Design Patterns

Considere o sistema de um estacionamento que precisa utilizar diversos critérios para calcular o valor que deve ser cobrado de seus clientes.
Para um veículo de passeio, o valor deve ser calculado como R$2,00 por hora.
Porém, caso o tempo seja maior do que 12 horas, será cobrada uma taxa diária, e caso o número de dias for maior que 15 dias, será cobrada uma mensalidade.
Existem também regras diferentes para caminhões, que dependem do número de eixos e do valor da carga carregada, e para veículos para muitos passageiros, como ônibus e vans.
O código a seguir apresenta um exemplo de como isto estava desenvolvido.

```java
public class ContaEstacionamento {

    private Veiculo veiculo;
    private long inicio, fim;

    public double valorConta() {
        long atual = (fim==0) ? System.currentTimeMillis():fim;
        long periodo = inicio - atual;
        if (veiculo instanceof Passeio) {
            if (periodo < 12 * HORA) {
                return 2.0 * Math.ceil(periodo / HORA);
            }else if (periodo > 12 * HORA && periodo < 15*DIA) {
                return 26.0 * Math.ceil(periodo / DIA);
            } else {
                return 300.0 * Math.ceil(periodo / MES);
            }
        } else if (veiculo instanceof Carga) {
            // outras regras para veículos de carga

            }
            // outras regras para outros tipos de veículo
        }

    }
```

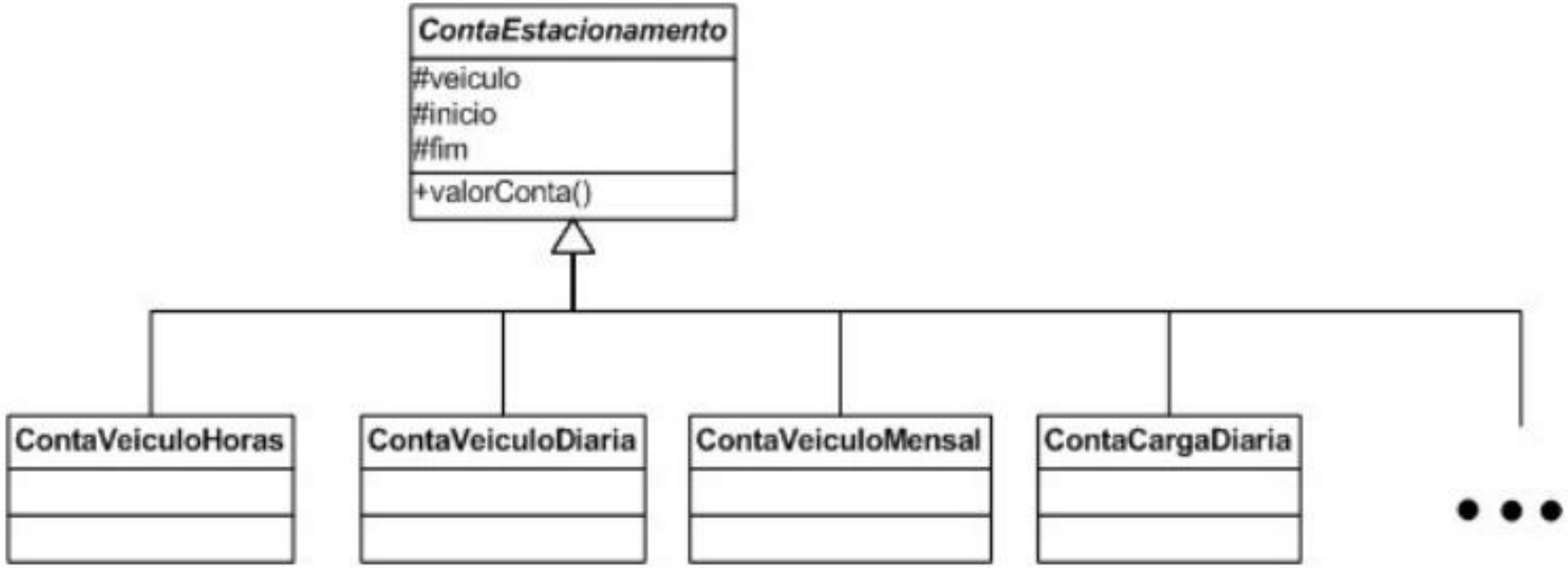
Como é possível observar, o código usado para calcular os diversos condicionais é complicado de se entender. Apesar de apenas parte da implementação do método ter sido apresentada, pode-se perceber que a solução utilizada faz com que o método valorConta() seja bem grande.
As instâncias da classe ContaEstacionamento relacionadas com os veículos que estão no momento estacionados são exibidas para o operador e têm seu tempo atualizado periodicamente.

