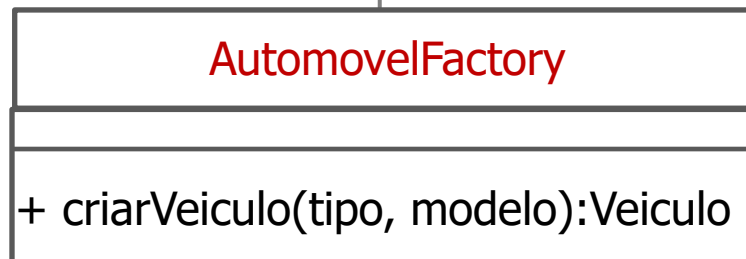
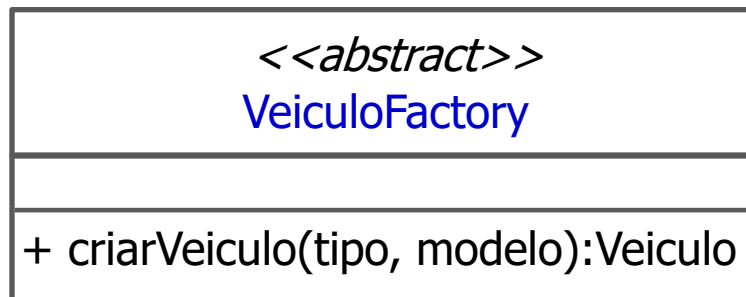
O que acontece se alterarmos o nome das classes Carro e Moto para Carro1 e Moto1?

- Para solucionar o problema anterior vamos usar o padrão de projeto Method Factory





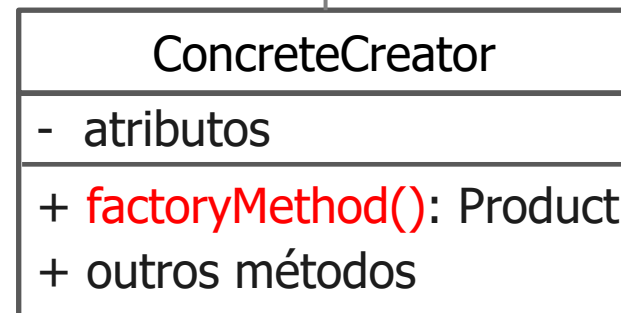
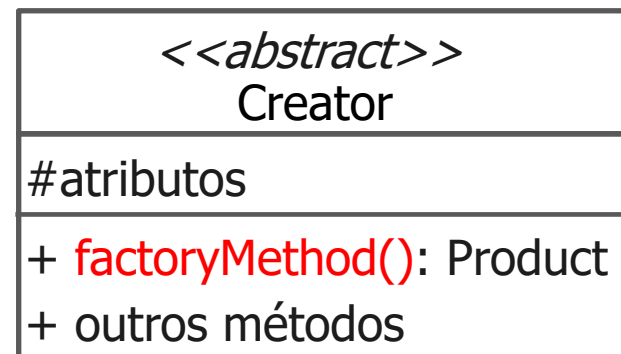
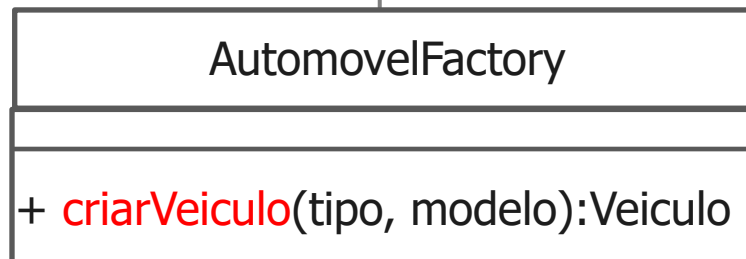
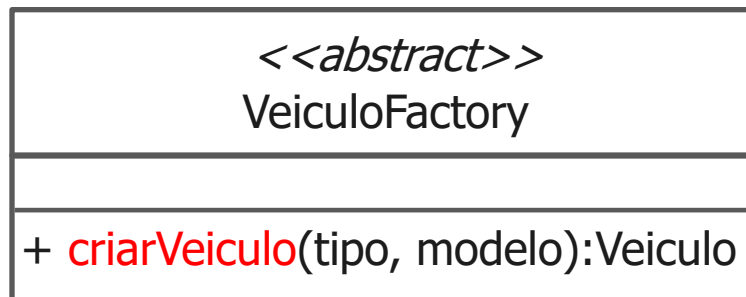
# Exemplo: Aplicação - Uber



