Programação Orientada e Objetos II

Anhanguera


AVALIE
SUA PROFISSÃO
QUANDO APARECER EM SEU PORTAL UMA AVALIAÇÃO SOBRE SEU CURSO, RESPONDA:
NOTAS
9 OU 10
SIGNIFICA QUE VOCÊ INDICA
NOTAS
7 OU 8
SIGNIFICA QUE VOCÊ NÃO INDICA
Anhanguera

Anhanguera

Padrões de projetos em orientação a objetos

Anhanguera

O desenvolvimento de software deve ser encarado como uma atividade correlacionada com a produção de outros elementos, como de circuito eletrônico, de uma casa ou de um motor. Em alguns momentos, é necessária a produção de um item específico para que o projeto funcione, por exemplo, um resistor de potência ou resistência que não pode ser encontrado no mercado geral ou um tipo ou tamanho de bloco de resistência que não está no catálogo de produção da fábrica. Todavia, pensando no processo como um todo, a maioria das peças utilizadas para resolver os problemas do cotidiano pode ser comprada, sem a necessidade de pedidos especiais. Porém, no desenvolvimento de software, não pensamos dessa forma, ou seja, criamos soluções mais genéricas e, em diversos casos, elas são para problemas recorrentes em diversos projetos de software, para os quais alguém já pode ter disponibilizado a solução.

Pense em um software que deve se conectar em um sistema de gerenciamento de banco de dados (SGBD), como um sistema Enterprise Resource Planning (ERP), que fará a conexão com o SGBD e, de forma geral, é necessária apenas uma conexão. Utilizando os mecanismos da orientação a objetos é possível garantir que, ao utilizar as classes de conexão do banco de dados, apenas uma conexão seja disponibilizada para todas as outras classes? Pensando nesse tipo de problema, a utilização de padrões de projetos oferece recursos para minimizar ou solucionar diversos problemas.

Um padrão é a descrição de uma solução para uma estrutura de projeto, tornando essa abordagem reutilizável por todo o software (GAMMA, 2000). Os padrões de projeto são utilizados para minimizar os problemas que podem ocorrer no desenvolvimento de um software. Além disso, estão relacionados à construção de softwares, e os consumidores diretos não são os usuários finais, mas sim os desenvolvedores. Eles podem ser aplicados a todos os tamanhos de projeto, mas alguns fazem sentido apenas em projetos de médio ou de grande porte, pois nesses tipos, as restrições de modelagem são muito maiores.

Existem 23 padrões de projeto catalogados pelos autores Erich Gamma, Richard Helm, John Vlissides e Ralph Johnson. Todavia, pesquisas trazem novas soluções para outros problemas ou para outras formas de programação, como a orientação a aspectos. Focaremos em alguns que possuem aplicação mais geral. Os padrões de projetos são divididos por propósito e escopo. Nesse livro veremos sobre o propósito, o qual podemos dividir em três tipos:

- **Criação:** tem o objetivo de encapsular a criação de elementos como subclasses ou objetos. Esse processo está literalmente relacionado à forma de manter a complexidade dentro das classes ou objetos.
- **Estrutural:** os padrões estruturais têm o objetivo de apresentar uma forma de entender uma parte do sistema de maneira
mais simples e padronizada, sempre pensando nos elementos de coesão e acoplamento.
- **Comportamental:** o padrão comportamental tem o foco em apresentar as formas de como um conjunto de objetos podem se relacionar de maneira controlada, deixando claro qual é o fluxo de informação ou notificação.

Quadro 3.11 | Padrões de projeto organizados dado o propósito de sua aplicação

Propósito		
De criação	*Estrutural*	Comportamental
Factory Method	*Adapter*	*Interpreter*
Abstract Factory	*Bridge*	*Template Method*
Builder	*Composite*	*Chain of Responsibility*
Prototype	*Decorator*	*Command*
Singleton	*Façade*	*Iterator*
	Flyweight	*Mediator*
	Proxy	*Memento*
		Observer
		State
		Strategy
		Visitor

Fonte: adaptado de Gamma (2000, p. 26).