Até então, o próprio desenvolvedor sabia que o código estava ruim, mas como o código estava funcionando, preferiu deixar da forma que está. Porém, o software começou a ser vendido para outras empresas e outras regras precisariam ser incluídas.

Alguns municípios possuem leis específicas a respeito do intervalo de tempo para o qual um estacionamento deve definir sua tarifa. Além disso, diferentes empresas podem possuir diferentes critérios para cobrar o serviço de seus clientes.

A solução da forma como está não vai escalar para um número grande de regras! O código que já não estava bom poderia crescer de uma forma descontrolada e se tornar não gerenciável.
O desenvolvedor sabia que precisaria refatorar esse código para uma solução diferente. O que ele poderia fazer?

A primeira ideia que o desenvolvedor pensou foi na utilização de herança para tentar dividir a lógica. Sua ideia era criar uma superclasse em que o cálculo do valor da conta seria representado por um método abstrato, o qual seria implementado pelas subclasses Usando herança, a figura a seguir apresenta como seria essa solução.



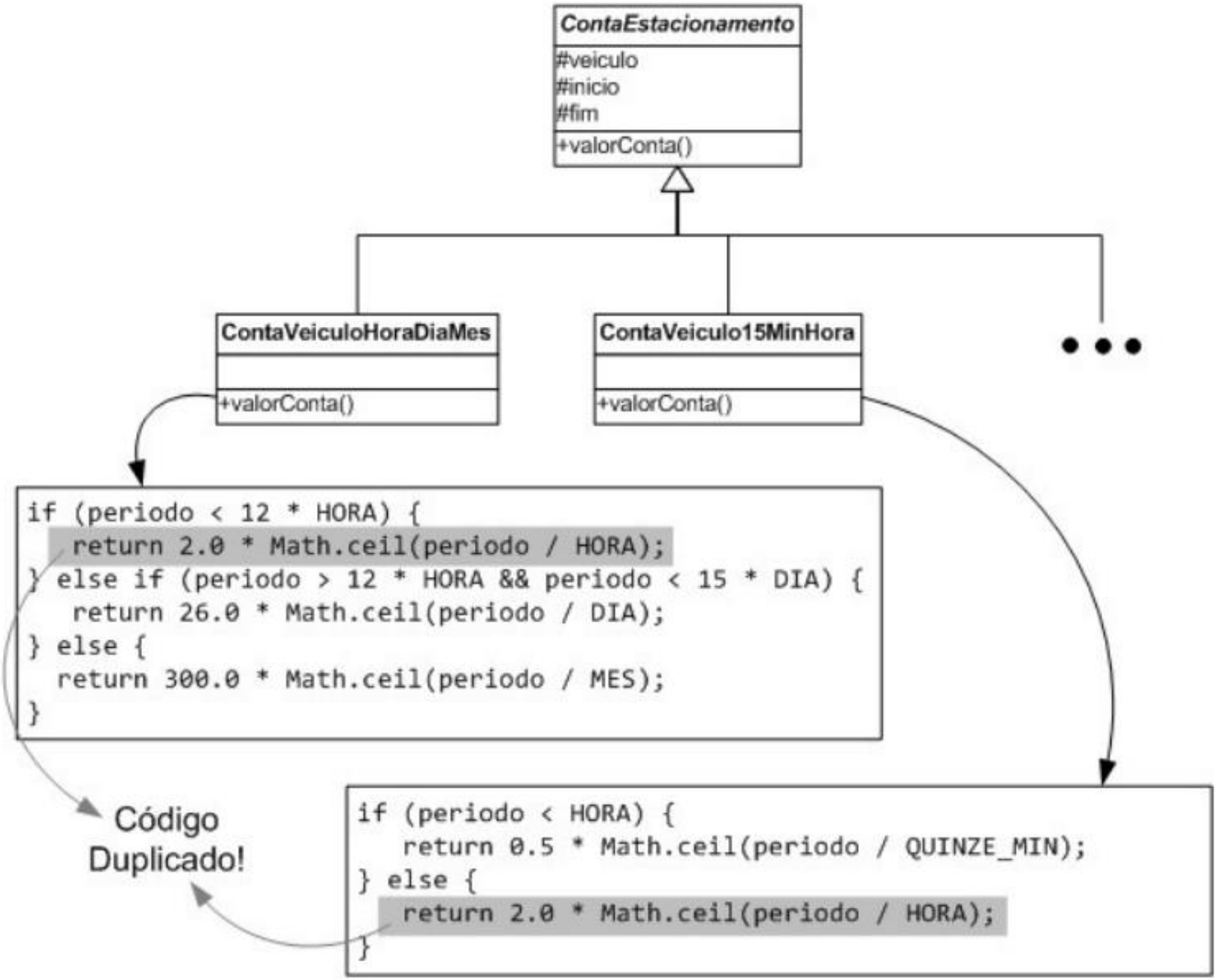
Um dos problemas dessa solução é a explosão de subclasses que vai acontecer, devido às várias possibilidades de implementação. Outra questão é que, utilizando herança, depois não é possível alterar o comportamento, uma vez que a classe foi instanciada.

