Design Patterns em Java

Continuação do estudo de FSM

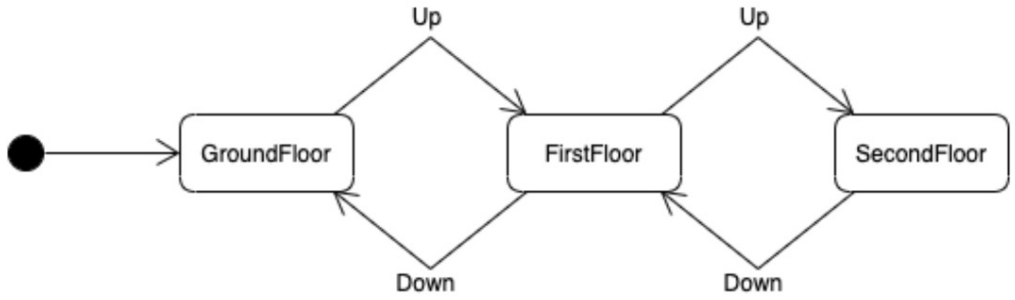
Factory

Singleton

Exercício (continuação)

- Resolva a alínea b) do exercício 24 da ficha de exercícios

24. Considere o seguinte diagrama de estados, que descreve o funcionamento de um elevador num prédio com 3 pisos.



...

- Faça as modificações necessárias para garantir a seguinte funcionalidade:
  i. Quando o utilizador carrega no botão para subir ou descer, existe uma probabilidade de o elevador ficar avariado de 10% se ocorrer no R/C, de 20% no 1º andar e de 30% no 2º andar. Sempre que é detectado um erro, o elevador entra num novo estado em que deixa de aceitar os comandos para subir e descer. O elevador só abandona este estado após a chave de segurança (*String*) ser aplicada, regressando ao estado em que se encontrava quando o erro ocorreu.

Factory

- O padrão *Factory*, também designado por *Factory method*, prevê a disponibilização de um mecanismo (método) que permite a criação de uma instância de uma classe entre um determinado conjunto de possibilidades
  - Normalmente é aplicado num contexto em que existe uma hierarquia, da qual derivam um conjunto de subclasses, sendo este padrão usado como mecanismo para criar uma instância de uma dessas subclasses

Factory

- Implementação usual
  - Define-se um método estático
    - Na classe base da hierarquia onde se encontram os tipos de objeto a criar
    - Numa classe criada para esse efeito (por exemplo: *`ElevatorStateFactory`*)
  - ... recebe como parâmetros os dados necessários à identificação do tipo de objeto pretendido e parâmetros necessários à sua criação
  - ... o tipo de retorno do método será a da base da hierarquia que abrange todos os tipos de objeto que o método pode criar

Factory

```
public interface IElevatorState {
    boolean up();
    boolean down();
    boolean useSecurityKey(); // b)

    ElevatorState getState();

    static IElevatorState createState(ElevatorState type,
                                      ElevatorContext context, Elevator elevator) {
        return switch (type) {
            case GROUND_FLOOR -> new GroundFloorState(context, elevator);
            case FIRST_FLOOR -> new FirstFloorState(context, elevator);
            case SECOND_FLOOR -> new SecondFloorState(context, elevator);
            case UNDER_MAINTENANCE -> new UnderMaintenanceState(context, elevator);
            //default -> null;
        };
    }
}
```

Factory

```
class ElevatorStateFactory {
   static IElevatorState createState(ElevatorState type,
                                     ElevatorContext context, Elevator elevator) {
      return switch (type) {
         case GROUND_FLOOR -> new GroundFloorState(context, elevator);
         case FIRST_FLOOR -> new FirstFloorState(context, elevator);
         case SECOND_FLOOR -> new SecondFloorState(context, elevator);
         case UNDER_MAINTENANCE->new UnderMaintenanceState(context, elevator);
         //default -> null;
      };
   }
}
```

Factory

- O padrão pode ser implementado de muitas outras formas
  - Por exemplo, recorrendo a métodos de instância nas subclasses, para permitir a sua redefinição em classes derivadas (para incluir outros tipos de objeto na *factory*)

- Método alternativo:
  - Sendo necessário existir uma forma de identificação do tipo de objeto a criar, pode-se usar um *enum* para identificar o tipo de objeto pretendido
  - Como em Java os *enums* podem incluir a definição de métodos, esse poderá ser um enquadramento interessante para implementar a *factory*

Factory – exemplo com *enum* (1)

```
public enum ElevatorState {
  GROUND_FLOOR, FIRST_FLOOR, SECOND_FLOOR, UNDER_MAINTENANCE;

  // factory - Classic implementation of factory pattern
  static IElevatorState createState(ElevatorState type,
                                    ElevatorContext context, Elevator elevator) {
    return switch (type) {
       case GROUND_FLOOR -> new GroundFloorState(context, elevator);
       case FIRST_FLOOR -> new FirstFloorState(context, elevator);
       case SECOND_FLOOR -> new SecondFloorState(context, elevator);
       case UNDER_MAINTENANCE->new UnderMaintenanceState(context, elevator);
       // default -> null;
    };
  }
}
```

Factory – exemplo com *enum* (2)

```
public enum ElevatorState {
  GROUND_FLOOR, FIRST_FLOOR, SECOND_FLOOR, UNDER_MAINTENANCE;

  // factory - instance method
  IElevatorState createState(ElevatorContext context, Elevator elevator) {
     return switch (this) {
        case GROUND_FLOOR -> new GroundFloorState(context, elevator);
        case FIRST_FLOOR -> new FirstFloorState(context, elevator);
        case SECOND_FLOOR -> new SecondFloorState(context, elevator);
        case UNDER_MAINTENANCE->new UnderMaintenanceState(context, elevator);
     };
  }
}
```

Singleton

- O padrão *Singleton* permite garantir que só existe uma instância de uma determinada classe
  - Por exemplo, permite o acesso facilitado às funcionalidades do objeto sem haver a necessidade de passar a referência para o mesmo em chamadas sucessivas entre funções
  - Pode ser implementado com recurso a uma classe totalmente estática, mas
    - Não permite beneficiar de herança (extensão de classes ou implementação de interfaces)
    - Dificulta a herança das caraterísticas e eventual extensão através de classes derivadas

Singleton

- Implementação típica

```
class Singleton {
   private static Singleton instance = null;

   public static Singleton getInstance() {
      if (instance == null)
         instance = new Singleton();
      return instance;
   }

   protected Singleton() { ... }

   // other variables and methods
   // Example:
   private int count = 1234;
   public int getCount() { return count; }
}
```

- Forma de utilização: `int i = Singleton.getInstance().getCount();`

Exercício de aplicação

- Criar uma classe `ModelLog` no exercício 24 que permita registar *logs* diversos de situações que vão ocorrendo em todo modelo de dados: mudanças de piso, entrada/saída de manutenção, etc.
  - Implementar a classe segundo o modelo *Singleton*
  - Fornecer métodos para: adicionar nova mensagem, aceder a uma lista de todos os *logs* e um método para remover todas as mensagem
  - Na interface com o utilizador, a cada mudança de estado deverão ser mostradas todas as mensagens pendentes no log
    - Depois de mostradas todas as mensagens, as mensagens de *log* deverão ser eliminadas