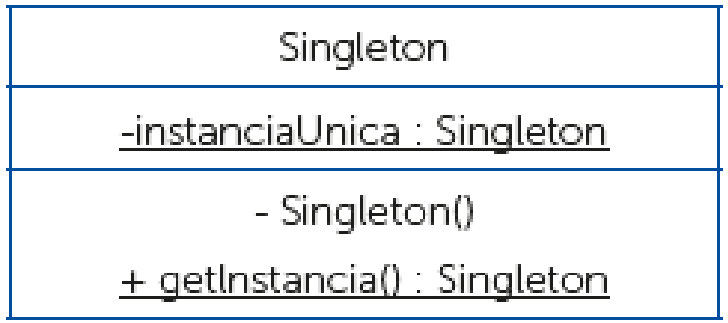
Para exemplificar o uso, vamos ver a aplicação de um padrão de cada tipo. Do tipo de criação, observaremos o Singleton, do tipo estrutura o Façade e do tipo comportamental o Template Method. Esses padrões são relacionados a situações que podem ocorrer em projetos de todos os portes. Portanto, veremos como fazer um código que poderá ser reutilizado, garantindo o acoplamento e a coesão do código.

O primeiro padrão que vamos explorar será o projeto de criação chamado Singleton. O padrão de projeto Singleton está relacionado à necessidade de manter apenas uma instância de certa classe parao sistema inteiro, criando um ponto de acesso global para o sistema (GAMMA, 2000). Como exemplos de situações, podemos destacar a necessidade de garantir que se tenha apenas uma conexão ao SGDB, o acesso único ao sistema de controle (como a conexão serial de um Arduino) ou ainda uma fila de impressão.

O padrão de projeto Singleton usa diversos arcabouços da orientação a objetos para prover a capacidade de fornecer apenas uma instância de algum objetivo específico. Vamos trabalhar com um cenário em que é necessário fornecer apenas uma instância de uma classe que implementa a conexão serial com um Arduino (hardware que possui um microcontrolador com interface de fácil programação. Para mais informações, acesse https://www.arduino.cc/, acesso em: 21 ago. 2018). Esse microcontrolador utiliza um sensor de temperatura e envia as medidas para o computador através de uma porta serial (normalmente emulada pela porta USB). Esse tipo de conexão não pode ser utilizado por mais de uma instância por causa de suas próprias restrições. Caso seja usado, o sistema informará que já existe uma conexão aberta.

Antes de analisarmos o padrão de projeto, vamos relembrar os conceitos de métodos e atributos estáticos no Java. Esses elementos marcados com a palavra-chave static não precisam da instância da classe para serem utilizadas, e um exemplo do uso é a classe java. lang.Math, em que temos diversos métodos static, como Math.sqrt(), que calcula a raiz quadrada de um número, ou Math.PI que contém o número pi. Veja que nesses casos não é necessário usar a palavra new, ou seja, não é criada uma instância da classe Math. O uso de métodos static é direcionado a métodos ou atributos que necessitam de configuração inicial, não sendo necessário um construtor. O mesmo ocorrerá em outros casos, como veremos no padrão Singleton.

Antes de programarmos a solução em Java, vamos observar o diagrama da Figura 3.6, que representa a visão do padrão de projeto Singleton. Repare que o construtor é privado (- Singleton()), assim, apenas elementos de dentro da classe podem instanciar o objeto. Além disso, o atributo do tipo Singleton estático (- instanciaUnica : Singleton) também é privado, então, novamente, apenas elementos dentro da classe podem utilizá-lo. Essa configuração permite construir a classe apenas dentro dela mesma e, ainda, o atributo estático só pode ser acessado pelo método estático getInstancia(), que é público (veja o sinal de + antes do nome do método) e retorna um objeto da classe Singleton, portanto, a única forma de acesso auma instância da classe é o método getInstancia().