Por exemplo, depois de 12 horas que um veículo estiver estacionado, o comportamento do cálculo da tarifa deve ser alterado da abordagem por hora para a abordagem por dia. Quando se cria o objeto como sendo de uma classe, para mudar o comportamento que ela implementa, é preciso criar uma nova instância de outra classe. Isso é algo indesejável, pois a mudança deveria acontecer dentro da própria classe!

A segunda solução que o desenvolvedor pensou foi utilizar a herança, mas com uma granularidade diferente. Sua ideia era que cada subclasse possuísse o código relacionado a uma abordagem de cobrança para um tipo de veículo. Por exemplo, no caso anterior, seria apenas uma subclasse para veículos de passeio e ela conteria os condicionais de tempo necessários. Dessa forma, a mesma instância seria capaz de fazer o cálculo.

Ao começar a seguir essa ideia, o desenvolvedor percebeu que nas subclasses criadas ocorria bastante duplicação de código. Um dos motivos é que, a cada pequena diferença na abordagem, uma nova subclasse precisaria ser criada. Se a mudança fosse pequena, o resto do código comum precisaria ser duplicado.

A figura seguinte ilustra essa questão. Uma das classes considera a tarifa de veículo de passeio como apresentado anteriormente, e a outra considera que a primeira hora precisa ser dividida em períodos de 15 minutos (uma lei que existe no município de Juiz de Fora) e, em seguida, faz a cobrança por hora como mostrada anteriormente. Observe que se houvesse uma classe com os períodos de 15 minutos mais a cobrança por dia e por mês, mais código ainda seria duplicado.

