Capítulo 3

Padrões de projeto Singleton e MVC

Padrões de projetos podem solucionar diversos problemas durante o desenvolvimento de um sistema. Existem diversos tipos de padrões, que, como vimos, são divididos em padrões de criação, comportamentais e estruturais. Dentre os padrões estruturais podemos encontrar uma outra categoria muito importante, os padrões arquiteturais. Os padrões arquiteturais ocupam-se de como as camadas lógicas do projeto serão criadas e organizadas, não estando no nível dos objetos do sistema, como os padrões estruturais (FOWLER, 2002; GAMMA *et al.*, 2009).

Neste capítulo, serão apresentados dois padrões de projetos, um de criação e outro arquitetural: os padrões Singleton e MVC (model-view-controller), que são, sem dúvidas, os mais utilizados em projetos corporativos. O primeiro consegue gerenciar a quantidade de instâncias de

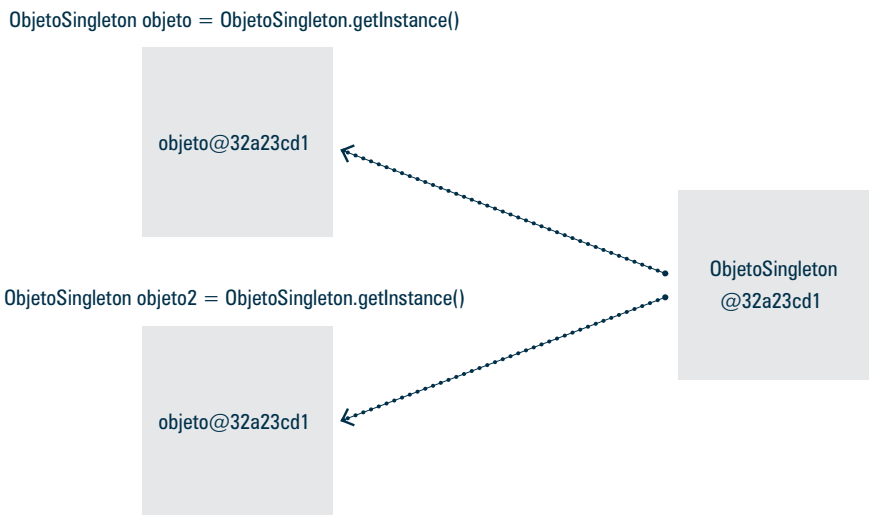
determinado objeto na memória. O segundo apresenta uma maneira de organizar o código em três camadas.

1 Padrão de projeto Singleton

O padrão de projeto Singleton garante a criação de apenas uma única instância do objeto no sistema inteiro. Caso seja solicitada uma instância de um objeto com esse padrão implementado, assegura-se que ele sempre retornará a mesma instância criada inicialmente. Assim, é possível garantir que os objetos criados sejam sempre da mesma instância, independentemente do tamanho da aplicação e da quantidade de usuários do sistema (GAMMA *et al.*, 2009; ENGHOLM JUNIOR, 2017).

De maneira prática, isso funciona, pois o construtor da classe é criado de maneira privada, e utiliza-se um método que retorna a instância da classe em questão. O método será chamado de *getInstance*; em tradução livre, "pegue a instância". A figura 1 ilustra esse exemplo.

Figura 1 – Processo de instância de um objeto Singleton



Na figura 1, temos o processo de geração de um objeto Singleton no sistema. À esquerda da imagem, temos dois quadrados que representam os objetos em memória obtidos a partir do código posicionado acima de cada quadrado. Esse código é a declaração de um objeto do tipo *ObjetoSingleton*, e é realizada uma atribuição a partir do método *getInstance* da classe *ObjetoSingleton*. Nos quadrados que representam esses objetos, temos os nomes dados para cada um na declaração, seguidos pelo símbolo "@", que em inglês significa o local exato, seguido pelo endereço de memória de onde o objeto é armazenado. Esse endereço é representado por um número hexadecimal.

Repare que ambos os objetos possuem o mesmo endereço de memória, o que significa que são os mesmos objetos, mesmo partindo de declarações diferentes. À direita da figura 1, existe um quadrado com o nome da classe, o símbolo "@" e o mesmo endereço de memória dos objetos em uso. Isso significa que o objeto foi instanciado e ele devolve esse endereço toda vez que o método *getInstance* é executado, representado pelas setas que saem dele em direção aos demais objetos.

NA PRÁTICA

Uma aplicação que sempre se encaixa bem com o padrão Singleton são as classes que fazem o gerenciamento e controle de vários recursos – por exemplo, quando é necessário realizar a gestão de uma fila de impressão dentro de uma empresa. Perceba que podem existir diversas impressoras espalhadas na rede, mas há apenas um controle da fila de impressão (GAMMA *et al.*, 2009).

