6.20 Mẫu thiết kế COMMAND

6.20.1 Mở đầu

Mẫu thiết kế *Command* là mẫu thiết kế hành vi và trợ giúp phân tách người triệu gọi khỏi người nhận được báo cáo.

Để hiểu mẫu thiết kế *Command*, chúng ta sẽ tạo một ví dụ mà thực hiện các kiểu công việc khác nhau. Một công việc có thể là bất cứ điều gì trong hệ thống, chẳng hạn, gửi thư điện tử, nhắn tin SMS, ghi lưu logging hay thực hiện các chức năng IO.

Mẫu *Command* sẽ giúp phân tách người triệu gọi và người nhận và trợ giúp để thực hiện bất cứ kiểu công việc nào mà không cần biết về cài đặt của nó. Giả sử chúng ta làm cho ví dụ này thú vị hơn bằng cách tạo ra các luồng mà sẽ giúp thực hiện các công việc một cách đồng thời. Như các công việc này là độc lập với nhau, vì vậy dãy thực hiện các công việc này thực tế là không quan trọng, Chúng ta sẽ tạo một bể luồng để giới hạn số luồng thực hiện các công việc. Đối tượng *Command* sẽ đóng gói các công việc và sẽ truyền nó cho luồng từ bể này mà sẽ thực thi công việc đó.

Trước khi cài đặt Ví dụ này, chúng ta sẽ tìm hiểu thêm về mẫu thiết kế *Command.*

6.20.2 Mẫu thiết kế Command là gì

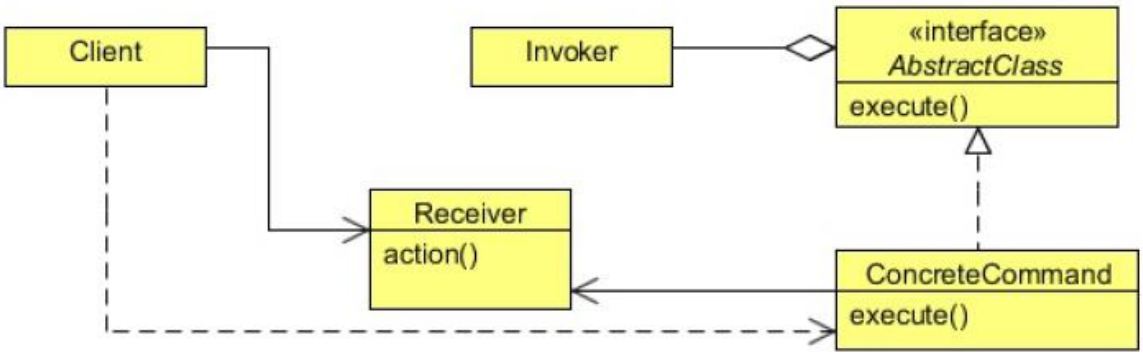
Mục đích của mẫu thiết kế *Command* là đóng gói yêu cầu như một đối tượng, do đó cho phép người phát triển tham số hóa *client* với các yêu cầu khác nhau, hàng đợi hoặc các yêu cầu ghi lại và hỗ trợ các thao tác mà *undo* được.

Nói chung, ứng dụng lập trình hướng đối tượng bao gồm một tập các đối tượng tương tác, mỗi cái đưa ra chức năng chuyên biệt và hạn chế. Để phản hồi tương tác của người sử dụng, ứng dụng tiến hành một kiểu xử lý nào đó. Để làm việc đó, ứng dụng sử dụng dịch vụ của các đối tượng khác nhau để xử lý yêu cầu.

Theo thuật ngữ cài đặt, một ứng dụng có thể phụ thuộc vào đối tượng dành riêng mà triệu gọi các phương thức trên các đối tượng này bằng việc truyền dữ liệu được yêu cầu như các đối số. Đối tượng dành riêng này có thể được tham chiếu đến như người triệu gọi mà nó triệu gọi các thao tác trên các đối tượng khác nhau. Người triệu gọi *Invoker* có thể được coi như một phần của ứng dụng client. Tập các đối tượng mà thực tế chứa cài đặt để đưa ra các dịch vụ đáp ứng cho việc xử lý yêu cầu được tham chiếu như các đối tượng nhận *Receiver*.