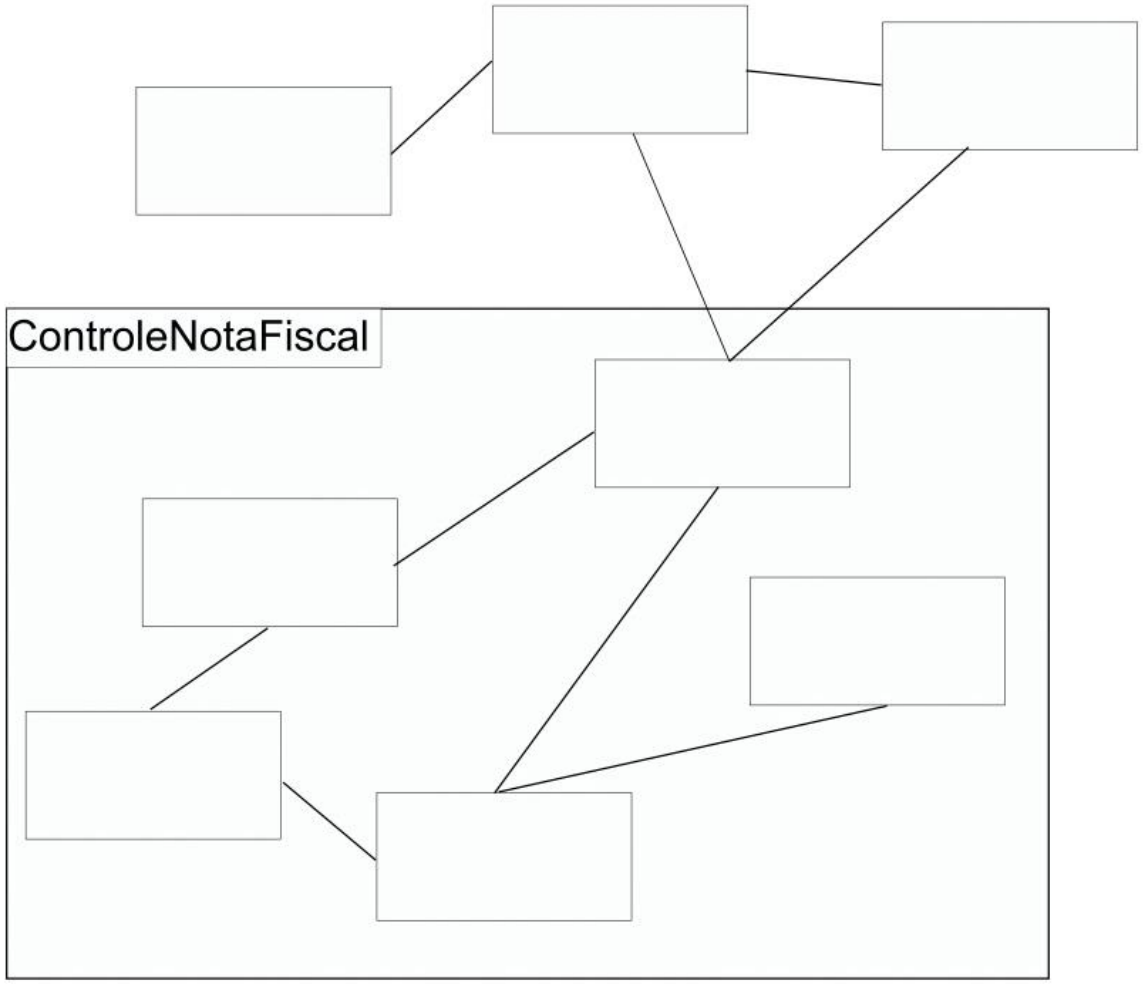


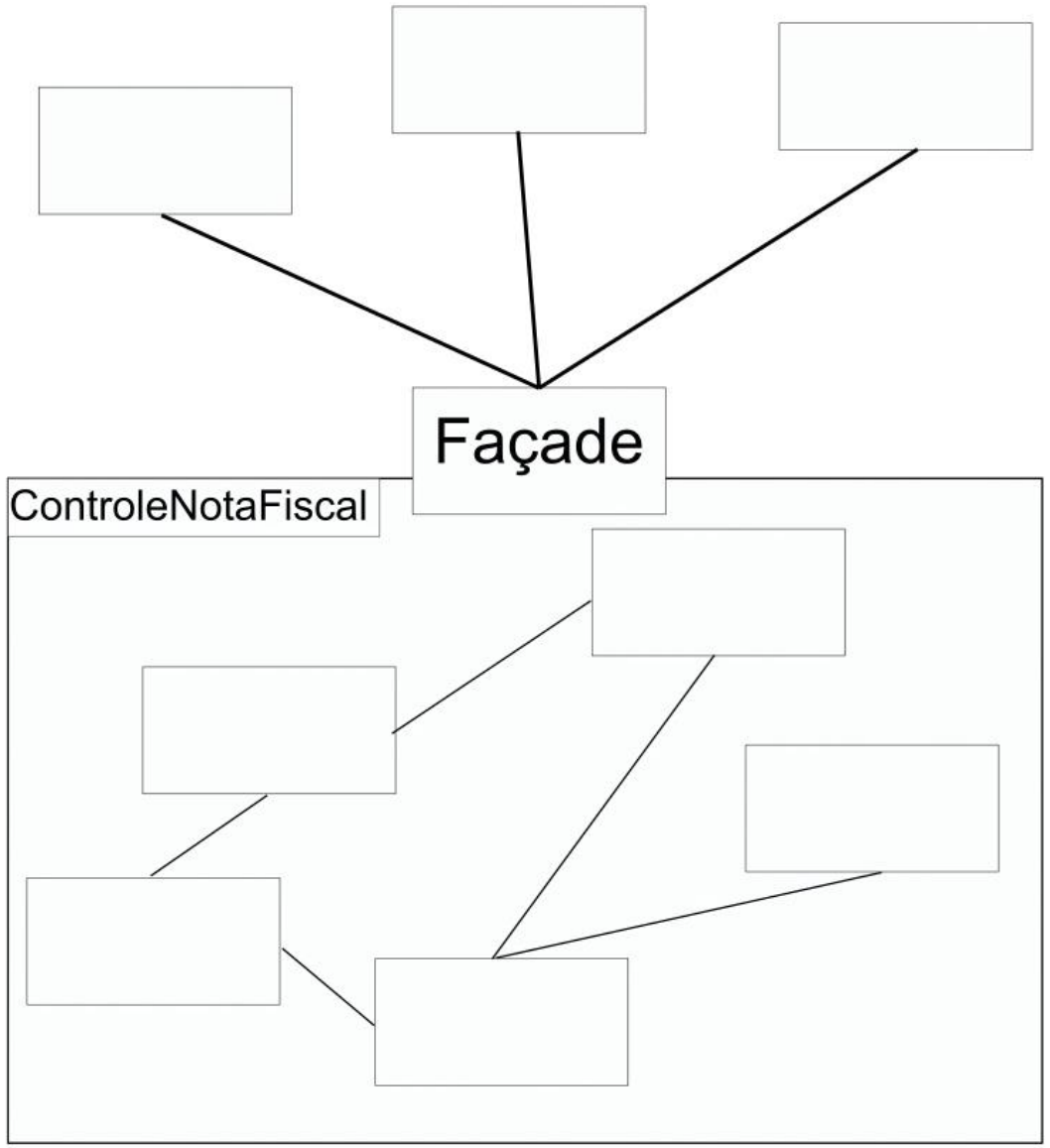
Para que fique claro o padrão Singleton, vamos analisar o código dele implementado em um cenário da conexão serial do Arduino. O Quadro 3.12 apresenta esse código utilizando uma biblioteca JSerialCom para leitura da porta (disponível em: <http://bit.ly/2MIcZ9O>. Acesso em: 21 ago. 2018). A linha 2 apresenta o import necessário para a biblioteca JSerialComm. A classe ComSerialSingleton possui um atributo não estático na linha 4 do tipo SerialPort que depende da instância da classe e um atributo estático na linha 5 (ComSerialSingleton). O construtor privado da classe na linha 6 até a 12 faz a configuração quando ela é instanciada pelo método estático na linha 13. Repare que, no método getInstancia(), na linha 14, se o atributo estático comSerial for null, ele vai criar uma instância da classe na linha 15 e no método getInstancia() ele é synchronized, com isso, mesmo que diferentes threads chamem o método, será executada apenas uma por vez, garantindo a criação de apenas uma instância da classe. Por fim, no método retornaDados() das linhas 17 a 27, é feita a leitura da serial. Esse método também é synchronized para impedir que duas threads diferentes peçam a leitura da porta serial ao mesmo tempo.

```
1.  package U3S2;

2.  import com.fazecast.jSerialComm.SerialPort;

3.  public class ComSerialSingleton {
        // instância privada da classe
4.      private SerialPort comPort;
        // atributo static para guarda a instância da
5.  classe