Sempre que esse assunto é abordado, a primeira pergunta que passa na cabeça é: como garantir que apenas uma instância para determinada classe seja criada? De forma simples, a resposta é deixar que a própria classe gerencie a criação dos seus objetos (GAMMA *et al.*, 2009). Para

que isso ocorra, precisamos pensar em alguns detalhes da implementação de um objeto Singleton, como (ENGHOLM JUNIOR, 2017):

- A declaração de um construtor padrão com o modificador de acesso privado é essencial nesse processo.
- O método *getInstance* precisa ser declarado junto com a palavra reserva *synchronized* para evitar problemas com processamento concorrente nos sistemas.
- É necessário anular o efeito da invocação do método *clone* para que não exista maneira de criar adaptações de novas instâncias.

Vamos implementar o *ObjetoSingleton* na prática, com o uso de linguagem de programação Java por meio do código:

```
1.  public class ObjetoSingleton {
2.
3.      private static ObjetoSingleton objetoSingleton;
4.
5.      private ObjetoSingleton() {
6.          System.out.println("ObjetoSingleton criado com
    sucesso.");
7.      }
8.
9.      public static synchronized ObjetoSingleton getInstance() {
10.         if(objetoSingleton == null) {
11.             objetoSingleton = new ObjetoSingleton();
12.         }
13.         return objetoSingleton;
14.     }
15.
16.     @Override
17.     public Object clone() throws  CloneNotSupportedException{
18.        throw new CloneNotSupportedException();
19.     }
20.
21. }
```

Na linha 3 do código, é declarado um atributo estático do mesmo tipo da classe (linha 1), onde será armazenada a única instância da classe. O construtor padrão com modificador de acesso privado é declarado entre as linhas 5 e 7. Como é um objeto simples, o construtor apenas apresenta a mensagem quando o objeto é criado. O método *getInstance* é criado entre as linhas 9 e 14. Na linha 10, é verificado se o atributo que contém a instância desse objeto é nulo. Em caso de verdade, a instância é criada na linha 11, uma única vez. E a instância é retornada na linha 13.

Da linha 16 a 19, é feita a sobrescrita do método *clone*, lançando uma exceção caso ele seja chamado. A exceção lançada é a CloneNotSupportedException, que significa que esse método *clone* não é permitido na classe *ObjetoSingleton*.

Vamos agora criar uma classe para testar esse objeto, com o código:

```
1.  public class Main {
2.
3.      public static void main(String[] args) {
4.          ObjetoSingleton obj1 = ObjetoSingleton.getInstance();
5.          ObjetoSingleton obj2 = ObjetoSingleton.getInstance();
6.
7.          if (obj1 == obj2) {
8.              System.out.println("Possuem a mesma instância.");
9.          }
10.     }
11.
12. }
```

Nas linhas 4 e 5, as instâncias são chamadas através do método *getInstance* para os objetos *obj1* e *obj2*. Na linha 7, os objetos são comparados com o símbolo de igual (==), pois a comparação deve ser feita pelo endereço de memória. Se o endereço for o mesmo, significa que a instância é a mesma e o Singleton foi implementado com sucesso. E na mensagem exibida na linha 8, surgirá como resposta: "ObjetoSingleton criado com sucesso" e "Possuem a mesma instância".

A mensagem de objeto criado com sucesso apareceu apenas uma vez. Isso significa que o construtor do *ObjetoSingleton* foi chamado apenas uma única vez. E, após a comparação, foi exibida a mensagem de mesma instância. Claro que esse é um objeto simples, para demonstrar como o padrão é implementado. Mas pode ser utilizado com objetos mais complexos e diversos outros métodos. Estes implementados aqui são os essenciais para um objeto Singleton.