Sử dụng mẫu *Command, Invoker* mà đưa ra các yêu cầu thay mặt *client* và tập các đối tượng *Receiver* cho thuê dịch vụ cần là tách bạch nhau. Mẫu *Command* đề xuất tạo một trừu tượng cho việc xử lý được tiến hành hoặc hành động được thực hiện để đáp ứng yêu cầu của *client.* Trừu tượng này cần được thiết kế để khai báo giao diện chung được cài đặt bởi các *concrete implementer* khác nhau được tham chiếu như các đối tượng *Command*. Mỗi đối tượng *Command* thể hiện một kiểu yêu cầu của client và một xử lý tương ứng.

Một đối tượng *Command* cho trước là chịu trách nhiệm đưa ra một chức năng đáp ứng để xử lý một yêu cầu mà nó thể hiện, nhưng nó không chứa cài đặt thực tế của chức năng đó. Các đối tượng *Command* sử dụng các đối tượng Receiver để xuất chức năng đó.



Command

- Khai báo giao diện cho việc thực hiện một thao tác.

ConcreteCommand

- Định nghĩa gắn kết giữa một đối tượng *Receiver* và một hành động
- Cài đặt *Execute* bằng việc triệu gọi thao tác tương ứng trên *Receiver*

Client

- Tạo một đối tượng *ConcreteCommand* và thiết lập *Receiver* của nó.

Invoker

- Yêu cầu một *command* để xử lý một yêu cầu.

Receiver

- Biết thực hiện thao tác liên kết nào để xử lý được yêu cầu đó. Lớp bất kỳ có thể được dùng như lớp *Receiver*.

6.20.3 Cài đặt mẫu Command

Chúng ta sẽ cài đặt ví dụ sử dụng đối tượng *Command*. Một đối tượng *Command* sẽ được tham chiếu bằng giao diện chung và sẽ chứa một phương thức mà sẽ được sử dụng để thực hiện yêu cầu. Các lớp *command* cụ thể sẽ ghi đè phương thức này và sẽ cung cấp cài đặt chuyên biệt của riêng họ để thực hiện yêu cầu đó.

```
package com.javacodegeeks.patterns.commandpattern;

public interface Job {

        public void run();
}
```

Giao diện Job là giao diện *Command*, chứa một phương thức đơn giản là *run*, mà được thực hiện bởi một luồng. Phương thức *execute* của *command* của chúng ta là phương thức *run* mà sẽ được sử dụng để thực thi bởi một luồng để hoàn thành công việc.

Ở đây cần có một kiểu job khác mà cần được thực hiện. Sau đây là các lớp cụ thể khác nhau mà khởi tạo của chúng sẽ được thực thi bởi các đối tượng *command* khác nhau.

```
package com.javacodegeeks.patterns.commandpattern;

public class Email {

        public void sendEmail(){
                System.out.println("Sending email.......");
        }
}
```

```
package com.javacodegeeks.patterns.commandpattern;

public class FileIO {

        public void execute(){
                System.out.println("Executing File IO operations...");
        }
}
```

```
package com.javacodegeeks.patterns.commandpattern;

public class Logging {

        public void log(){
                System.out.println("Logging...");
        }
}
```

```
package com.javacodegeeks.patterns.commandpattern;

public class Sms {

        public void sendSms(){
                System.out.println("Sending SMS...");
        }
}
```

Sau đây là các lớp *Command* khác nhau mà đóng gói các lớp trên và cài đặt giao diên *Job*.

```
package com.javacodegeeks.patterns.commandpattern;

public class EmailJob implements Job{

	private Email email;

	public void setEmail(Email email){
		this.email = email;
	}

	@Override
	public void run() {
		System.out.println("Job ID: "+Thread.currentThread().getId()+" executing ↩
		    email jobs.");
		if(email!=null){
			email.sendEmail();
		}

		try {
			Thread.sleep(1000);
		} catch (InterruptedException e) {
			Thread.currentThread().interrupt();
		}

	}

}
```

```
package com.javacodegeeks.patterns.commandpattern;

public class FileIOJob implements Job{

	private FileIO fileIO;

	public void setFileIO(FileIO fileIO){
		this.fileIO = fileIO;
	}

	@Override
	public void run() {
		System.out.println("Job ID: "+Thread.currentThread().getId()+" executing ↩
		    fileIO jobs.");
		if(fileIO!=null){
			fileIO.execute();
		}

		try {
			Thread.sleep(1000);
		} catch (InterruptedException e) {
			Thread.currentThread().interrupt();
		}

	}
}
```

```
package com.javacodegeeks.patterns.commandpattern;

public class LoggingJob implements Job{

        private Logging logging;

        public void setLogging(Logging logging){
                this.logging = logging;
        }

        @Override
        public void run() {
                System.out.println("Job ID: "+Thread.currentThread().getId()+" executing ↩
                    logging jobs.");
                if(logging!=null){
                        logging.log();
                }

                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                }

        }
}
```

```
package com.javacodegeeks.patterns.commandpattern;
```

```
public class SmsJob implements Job{

        private Sms sms;

        public void setSms(Sms sms) {
                this.sms = sms;
        }


        @Override
        public void run() {
                System.out.println("Job ID: "+Thread.currentThread().getId()+" executing ↩
                    sms jobs.");
                if(sms!=null){
                        sms.sendSms();
                }

                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                }

        }

}
```