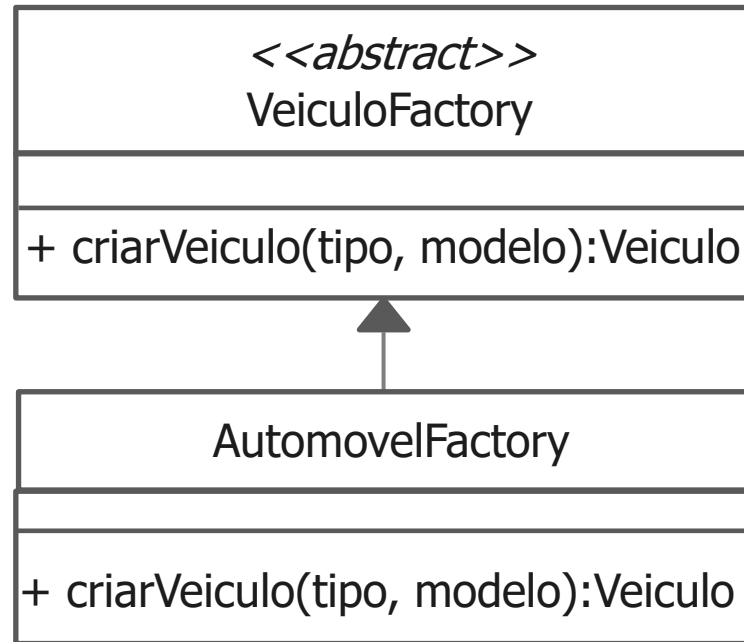




# Exemplo: Aplicação - Uber

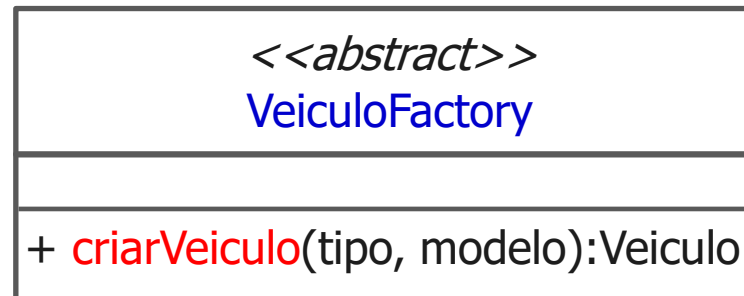


# Implementar a estrutura abaixo



```
public abstract class VeiculoFactory
{
    public abstract Veiculo criarVeiculo (String tipo, String modelo);
}
```

VeiculoFactory.java



```
public class AutomovelFactory extends VeiculoFactory
{
    public Veiculo criarVeiculo (String tipo, String modelo)
    {

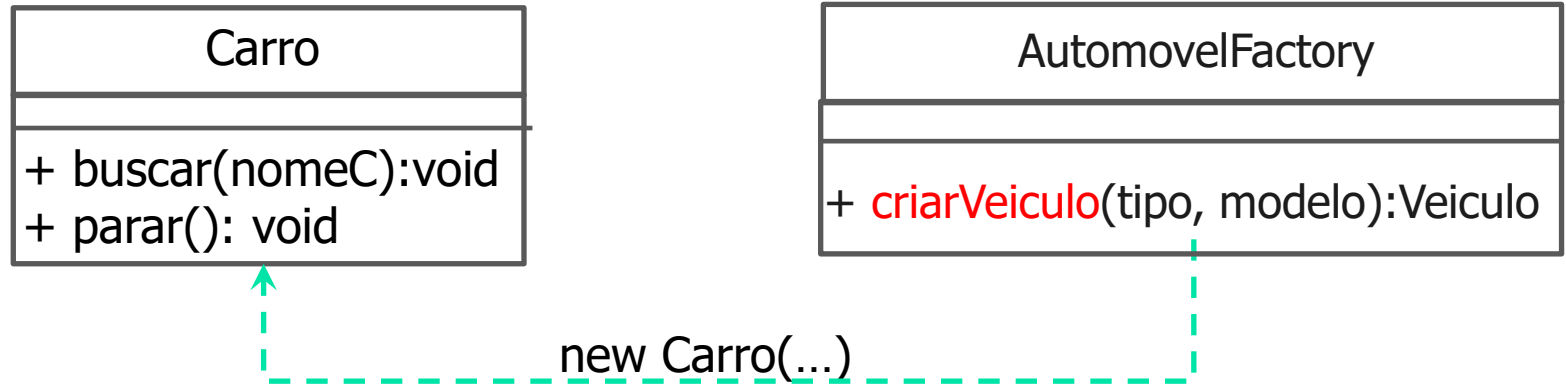
    }
}
```