        private static ComSerialSingleton comSerial;

6.      // construtor privado para evitar a instância
    da classe sem controle
7.      private ComSerialSingleton() {
8.
9.          comPort = null;
            try {
10.             comPort = SerialPort.getCom-
11. mPort("COM4");
12.             comPort.openPort();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
13.
        public static synchronized  ComSerialSingle-
    ton getInstancia()
        {
14.         // se a classe nunca foi construida
            if(comSerial == null)
            {
                // usa o construtor private para
15. construir a classe
                comSerial = new ComSerialSingle-
    ton();
            }
```

```
16.             return comSerial;
        }
        // como é a mesma instância não se pode dei-
xar duas threads fazerem a leitura ao mesmo, com
isso é necessário synchronized
17.     public synchronized String retornaDados()
        {
18.         while (comPort.bytesAvailable() == 0)
19.             try {
20.                 Thread.sleep(20);
21.             } catch (InterruptedException e) {
                    e.printStackTrace();
22.             }
            // cria o buffer de leitura da porta se-
rial
23.         byte[] readBuffer = new byte[comPort.by-
tesAvailable()];
24.         // faz a leitura da porta serial
            int numRead = comPort.readBytes(readBuf-
fer, readBuffer.length);
25.         // cria uma string com os dados vindos
da porta serial
            String dados = new String(readBuffer);
            return dados;
26.     }
27. }
```

