(http://baeldung.com)

El patrón de comando en Java

Última modificación: 10 de mayo de 2018

> por baeldung (http://www.baeldung.com/author/baeldung/)
> (http://www.baeldung.com/author/baeldung/)

Java (http://www.baeldung.com/category/java/) +

Patrón (http://www.baeldung.com/category/pattern/)

1. Información general

El patrón de comando es un patrón de diseño de comportamiento y es parte de la lista formal de patrones de diseño del GoF (https://en.wikipedia.org/wiki/Design_Patterns). En pocas palabras, el patrón tiene la intención de **encapsular en un objeto todos los datos necesarios para realizar una acción (comando) determinada,** incluido el método para llamar, los argumentos del método y el objeto al que pertenece el método.

Este modelo nos permite **desacoplar objetos que producen los comandos de sus consumidores** , por eso es que el patrón se conoce comúnmente como el patrón productor-consumidor.

En este tutorial, aprenderemos cómo implementar el patrón de comando en Java utilizando enfoques orientados a objetos y funcionales de objeto, y veremos en qué casos de uso puede ser útil.

2. Implementación orientada a objetos

En una implementación clásica, el patrón de comando requiere **implementar cuatro componentes: el Comando, el Receptor, el Invitante y el Cliente** .

Para comprender cómo funciona el patrón y el rol que desempeña cada componente, creemos un ejemplo básico.

Supongamos que queremos desarrollar una aplicación de archivo de texto. En tal caso, debemos implementar toda la funcionalidad requerida para realizar algunas operaciones relacionadas con archivos de texto, como abrir, escribir, guardar un archivo de texto, etc.

Entonces, debemos dividir la aplicación en los cuatro componentes mencionados anteriormente.

2.1. Clases de comando

Un comando es un objeto cuya función es **almacenar toda la información requerida para ejecutar una acción** , incluido el método para llamar, los argumentos del método y el objeto (conocido como el receptor) que implementa el método.

Para tener una idea más precisa de cómo funciona un objeto de comando, comencemos a desarrollar una capa de comando simple que incluye una sola interfaz y dos implementaciones:

```
1  @FunctionalInterface
2  public interface TextFileOperation {
3      String execute();
4  }
```

```
public class OpenTextFileOperation implements TextFileOperation {

    private TextFile textFile;

    // constructors

    @Override
    public String execute() {
        return textFile.open();
    }
}
```

```
public class SaveTextFileOperation implements TextFileOperation {

    // same field and constructor as above

    @Override
    public String execute() {
        return textFile.save();
    }
}
```

En este caso, la interfaz *TextFileOperation* define la API de los objetos de comando, y las dos implementaciones, *OpenTextFileOperation* y *SaveTextFileOperation,* realizan las acciones concretas. El primero abre un archivo de texto, mientras que el segundo guarda un archivo de texto.

Está claro ver la funcionalidad de un objeto de comando: los comandos *TextFileOperation* **encapsulan toda la información requerida** para abrir y guardar un archivo de texto, incluyendo el objeto receptor, los métodos para llamar y los argumentos (en este caso, no se requieren argumentos, pero podrían ser).

Vale la pena destacar que **el componente que realiza las operaciones de archivo es el receptor (la instancia de *TextFile*)** .

2.2. La clase de receptor

Un receptor es un objeto que **realiza un conjunto de acciones cohesivas** . Es el componente que realiza la acción real cuando se llama al método *execute ()* del comando .

En este caso, necesitamos definir una clase de receptor, cuya función es modelar los objetos de *TextFile* :

```
public class TextFile {

    private String name;

    // constructor

    public String open() {
        return "Opening file " + name;
    }

    public String save() {
        return "Saving file " + name;
    }

    // additional text file methods (editing, writing, copying, past
}
```

2.3. La clase de Invoker

Un invocador es un objeto que **sabe cómo ejecutar un comando dado pero no sabe cómo se ha implementado el comando.** Solo conoce la interfaz del comando.

En algunos casos, el invocador también almacena y pone en cola los comandos, además de ejecutarlos. Esto es útil para implementar algunas características adicionales, como la grabación de macro o la funcionalidad deshacer y rehacer.

En nuestro ejemplo, se hace evidente que debe haber un componente adicional responsable de invocar los objetos de comando y ejecutarlos a través del método de *ejecución de* comandos *()* . **Aquí es exactamente donde la clase de invocador entra en juego** .

Veamos una implementación básica de nuestra invocador:

```
public class TextFileOperationExecutor {

    private final List<TextFileOperation> textFileOperations
     = new ArrayList<>();

    public String executeOperation(TextFileOperation textFileOperat
        textFileOperations.add(textFileOperation);
        return textFileOperation.execute();
    }
}
```

La clase *TextFileOperationExecutor* es solo una **capa delgada de abstracción que desacopla los objetos de comando de sus consumidores** y llama al método encapsulado dentro de los objetos del comando *TextFileOperation* .

En este caso, la clase también almacena los objetos de comando en una *lista* . Por supuesto, esto no es obligatorio en la implementación del patrón, a menos que necesitemos agregar un control adicional al proceso de ejecución de las operaciones.