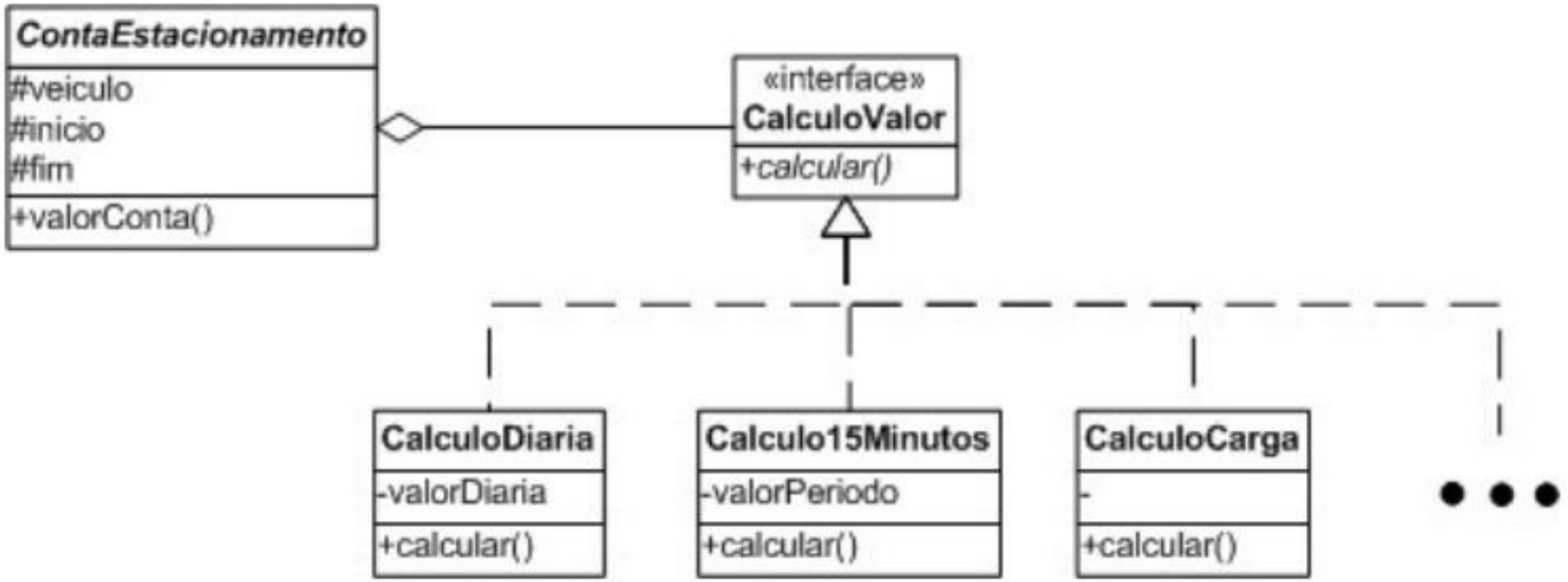

ContaEstacionamento
#veiculo
#inicio
#fim
+valorConta()
ContaVeiculoHoraDiaMes
+valorConta()
ContaVeiculo15MinHora
+valorConta()
if (periodo < 12 * HORA) {
return 2.0 * Math.ceil(periodo / HORA);
} else if (periodo > 12 * HORA && periodo < 15 * DIA) {
return 26.0 * Math.ceil(periodo / DIA);
} else {
return 300.0 * Math.ceil(periodo / MES);
}
Código
Duplicado!
if (periodo < HORA) {
return 0.5 * Math.ceil(periodo / QUINZE_MIN);
} else {
return 2.0 * Math.ceil(periodo / HORA);
}

Mas será que é possível ter uma forma de manter a mesma instância de ContaEstacionamento sem precisar duplicar código?

Ao pensar melhor, o desenvolvedor decide que a herança não é a melhor abordagem para resolver o problema. Ele precisa de uma solução que permita que diferentes algoritmos de cálculo de tarifa possam ser usados pela classe. Adicionalmente, é desejável que não haja duplicação de código e que o mesmo algoritmo de cálculo possa ser utilizado para diferentes empresas. Além disso, uma classe deve poder iniciar a execução com um algoritmo e este ser trocado posteriormente.

Uma solução que se encaixa nos requisitos descritos é a classe ContaEstacionamento delegar a lógica de cálculo para a instância de uma classe que a compõe. Dessa forma, para trocar o comportamento do cálculo do valor do estacionamento, basta inserir outra instância dessa classe, com outra estratégia de calculo, em ContaEstacionamento . Essa classe também poderia ser parametrizada e ser reutilizada no contexto de diversas empresas. A figura a seguir mostra o diagrama que representa essa solução.

ContaEstacionamento delega a lógica de cálculo para a instância de uma classe que a compõe. Dessa forma, para trocar o comportamento do cálculo do valor do estacionamento, basta inserir/atribuir outra instância (com outra estratégia de cálculo) dessa classe em ContaEstacionamento .



Agora, apresentamos a classe ContaEstacionamento delegando para uma classe que compõe o calculo do valor do estacionamento. A interface CalculoValor abstrai o algoritmo de cálculo da tarifa. Observe que existe um método que permite que o atributo calculo seja alterado, permitindo a mudança desse algoritmo depois que o objeto foi criado.

```
public class ContaEstacionamento {

    private CalculoValor calculo;

    private Veiculo veiculo;
    private long inicio;
    private long fim;

    public double valorConta() {
        return calculo.calcular(fim-inicio, veiculo);
    }

    public void setCalculo(CalculoValor calculo){
        this.calculo = calculo;
    }
}
```

A seguir, a classe CalculoDiaria mostra um exemplo de uma classe que faz o cálculo da tarifa por dia. Observe que essa classe possui um atributo que pode ser utilizado para parametrizar partes do algoritmo. Dessa forma, quando a estratégia for alterada para o calculo do valor por dia, basta inserir a instância dessa classe em ContaEstacionamento.

Vale também ressaltar que essa mesma classe pode ser reaproveitada para diferentes empresas em diferentes momentos, evitando assim a duplicação de código.

Veja o exemplo de classe que faz o cálculo da tarifa:

```
public class CalculoDiaria implements CalculoValor {

    private double valorDiaria;

    public CalculoDiaria(double valorDiaria){
        this.valorDiaria = valorDiaria;
    }

    public double calcular(long periodo, Veiculo veiculo) {
        return valorDiaria * Math.ceil(periodo / HORA);
    }
}
```

Após implementar a solução, o desenvolvedor, orgulhoso de sua capacidade de modelagem, começa a pensar que essa mesma solução pode ser usada em outras situações. Quando houver um algoritmo que pode variar, essa é uma forma de permitir que novas implementações do algoritmo possam ser plugadas no software. Por exemplo, o cálculo de impostos, como pode variar de acordo com a cidade, poderia utilizar uma solução parecida.