Além dos padrões de criação, temos os padrões para descrever estruturas de um sistema. Eles são muito úteis para descrever um sistema e corrigir problemas de encapsulamento, coesão e acoplamento, além de terem uma visão mais global do sistema. Um deles é Façade, cujo objetivo é prover uma forma de acesso mais simples ao subsistema em um grande software. Como exemplo, podemos pensar em um sistema de processamento de notas fiscais. Para fazer a emissão é necessário usar as classes produto, pedido, nota fiscal, transmissor, validador, impressor e outras. A Figura 3.7 apresenta uma visão da utilidade desse padrão no contexto de processamento de notas. Com ele é possível aumentar a coesão e diminuir o acoplamento. Repare que na imagem sem Façade (lado esquerdo) é necessário que diversas classes tenham relação com classes de todos os níveis do pacote ControleNotaFiscal, porém, com o Façade (lado direito) é possível ter apenas uma classe de comunicação, não sendo necessário ter acesso a pacotes internos e a complexidade do pacote.


ControleNotaFiscal


Façade
ControleNotaFiscal

O código no Quadro 3.13 apresenta um exemplo da aplicação do padrão Façade. Com ele é possível deixar toda a complexidade do uso de um pacote em apenas uma classe. Repare que entre as linhas 3 e 8 são declaradas todas as classes do pacote como atributo. O construtor entre as linhas 9 e 15 fazem a instanciação e o método public void criaNotaFiscal() faz a toda a operação.

```
1.  package U382;

2.  public class FacadeControleNotaFiscal {

3.       private Produtos[] lstProdutos;
4.       private Pedido pedido;
5.       private NotaFiscal nota;
6.       private Transmissor enviar;
7.       private Validador valida;
8.       private Impressor impressora;

9.       public  FacadeControleNotaFiscal(int nPedido)
         {
10.           pedido = new Pedido(nPedido);
11.           lstProdutos = pedido.getProdutos();
12.           nota = new NotaFiscal();
13.           enviar = new Transmissor();
14.           valida = new Validador();
15.           impressora = new Impressor();
         }

16.      public void criaNotaFiscal()
         {
17.           nota.insereProdutos(lstProdutos);
18.           enviar.enviarNota(nota);
19.           valida.validar(nota);
20            impressora.imprimir(nota);
         }
    }
```

Fonte: elaborado pelo autor.

Para finalizar a apresentação dos padrões de projeto, temos os padrões comportamentais, cujo objetivo é padronizar a maneira como é feita a comunicação entre as classes. Um deles é o Template Method, que define um esqueleto ou um padrão para um conjunto de operações, que são implementadas por subclasses.