AutomovelFactory.java

AutomovelFactory

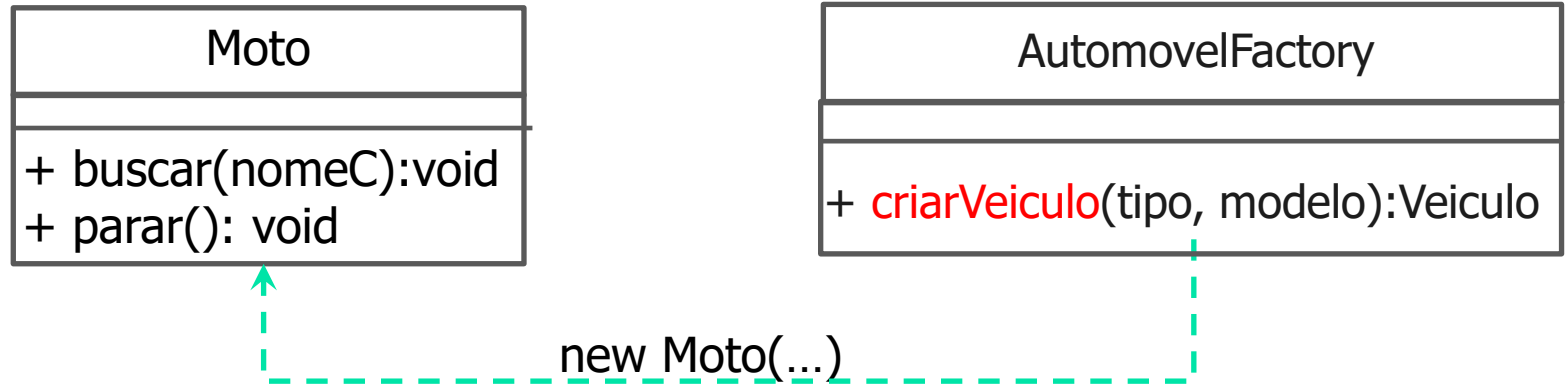
+ criarVeiculo(tipo, modelo):Veiculo


```
public class AutomovelFactory extends VeiculoFactory
{
    public Veiculo criarVeiculo (String tipo, String modelo){
        ➡ if(tipo == "carro")
            return new Carro(modelo);
    }
}
```



