Padrões de projetos em orientação a objetos.

Programação Orientada a Objetos

2023-04-25

- No desenvolvimento de software geralmente cria-se soluções mais genéricas para um problema
- Os problemas podem ser recorrentes em diversos projetos de software, para os quais alguém já pode ter disponibilizado a solução
  - Pergunta: existe uma melhor forma de desenvolver um sistema?
- Exemplo
  - Conexão a um banco de dados
    - Ao utilizar as classes de conexão, é possível que apenas uma conexão seja disponibilizada para todas as outras classes?

- Padrões de projetos oferecem 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
- Alguns padrões fazem sentido apenas em projetos de médio ou de grande porte, pois nesses tipos as restrições de modelagem são muito maiores

- Padrões de projeto são soluções a serem aplicadas em cenários bem definidos durante o desenvolvimento de um projeto
- Os usuários são outros desenvolvedores que farão a reutilização que o padrão propõe
- Gamma et al. (2000) catalogou 23 padrões de projeto

- Coesão: classes bem definidas e de propósito único
- Acoplamento: grau com que uma classe conhece a outra, modularidade, flexibilidade
- Encapsulamento: controle de acesso aos atributos e métodos de uma classe

- Focaremos nos padrões com aplicação mais geral
- Padrões são divididos por propósito e escopo
- Veremos os por propósito (3 tipos)
  - Criação: encapsula a criação de elementos como subclasses ou objetos, mantendo complexidades dentro das classes, ex.: uma classe para acesso global
  - Estrutural: entende uma parte do sistema de maneira mais simples e padronizada, pensando em coesão e acoplamento, ex.: uma classe que faz toda comunicação
  - Comportamental: apresenta formas de como um conjunto de objetos podem se relacionar de maneira controlada, ex: classe abstrata

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	*Facade*	*Iterator*
	Flyweight	*Mediator*
	Proxy	*Memento*
		Observer
		State
		Strategy
		Visitor

tipos escolhidos para os exemplos

"fassaad" (fachada)

Padrões

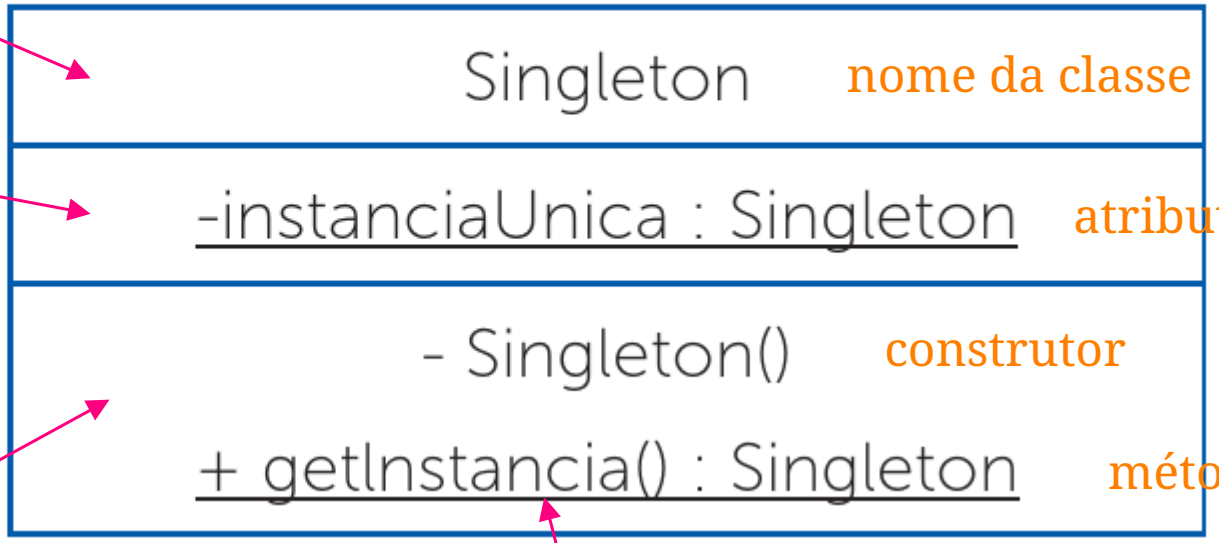
- São relacionados a situações que podem ocorrer em projetos de todos os portes
- Permitem que código possa ser reutilizado, garantindo o acoplamento e a coesão do código
- Padrões de projeto são essenciais em grandes projetos

Singleton

- Manter apenas uma instância de certa classe para o sistema inteiro
  - Criando um ponto de acesso global
- Ex: garantir que se tenha apenas uma conexão ao SGDB
- Utiliza recursos da OO para garantir uma única classe
- Ex.: fornecer apenas uma instância de uma classe que implementa a conexão serial com um Arduino
  - Sensor de temperatura que envia as medidas para o computador através de uma porta serial