Na verdade, ele se lembrou de que já tinha visto uma solução parecida em um sistema com que havia trabalhado. Como o algoritmo de criptografia que era usado para enviar um arquivo pela rede poderia ser trocado, existia uma abstração comum a todos eles que era utilizada para representá-los. Dessa forma, a classe que enviava o arquivo, era composta pela classe com o algoritmo.

Sendo assim, o desenvolvedor decidiu conversar com o colega a Reconhecendo a recorrência da solução respeito da coincidência da solução usada. Sua surpresa foi quando ele disse que não era uma coincidência, pois ambos haviam utilizado o **padrão de projeto Strategy .**

SINGLETON

"Só pode haver um!" – Connor MacLeod, Highlander

Imagine um software que represente um jogo de xadrez entre duas pessoas. Nesse contexto provavelmente fará sentido apenas um tabuleiro de jogo. A estrutura para permitir esse tipo de construção é o padrão Singleton

A listagem a seguir apresenta um exemplo de implementação do padrão Singleton . Nele, o Singleton é usado em uma classe que representa e armazena configurações do sistema. Esse é um exemplo em que o uso desse padrão é adequado, pois só existe uma única configuração para todo o sistema, e dessa forma ela pode ser facilmente obtida a partir de qualquer classe. Veja um exemplo de Singleton :

```
public class Configuracao {

    private static Configuracao instancia;

    public static Configuracao getInstancia() {
        if(instancia == null)
            instancia = new Configuracao();
        return instancia;
    }

    // Construtor privado!
    private Configuracao() {
        //lê as configurações
    }

    // ...
```

Nesse caso, o método fábrica é normalmente definido na própria classe. O artifício de definir um construtor privado é frequentemente utilizado para impedir a criação de outras instâncias da classe.

Um atributo estático é definido para armazenar essa instância, e um método estático é disponibilizado de forma pública para sua recuperação.

Observe que, na listagem de exemplo, o objeto é criado no primeiro acesso ao método. Uma implementação alternativa seria incluir essa criação em um bloco estático para que ela fosse realizada quando a classe fosse carregada pela máquina virtual.

A listagem a seguir mostra o exemplo de como uma classe pode obter a instância da que implementa o padrão Singleton . Veja que o processo não é muito diferente da invocação de um Static Factory method .

Em qualquer local da aplicação em que o método getInstancia() for invocado, o mesmo objeto será retornado.

Veja a obtenção de um Singleton :

```
Configuracao c = Configuracao.getInstancia();
```

Uma das vantagens do Singleton é a facilidade de acesso dessa instância por qualquer objeto na aplicação. De qualquer classe é possível chamar o método getInstancia() e obtê-la. Isso evita, por exemplo, que essa instância única precise ser passada como parâmetro para diversos lugares.

No exemplo do tabuleiro de xadrez, essa provavelmente seria uma classe que precisaria estar acessível de diversos locais de acordo com a lógica do jogo.

Uma solução usando um Singleton oferece uma flexibilidade muito maior do que uma solução que utiliza métodos estáticos para a execução das regras de negócio. Por ser um objeto, a instância única pode ser especializada e encapsulada por um Proxy , o não é possível fazer com métodos estáticos.

Essa questão prejudica bastante a testabilidade de classes que acessam métodos estáticos, pois não é possível substituí-los por um objeto falso com propósitos de teste, conhecido como Mock Object (mais em Mock Roles, Not Objects, por Steve Freeman, Nat Pryce, Tim Mackinnon e Joe Walnes). Uma solução que concilia a praticidade dos métodos estáticos com a flexibilidade do Singleton seria os métodos estáticos delegarem a lógica de sua execução para os métodos do Singleton .

O padrão Singleton deve ser usado com muito cuidado e nas situações em que realmente fizer sentido ter apenas uma instância de uma classe.

Muitos acham que, por ser um padrão, ele pode ser utilizado em qualquer parte do sistema. O resultado é que o Singleton acaba sendo usado como uma variável global da Orientação a Objetos, o que pode reduzir a flexibilidade da aplicação deixando sua modelagem deficiente.

Por ele ser usado mais em situações inadequadas, muitos acabam considerando o Singleton como uma má prática. Sendo assim, toda vez que for utilizar um Singleton , reflita bastante se seu uso é mesmo necessário.