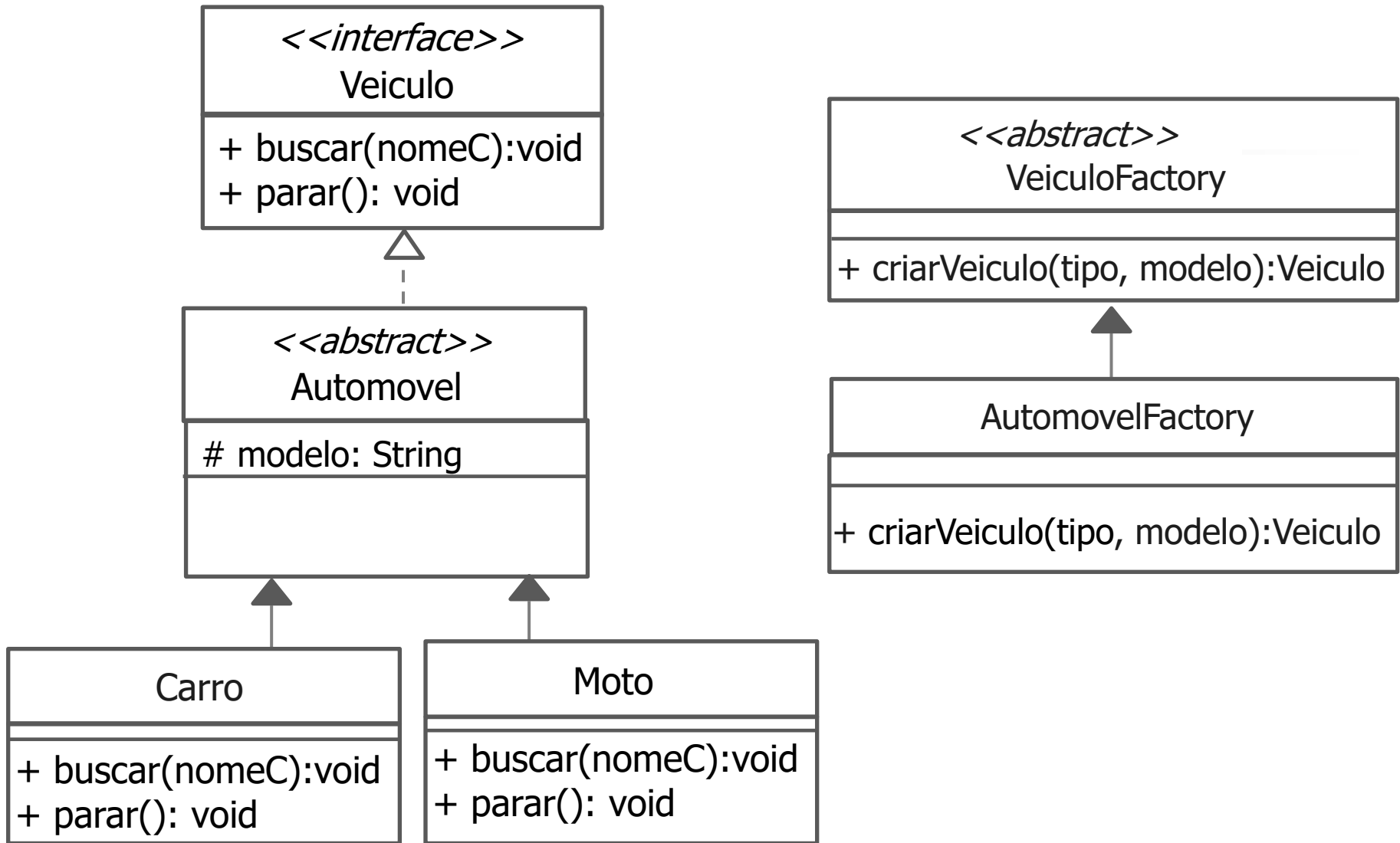
```
public class AutomovelFactory extends VeiculoFactory
{
    public Veiculo criarVeiculo (String tipo, String modelo){
        if(tipo == "carro")
            return new Carro(modelo);
        ➡ else if(tipo == "moto")
            return new Moto(modelo);

    }
}
```



```
public class AutomovelFactory extends VeiculoFactory
{
    public Veiculo criarVeiculo (String tipo, String modelo){
        if(tipo == "carro")
            return new Carro(modelo);
        else if(tipo == "moto")
            return new Moto(modelo);
         else
            return null;
        }
    }
}
```

# Estrutura completa: Aplicação - Uber





```
public class Principal
```

```
{
```

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

```
    {
```

```
        // Código usando o padrão Method Factory
```