2 Padrão MVC (model-view-controller)

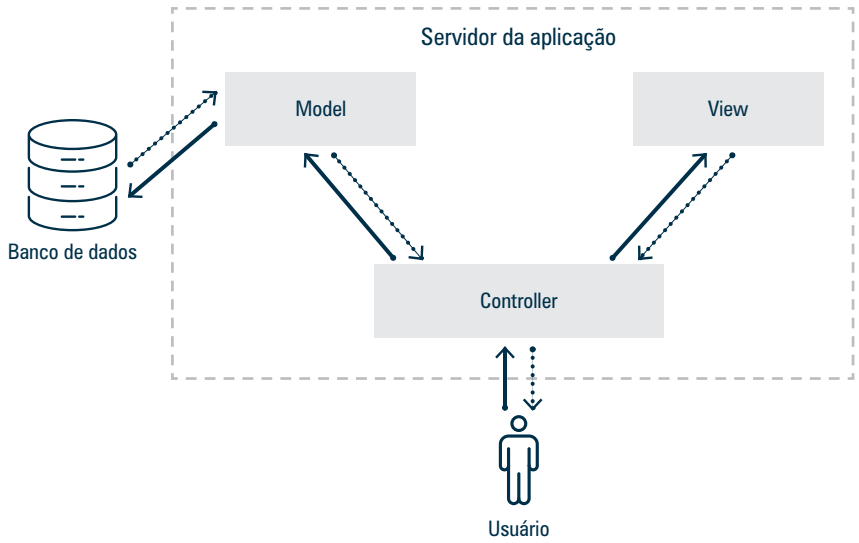
O padrão arquitetural model-view-controller, ou MVC, representa um marco para o desenvolvimento de sistemas web, tendo causado uma grande revolução na maneira como os sistemas web são desenvolvidos. Antes do MVC, o sistema era desenvolvido sem preocupações com a interação do designer com o projeto e sem separar a parte do cliente e a lógica de negócio no servidor. O resultado disso era que a manutenção e a implantação dos sistemas eram mais complicadas e muitas vezes complexas, principalmente para grandes equipes (FOWLER, 2002; SILVEIRA *et al.*, 2011).

Originado na comunidade de desenvolvimento do Smalltalk, o MVC começou sua aplicação no mundo web com a comunidade após sua documentação no catálogo de padrões Core J2EE Pattern. Claro que não veio com o mesmo propósito do padrão aplicado no Smalltalk, e no mundo Java Web houve diversos outros padrões para suportar o MVC. Mas, com o passar do tempo, diversos frameworks implementando o padrão MVC foram apresentados às comunidades de desenvolvimento. Independentemente do framework, obteve-se a unanimidade do padrão nas diversas comunidades (SILVEIRA *et al.,* 2011).

O grande objetivo do MVC é tornar a manutenção do sistema mais fácil, por meio da separação em camadas. A separação em camadas ocorre segundo as responsabilidades dos objetos, havendo a camada

visual (view), a camada de negócio (model) e a camada de controle (controller), que faz a comunicação entre as outras duas (SILVEIRA *et al.*, 2011). Na figura 2, é apresentada uma abordagem para o padrão MVC em um sistema web simples.

Figura 2 – Exemplo de um padrão MVC aplicado a um sistema web simples



Na figura 2, temos o usuário, que faz uma solicitação à aplicação web por meio do cliente (navegador). O cliente dispara o pedido ao controller, que irá solicitar ou enviar à camada model as informações da requisição. Na camada model, pode-se encontrar classes de domínio, repositórios e classes de acesso ao banco de dados. Após o retorno da informação por parte da model, o controller redireciona as informações para serem formatadas na camada de apresentação view. Nessa camada, as informações são processadas e produzidas com o estilo de apresentação adequado ao usuário. Após esse processo, são devolvidas para o controller, que as envia em formato de resposta para exibição ao usuário através do navegador.

Existem muitas discussões sobre o que se encaixa na camada model, porém algo que é bem claro é que esta pode conter diversas subcamadas, de acordo com a necessidade do projeto. Mas, antes de pensar quais são as subcamadas e o que deve estar contido em cada uma, deve-se aplicar a divisão das três camadas básicas do padrão. Esse, sem sombra de dúvidas, é o ponto inicial para um desenvolvedor que deseja aperfeiçoar sua carreira como arquiteto de software.

Considerações finais

Neste capítulo, apresentamos os conceitos dos padrões de projeto Singleton e MVC. Com eles, podemos controlar as instâncias dos objetos que criamos no sistema e ainda estruturar as camadas do projeto. Dessa maneira, os projetos ficam mais bem estruturados e nenhuma instância de objetos é criada a mais, sem necessidade, além de se tornar mais fácil a manutenção com os demais membros da equipe.

Referências

ENGHOLM JUNIOR, Hélio. **Análise e design orientados a objetos**. São Paulo: Novatec, 2017.

FOWLER, Martin. **Patterns of enterprise application architecture**. Boston: Addison-Wesley, 2002.

GAMMA, Erich *et al*. **Design patterns elements of reusable object-oriented software**. Boston: Addison-Wesley, 2009.

SILVEIRA, Paulo *et al*. **Introdução à arquitetura de design de software**: uma introdução à plataforma Java. Rio de Janeiro: Elsevier Brasil, 2011.