- Caso seja usado mais que uma instância da classe
  - O sistema informará que já existe uma conexão aberta
- Relembrando alguns conceitos
  - Palavra-chave `static` ← não precisa instanciar
  - Exemplo de métodos estáticos em java.lang.Math
    - Math.sqrt()
    - Math.PI

Figura 3.6 | Diagrama de classe do padrão de projeto *Singleton*



- classe
- atributo do tipo Singleton estático
- privado
- apenas elementos de dentro da classe podem instanciar o objeto
- construtor privado
- retorna uma classe Singleton
- única forma de acesso a uma instância da classe
- público

Exemplo

Quadro 3.12 | Implementação do padrão *Singleton* para conexão serial

```
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;
```

biblioteca que lê a porta serial

tipo

atributo não estático, precisa instanciar

não precisa instanciar

mesmo tipo da classe

atributo

(continua)

- Garante que apenas um objeto seja criado por vez
- Restringe as instâncias de classe dentro da classe declarada

vide instanciação no próximo slide

```
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();
            }
        }
```

construtor

inicializa o atributo

cria e armazena um objeto SerialPort

abre a porta serial para leitura e escrita

(continua)

- usa-se este método para instanciar
- caso a instância já exista, ele não cria outra

```
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;
     }
```

somente uma thread

tipo da classe

só vai instanciar a classe se ela não existir

nome do método

atributo estático definido no slide 12

instanciação da classe

retorna o objeto

(continua)

```
        // 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. }
```

impede execução e leitura simultânea da porta

criado no slide anterior

pausa a thread

retorna a string lida

(fim)