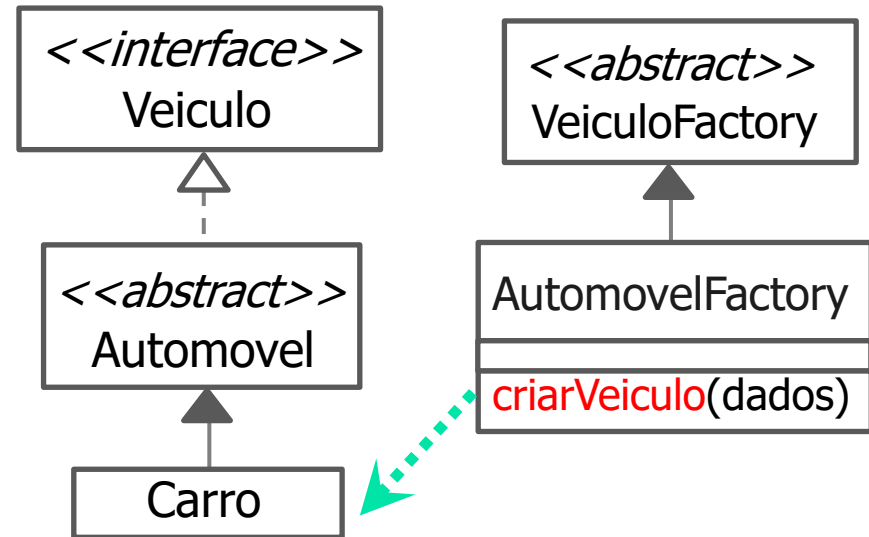
```
        VeiculoFactory autoF = new AutomovelFactory();
```

```
        Veiculo v = autoF.criarVeiculo("carro", "Fusca");
```

```
    }
```

```
}
```

```
}
```



```
public class Principal
```

```
{
```

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

```
    {
```

```
        // Código usando o padrão Method Factory
```

```
        VeiculoFactory autoF = new AutomovelFactory();
```

```
        Veiculo v = autoF.criarVeiculo("carro", "Fusca");
```

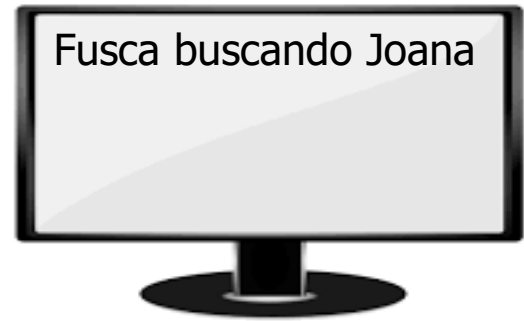
```
        if (v != null){
```

```
            v.buscar("Joana");
```

```
        }
```

```
    }
```

```
}
```



```
<<interface>>  
Veiculo
```



```
<<abstract>>  
Automovel
```



```
Carro  
  
buscar(...)
```

```
<<abstract>>  
VeiculoFactory
```



```
AutomovelFactory  
  
criarVeiculo(dados)
```



```
public class Principal
```

```
{
```

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

```
    {
```

```
        // Código usando o padrão Method Factory
```

```
        VeiculoFactory autoF = new AutomovelFactory();
```

```
        Veiculo v = autoF.criarVeiculo("carro", "Fusca");
```

```
        if (v != null){
```