Como exemplo, pense em sensores de temperatura via rede sem fio (wi-fi) para automação residencial, mas eles são de marcas diferentes e suas conexões são feitas de formas diferentes. Embora cada marca tenha suas particularidades, o processo padrão consiste em conectar o sensor e retornar à temperatura. Primeiramente, é necessário criar uma classe abstrata (não pode ser instância, possui apenas um modelo e deve ser especializada para ser utilizada). O Quadro 3.14 apresenta um exemplo de classe abstrata, que é o primeiro passo do Template Method. Nela definimos a sequência padrão de acesso ao dispositivo, no método public fl oat retornaTemperatura(String ip, int porta) entre as linhas 3 e 5. Veja que primeiro será executado o método conecta() e depois o retornaValor(). Nas linhas 6 e 7, os métodos são definidos com parâmetro abstrato, ou seja, deverão ser sobrescritos na classe que especializar.

```
1.  package U332;

2.  public abstract class ConectorSensor {

3.      public float retornaTemperatura(String ip,
    int porta)
        {
4.          conecta(ip, porta);
5.          return retornaValor();
        }

6.      protected abstract boolean conecta(String
    ip, int porta);
7.      protected abstract float retornaValor();
    }
```

Fonte: elaborado pelo autor.

Após criar a classe abstrata, é necessário estender a classe ConectorSensor e implementar os métodos conecta() e retornaValor() com as especificidades de cada sensor. Como amostra, o Quadro 3.15 apresenta um sensor que aceita conexões TCP. Na linha 4, a classe SensorModeloTCP que especializa a classe ConectorSensor. As linhas 5 e 6 descrevem os atributos para a conexão TCP. Nas linhas 7 a 12 está a implementação do método conecta(), que sobrescreve a classe abstrata para definir como conectar no sensor, bem como no método retornaValor(), descrito nas linhas de 13 a 24. O ponto principal desse padrão é a utilização dessas implementações entre as linhas entre as 26 a 29. Repare que é declarada a classe ConectorSensor que recebe a implementação do SensorModeloTCP, assim, quem utiliza a classe tem acesso apenas aos métodos conectar() e retornaValor().

Anhanguera

```
1.  package U3S2;
2.  import java.io.*;
3.  import java.net.Socket;
4.  public class SensorModeloTCP extends ConectorSen-
    sor{
5.      private Socket recebeSocket;
6.      private BufferedReader leitor;
        @Override
7.      public boolean conecta(String ip, int porta)
8.  {
9.              try {
10.                recebeSocket = new Socket(ip, porta);
11.             } catch (Exception e) {
12.                e.printStackTrace();
                }
                return false;
        }
```

```
13.
14.     @Override
15.     public float retornaValor() {
            try {
16.             leitor = new BufferedReader(new In-
putStreamReader(recebeSocket.getInputStream()));
17.             String dados = leitor.readLine();
18.             return Float.valueOf(dados);
19.         } catch (IOException e) {
20.             e.printStackTrace();
21.         }finally {
22.             try {
23.                 recebeSocket.close();
24.             } catch (IOException e) {
                    e.printStackTrace();
                }
            }
```

```
25.             return 0;
        }
 26     public static void main(String[] args) {
27.             ConectorSensor sensor = new SensorMode-
28. loTCP();
29.                             sensor.conec-
ta("192.168.0.1", 7894));
                System.out.println(sensor.retornaVa-
lor();
        }
}
```

Fonte: elaborado pelo autor.

Os padrões de projeto são amplamente utilizados em sistemas de todos os portes, e sua aplicação depende do conhecimento das pessoas que estão fazendo a implementação e a modelagem do sistema. Com isso, é possível aumentar as chances de reutilização das classes e dos módulos.

Qual é o objetivo dos padrões de projeto?

a) Apresentar o uso de classes e interfaces.

b) Mostrar a informações das classes de forma clara e direta.

c) Descrever como declarar os atributos de uma classe.

d) Fornecer soluções para problemas recorrentes em sistemas orientados a objetos.

e) Fornecer soluções para problemas incomuns em sistemas orientados a objetos.

Dentro de diversos padrões de projeto, podemos citar o Singleton como uma forma de garantir que certa instância de uma classe seja única em todo o sistema. Ele permite o uso de recursos como conexões a dispositivos externos ou outros elementos que necessitam de apenas uma instância.

Quando se utiliza o Singleton, qual é a função do construtor privado?

a) Garantir que não seja possível construir o objeto de fora da classe.

b) Permitir que a classe seja vista por todo o sistema.

c) Gerar elementos de controle da classe.

d) Permitir que o objeto seja construído fora da classe.

e) Bloquear o acesso das threads