Các lớp trên giữ tham chiếu đến các lớp tương ứng của chúng, mà sẽ được sử dụng để hoàn thành công việc. Các lớp ghi đè phương thức *run* và thực hiện công việc được yêu cầu. Chẳng hạn, lớp *SmsJob* được sử dụng để gửi *sms*, phương thức *run* của nó gọi phương thức *sendSms* của đối tượng *Sms* để hoàn thành công việc.

Bạn có thể đặt các đối tượng khác nhau cái nọ cạnh cái kia như cùng là đối tượng *Command.*

Dưới đây là lớp *ThreadPool* được sử dụng để tạo bể các luồng và cho phép một luồng được đẩy vào và thực thi công việc từ hàng đợi công việc.

```
package com.javacodegeeks.patterns.commandpattern;

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public class ThreadPool {

        private final BlockingQueue<Job> jobQueue;
        private final Thread[] jobThreads;
        private volatile boolean shutdown;

        public ThreadPool(int n)
        {
                jobQueue = new LinkedBlockingQueue<>();
                jobThreads = new Thread[n];

                for (int i = 0; i < n; i++) {
                        jobThreads[i] = new Worker("Pool Thread " + i);
                        jobThreads[i].start();
                }
        }

        public void addJob(Job r)
        {
                try {
                        jobQueue.put(r);
```

```
                } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                }
        }

        public void shutdownPool()
        {
                while (!jobQueue.isEmpty()) {
                        try {
                                Thread.sleep(1000);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
                shutdown = true;
                for (Thread workerThread : jobThreads) {
                        workerThread.interrupt();
                }
        }

        private class Worker extends Thread
        {
                public Worker(String name)
                {
                        super(name);
                }

                public void run()
                {
                        while (!shutdown) {
                                try {
                                        Job r = jobQueue.take();
                                        r.run();
                                } catch (InterruptedException e) {
                                }
                        }
                }
        }

}
```

Lớp trên được sử dụng để tạo *n* luồng (các luồng làm việc). Mỗi *worker thread* sẽ chờ một công việc trong hàng đợi và sau đó thực thi công việc và sẽ trở lại trạng thái chờ đợi. Lớp này chứa hàng đợi công việc; khi một công việc mới sẽ được bổ sung vào hàng đợi, một *worker* từ bể sẽ thực thi công việc này.

Chúng ta sẽ bổ sung phương thức *shutdownPool* mà sẽ được sử dụng để tắt *Pool* bằng cách ngắt mọi luồng làm việc chỉ khi hàng đợi công việc là rỗng. Phương thức *addJob* được sử dụng để bổ sung công việc vào hàng đợi.

Bây giờ chúng ta kiểm tra code.

```
package com.javacodegeeks.patterns.commandpattern;


public class TestCommandPattern {
        public static void main(String[] args)
    {
        init();
    }

    private static void init()
    {
        ThreadPool pool = new ThreadPool(10);
```

```
        Email email = null;
        EmailJob  emailJob = new EmailJob();

        Sms sms = null;
        SmsJob smsJob = new SmsJob();

        FileIO fileIO = null;;
        FileIOJob fileIOJob = new FileIOJob();

        Logging logging = null;
        LoggingJob logJob = new LoggingJob();

        for (int i = 0; i < 5; i++) {
                email = new Email();
                emailJob.setEmail(email);

                sms = new Sms();
                smsJob.setSms(sms);

                fileIO = new FileIO();
                fileIOJob.setFileIO(fileIO);

                logging = new Logging();
                logJob.setLogging(logging);

            pool.addJob(emailJob);
            pool.addJob(smsJob);
            pool.addJob(fileIOJob);
            pool.addJob(logJob);
        }
        pool.shutdownPool();
    }

}
```