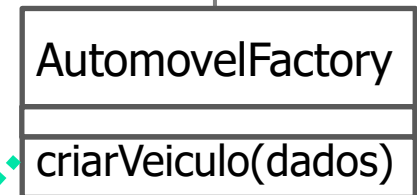
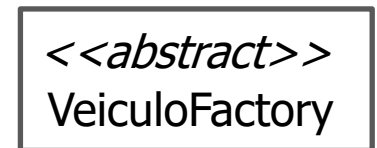
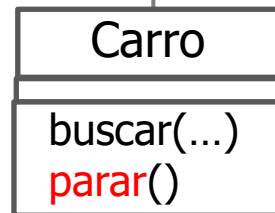
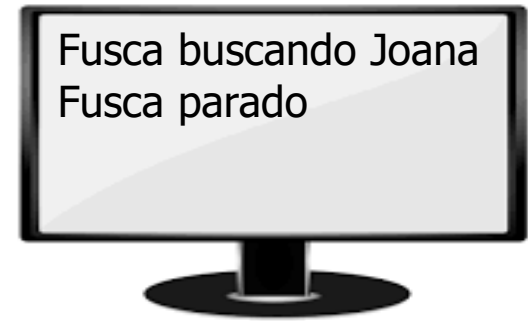
```
            v.buscar("Joana");
```

```
            v.parar();
```

```
        }
```

```
    }
```

```
}
```



```
public class Principal
{
    public static void main(String []args)
    {
        // Código usando o padrão Method Factory
        VeiculoFactory autoF = new AutomovelFactory();
        Veiculo v = autoF.criarVeiculo("carro", "Fusca");

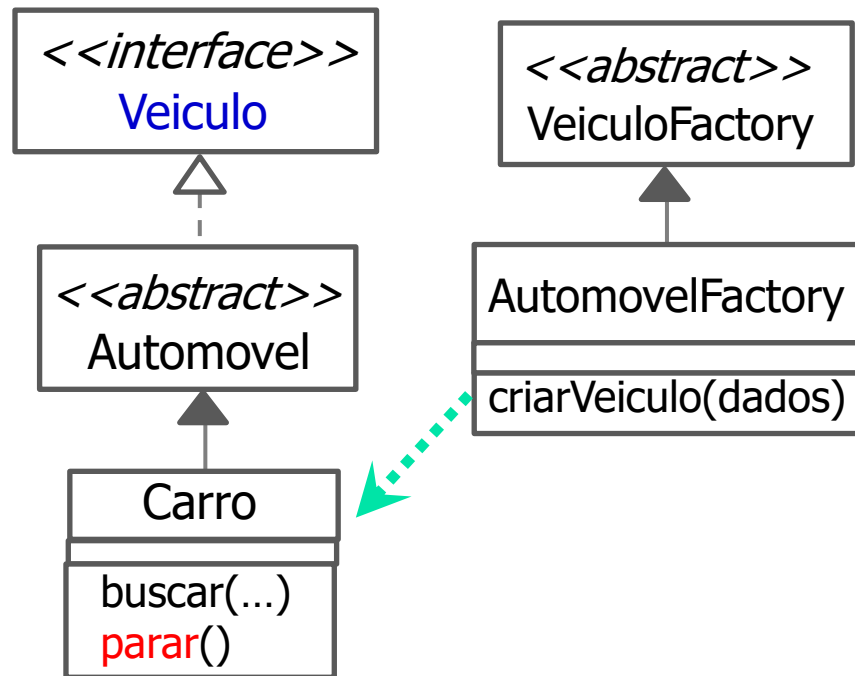
        if (v != null){
            v.buscar("Joana");
            v.parar();
        }
    }
}
```