2.4. La clase del cliente

Un cliente es un objeto que **controla el proceso de ejecución del comando** especificando qué comandos ejecutar y en qué etapas del proceso ejecutarlos.

Entonces, si queremos ser ortodoxos con la definición formal del patrón, debemos crear una clase de cliente utilizando el método *principal* típico :

```
public static void main(String[] args) {
    TextFileOperationExecutor textFileOperationExecutor
      = new TextFileOperationExecutor();
    textFileOperationExecutor.executeOperation(
      new OpenTextFileOperation(new TextFile("file1.txt"))));
    textFileOperationExecutor.executeOperation(
      new SaveTextFileOperation(new TextFile("file2.txt"))));
}
```

3. Implementación funcional del objeto

Hasta ahora, hemos utilizado un enfoque orientado a objetos para implementar el patrón de comando, que está muy bien.

Desde Java 8, podemos usar un enfoque funcional de objetos, basado en expresiones lambda y referencias de métodos, para **hacer que el código sea un poco más compacto y menos detallado**.

3.1. Usando Lambda Expressions

Como la interfaz *TextFileOperation* es una interfaz funcional (https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) , podemos **pasar objetos de comando en forma de expresiones lambda al invocador** , sin tener que crear las instancias de *TextFileOperation* explícitamente:

```
TextFileOperationExecutor textFileOperationExecutor
 = new TextFileOperationExecutor();
textFileOperationExecutor.executeOperation(() -> "Opening file file1.t
textFileOperationExecutor.executeOperation(() -> "Saving file file1.tx
```

La implementación ahora se ve mucho más racionalizada y concisa, ya que hemos **reducido la cantidad de código repetitivo** .

Aun así, la pregunta sigue en pie: ¿es este enfoque mejor, en comparación con el orientado a objetos?

Bueno, eso es complicado. Si suponemos que un código más compacto significa un mejor código en la mayoría de los casos, entonces sí lo es.

Como regla general, debemos evaluar caso por caso de uso cuando recurrir a expresiones lambda .

3.2. Usar referencias de método

Del mismo modo, podemos usar referencias de métodos para **pasar objetos de comando al invocador:**

```
TextFileOperationExecutor textFileOperationExecutor
 = new TextFileOperationExecutor();
TextFile textFile = new TextFile("file1.txt");
textFileOperationExecutor.executeOperation(textFile::open);
textFileOperationExecutor.executeOperation(textFile::save);
```

En este caso, la implementación es **un poco más detallada que la que usa lambdas** , ya que todavía teníamos que crear las instancias de *TextFile* .

4. Conclusión

En este artículo, aprendimos los conceptos clave del patrón de comando y cómo implementar el patrón en Java utilizando un enfoque orientado a objetos y una combinación de expresiones lambda y referencias de métodos.

Como es habitual, todos los ejemplos de código que se muestran en este tutorial están disponibles en GitHub (https://github.com/eugenp/tutorials/tree/master/patterns/behavioral-patterns/src/main/java/com/baeldung/pattern/command) .

Deja una respuesta

¡Sé el primero en comentar!

Start the discussion

Suscribir

CATEGORÍAS

PRIMAVERA (HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
DESCANSO (HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)

JAVA (HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SEGURIDAD (HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCIA (HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
HTTPCLIENT (HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTP://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIE

TUTORIAL "VOLVER A LO BÁSICO" DE JAVA (HTTP://WWW.BAELDUNG.COM/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (HTTP://WWW.BAELDUNG.COM/JACKSON)
TUTORIAL DE HTTPCLIENT 4 (HTTP://WWW.BAELDUNG.COM/HTTPCLIENT-GUIDE)
REST CON SPRING TUTORIAL (HTTP://WWW.BAELDUNG.COM/REST-WITH-SPRING-SERIES/)
TUTORIAL DE SPRING PERSISTENCE (HTTP://WWW.BAELDUNG.COM/PERSISTENCE-WITH-SPRING-SERIES/)
SEGURIDAD CON SPRING (HTTP://WWW.BAELDUNG.COM/SECURITY-SPRING)

ACERCA DE

ACERCA DE BAELDUNG (HTTP://WWW.BAELDUNG.COM/ABOUT/)
LOS CURSOS (HTTP://COURSES.BAELDUNG.COM)
TRABAJO DE CONSULTORÍA (HTTP://WWW.BAELDUNG.COM/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
EL ARCHIVO COMPLETO (HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)
ESCRIBIR PARA BAELDUNG (HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES)
CONTACTO (HTTP://WWW.BAELDUNG.COM/CONTACT)
INFORMACIÓN DE LA COMPAÑÍA (HTTP://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)
TÉRMINOS DE SERVICIO (HTTP://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)
POLÍTICA DE PRIVACIDAD (HTTP://WWW.BAELDUNG.COM/PRIVACY-POLICY)
EDITORES (HTTP://WWW.BAELDUNG.COM/EDITORS)
KIT DE MEDIOS (PDF) (HTTPS://S3.AMAZONAWS.COM/BAELDUNG.COM/BAELDUNG+-+MEDIA+KIT.PDF)