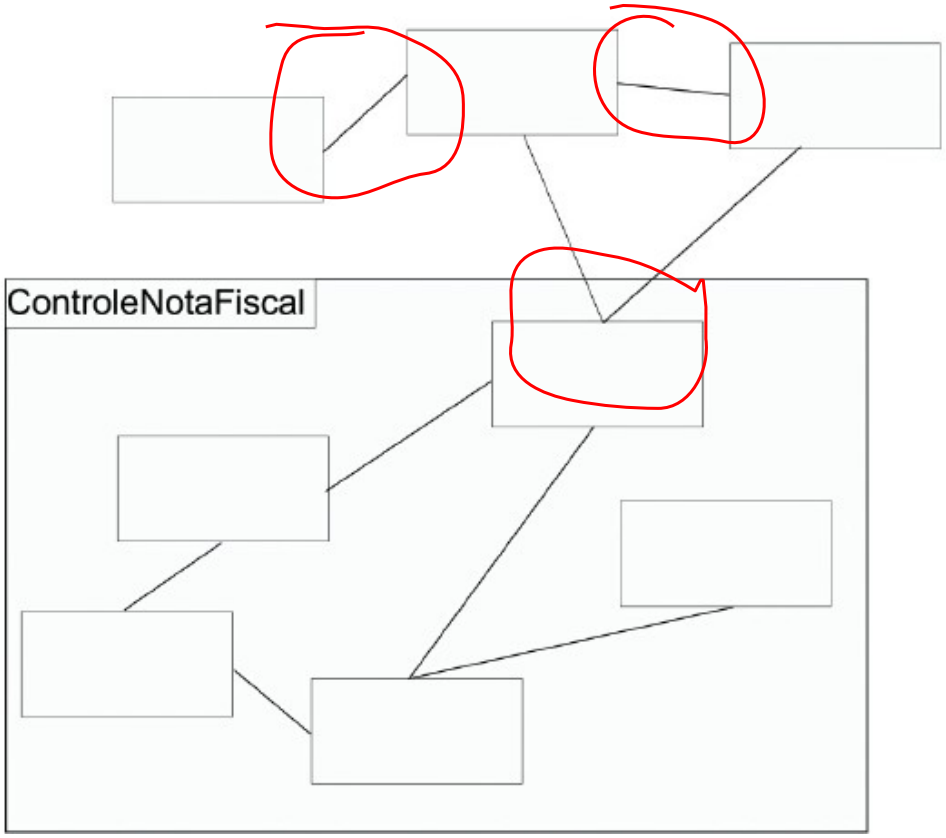
Padrões para estruturas de um sistema

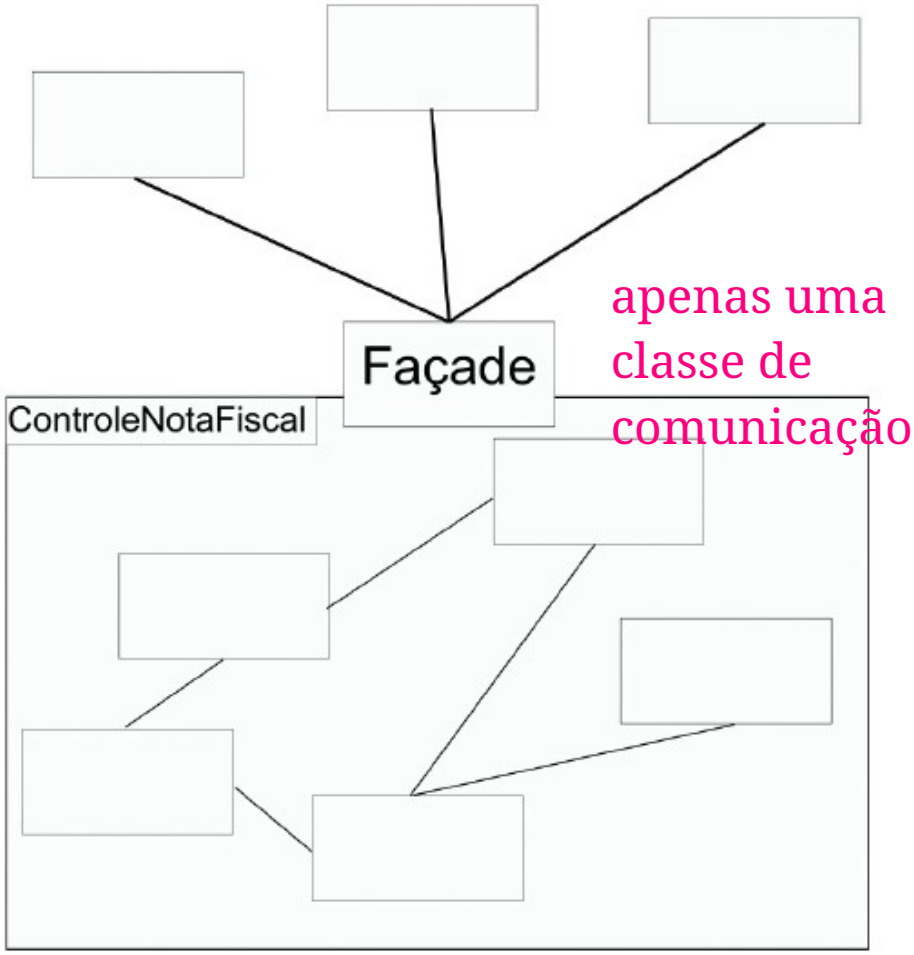
- Foca em encapsulamento, coesão e acoplamento
- Tem uma visão mais global do sistema
- O Facade provê uma forma de acesso mais simples ao subsistema em um grande software
  - Ex.: apenas uma classe de comunicação

Facade

Figura 3.7 | Exemplo de diagrama de classe sumarizado com e sem o padrão *Façade*

sem Facade, classes se comunicam sem a necessidade de uma centralização





apenas uma classe de comunicação

aumenta a coesão e diminui o acoplamento

Quadro 3.13 | Exemplo de utilização do *Façade*

uma única classe centralizando a comunicação com as demais

```
1.  package U3S2;

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;
```

(continua)

```
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);
        }
}
```

demais classes

demais métodos

(fim)

Padrões comportamentais

Template Method

- Objetivo padronizar a maneira como é feita a comunicação entre as classes
- Template Method
  - Define um esqueleto (padrão) para um conjunto de operações que são implementadas por subclasses
- Primeiramente, é necessário criar uma classe abstrata
  - Que não pode ser usada para criar objetos
  - Deve ser herdada em outra classe

Quadro 3.14 | Exemplo de *Template Method* (classe abstrata)

```
1.  package U3S2;

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();
    }
```

classes abstratas podem ser usadas para definir um padrão

método padrão de acesso ao dispositivo

primeiro executa esse

depois esse

este método será depois sobreescrito e implementado no slide 23

este no 24

(continua)

Quadro 3.15 | Especialização da classe **ConectorSensor**

sensor que aceita conexões TCP

```
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;
```

classe que especializa a classe

será usado em main

definida no slide anterior

atributos para a conexão TCP

(continua)

implementação do método que sobrescreve a classe abstrata do slide 21 para definir como conectar no sensor

```
7.      @Override
        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;
        }
```

(continua)

método que sobrescreve e implementa a classe abstrata do slide 21

```
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;
        }
```

(continua)

a execução inicia por aqui

```
26      public static void main(String[] args) {
27.           ConectorSensor sensor = new SensorMode-
28.  loTCP();                                          slide 22
29.                              sensor.conec-
     ta("192.168.0.1", 7894));                  slide 23
              System.out.println(sensor.retornaVa-
     lor();
          }
     }                                  (fim)
```

- quem utiliza a classe tem acesso aos métodos conecta() e retornaValor()
- o ponto principal é a utilização de classe abstrata para ajudar na padronização