Altere o código para que uma  
moto BMW busque João

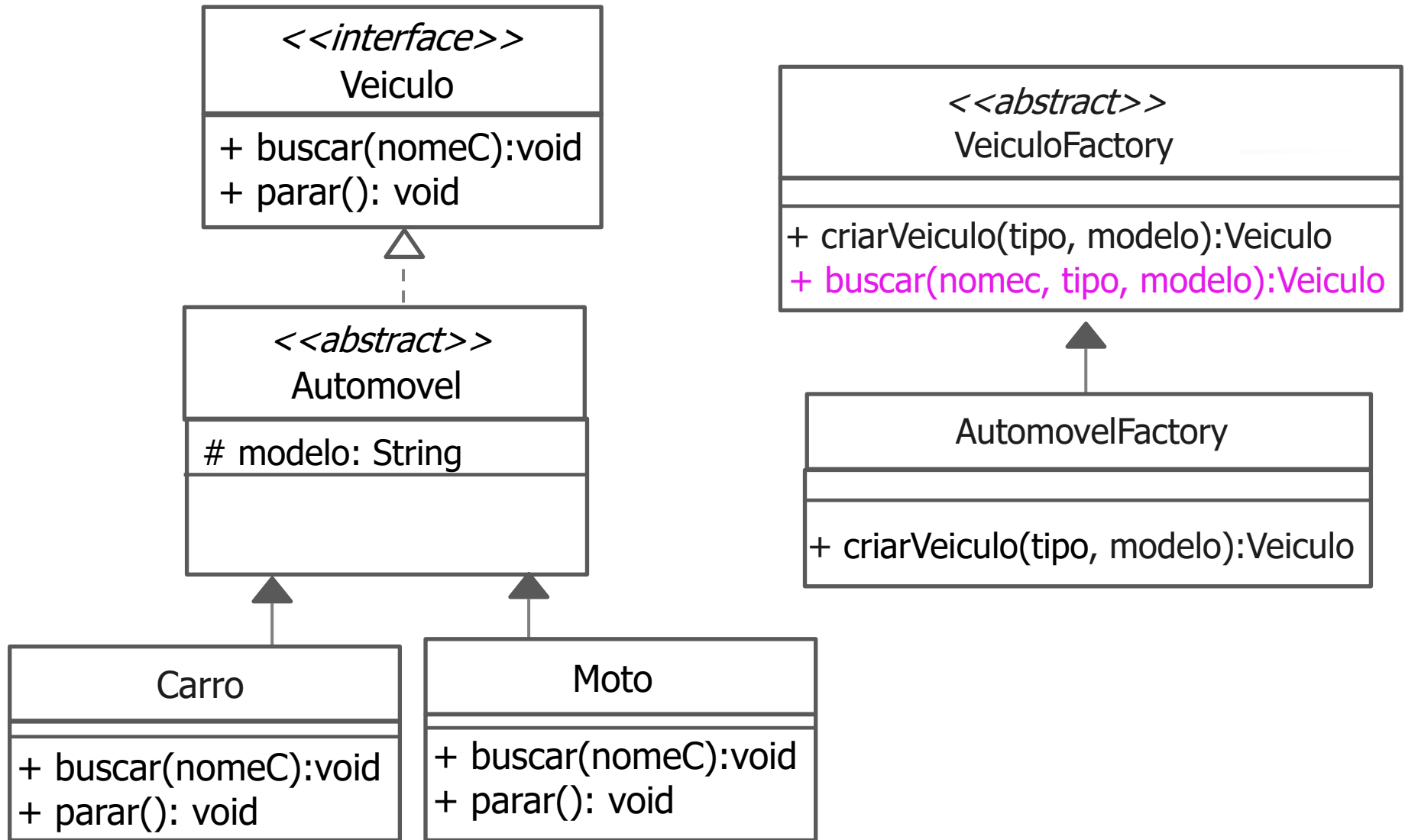
```
public class Principal{  
    public static void main(String []args)  
    {  
        // Código sem usar o padrão Method Factory  
        Veiculo fusca = new Carro("Fusca");  
        fusca.buscar("Joana");  
        fusca.parar();  
  
        // Código usando o padrão Method Factory  
        VeiculoFactory autoF = new AutomovelFactory();  
        Veiculo v = autoF.criarVeiculo("carro", "Fusca");  
        fusca.buscar("Joana");  
        fusca.parar();  
    }  
}
```

- Revendo a **intenção**: definir uma interface para criar um objeto, mas deixar as subclasses decidirem que classe instanciar.
- Permite a uma classe adiar a instanciação para as subclasses.

```
VeiculoFactory autoF = new AutomovelFactory();  
Veiculo v = autoF.criarVeiculo("carro", "Fusca");
```



- Podemos ter outros métodos na classe VeiculoFactory



```
public abstract class VeiculoFactory
{
    public abstract Veiculo criarVeiculo (String tipo, String modelo);

    public Veiculo buscar(String nomeC, String tipo, String modelo)
    {
        Veiculo v = this.criarVeiculo(tipo, modelo);
        v.buscar(nomeC);
        return v;
    }
}
```

<<abstract>>

VeiculoFactory

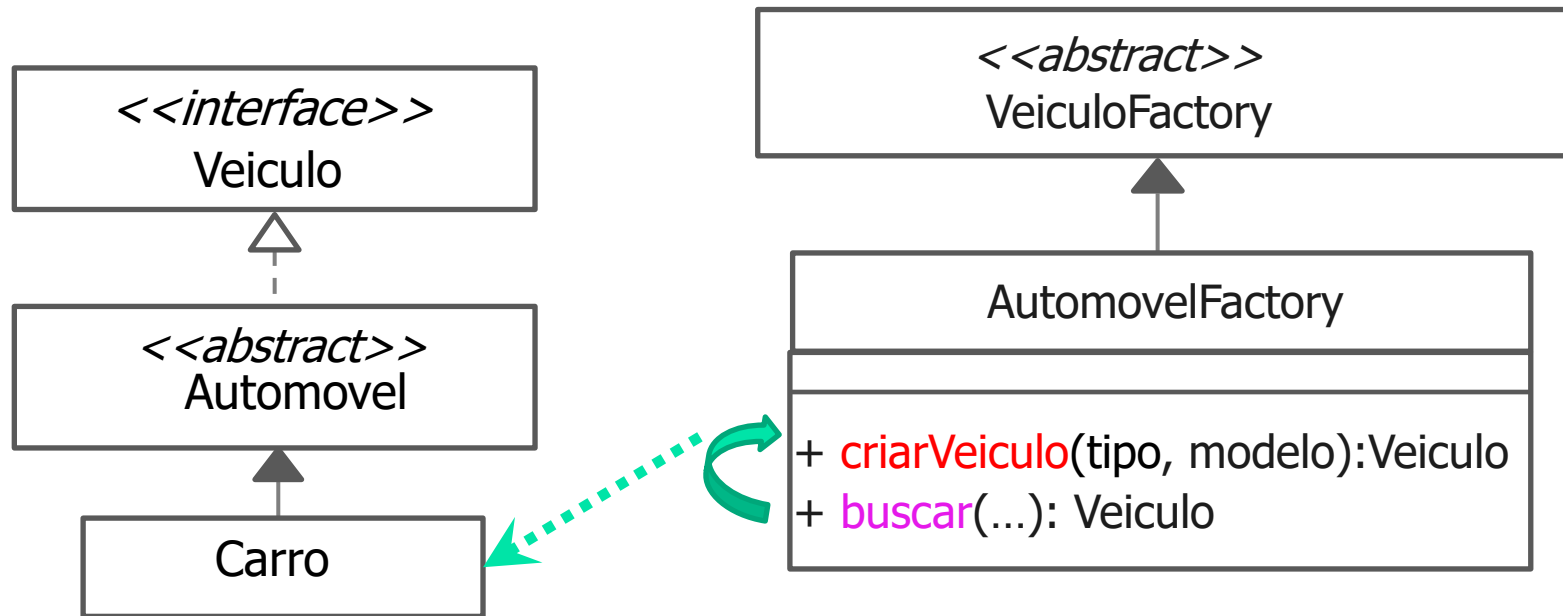
+ criarVeiculo(tipo, modelo):Veiculo  
+ buscar(nomeC, tipo, modelo):Veiculo



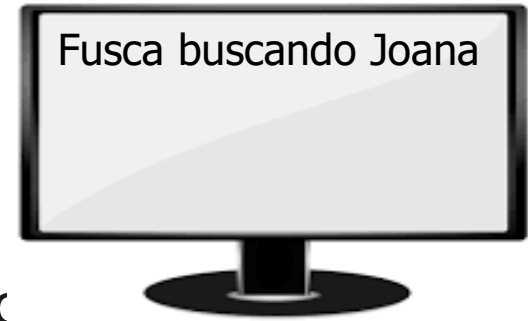
```

public class Principal{
    public static void main(String []args)
    {
        // Código usando o padrão Method Factory
        VeiculoFactory autoF1 = new AutomovelFactory();
        Veiculo v1 = autoF1.buscar("Joana", "carro", "Fusca");
    }
}

```



```
public class Principal{  
    public static void main(String []args)  
    {  
        // Código usando o padrão Method Factory  
        VeiculoFactory autoF1 = new AutomovelFacto  
        Veiculo v1 = autoF1.buscar("Joana", "carro", "Fusca");  
    }  
}
```



Como fazer para chamar  
o método parar()?



## Para praticar...

---

- Desenvolva uma aplicação para uma empresa com o objetivo de produzir diferentes tipos de bolos. Utilize o padrão *Factory Method*.
- Pense em uma aplicação em que o padrão *Factory Method* poderia ser usado. Em seguida, elabore a estrutura para este padrão.



# Referência

---

- Gamma, E., Helm, R., Johnson, R., Vlissides, J. (2007). Padrões de projeto: soluções reutilizáveis de software orientado a objetos. Porto Alegre, RS: Bookman.