Code trên cho kết quả đầu ra sau:

```
Job ID: 9 executing email jobs.
Sending email.......
Job ID: 12 executing logging jobs.
Job ID: 17 executing email jobs.
Sending email.......
Job ID: 13 executing email jobs.
Sending email.......
Job ID: 10 executing sms jobs.
Sending SMS...
Job ID: 11 executing fileIO jobs.
Executing File IO operations...
Job ID: 18 executing sms jobs.
Sending SMS...
Logging...
Job ID: 16 executing logging jobs.
Logging...
Job ID: 15 executing fileIO jobs.
Executing File IO operations...
Job ID: 14 executing sms jobs.
Sending SMS...
Job ID: 12 executing fileIO jobs.
Executing File IO operations...
Job ID: 10 executing logging jobs.
Logging...
Job ID: 18 executing email jobs.
```

```
Sending email.......
Job ID: 16 executing sms jobs.
Sending SMS...
Job ID: 14 executing fileIO jobs.
Executing File IO operations...
Job ID: 9 executing logging jobs.
Logging...
Job ID: 17 executing email jobs.
Sending email.......
Job ID: 13 executing sms jobs.
Sending SMS...
Job ID: 15 executing fileIO jobs.
Executing File IO operations...
Job ID: 11 executing logging jobs.
Logging...
```

Lưu ý rằng đầy ra có thể khác nhau do thứ tự thực hiện các luồng.

Trong lớp trên, chúng ta đã tạo một bể luồng gồm 10 luồng. Sau đó, chúng ta thiết lập các đối tượng *command* khác nhau với các công việc khác nhau và bổ sung các công việc này vào hàng đợi sử dụng phương thức *addJob* của lớp *ThreadPool.* Như chúng ta đã thấy, một công việc được chèn vào hàng đợi, một luồng thực thi công việc đó và xóa nó khỏi hàng đợi.

Chúng ta đã đặt kiểu khác nhau của công việc, nhưng sử dụng mẫu thiết kế *Command*, chúng ta tách bạch công việc khỏi luồng triệu gọi. Một luồng sẽ thực thi bất cứ kiểu đối tượng nào mà cài đặt giao diện *Job*. Các đối tượng *command* khác nhau đóng gói đối tượng khác nhau và thực thi thao tác được yêu cầu trên các đối tượng đó.

Đầu ra chỉ ra rằng các luồng khác nhau thực thi công việc khác nhau. Bằng cách quan sát id công việc ở đầu ra, bạn có thể thấy rõ là mỗi luồng đơn thực hiện nhiều hơn một công việc. Điều này xảy ra vì sau khi thực hiện xong một công việc luồng được gửi trở lại bể.

Ưu điểm của mẫu thiết kế *Command* là bạn có thể bổ sung nhiều kiểu công việc khác mà không thay đổi các lớp đã tồn tại. Nó dẫn đến tính linh hoạt và dễ bảo trì hơn và cũng giảm cơ hội lỗi trong code.

6.20.4 Khi nào sử dụng mẫu thiết kế Command

Sử dụng mẫu *Command* khi bạn muốn:

- Tham số hóa các đối tượng bởi hành động thực hiện.
- Đặc tả, hàng đợi, và thực thi các yêu cầu tại các thời điểm khác nhau. Đối tượng *Command* có thể có thời gian sống độc lập với yêu cầu gốc. Nếu *receiver* của yêu cầu có thể được thể hiện theo cách độc lập với không gian địa chỉ, thì bạn có thể truyền đối tượng *command* cho yêu cầu đến tiến trình khác nhau và thực hiện yêu cầu ở đó.
- Hỗ trợ *undo*. Thao tác *Execute* của *Command* có thể được lưu giữ trạng thái cho việc đảo lại các tác động trong bản thân *command*. Giao diện *Command* cần được bổ sung thao tác *Un-execute* mà đảo lại các tác động của lời gọi *Execute* trước. Các *command* đã thực hiện là được lưu giữ trong danh sách lịch sử. *Undo* mức không hạn chế và *redo* là đạt được bởi viếng thăm danh sách đó ngược chiều hay xuôi chiều khi gọi *Un-execute* hay *Execute* tương ứng.
- Hỗ trợ ghi lại các thay đổi sao cho chúng có thể được áp dụng lại trong trường hợp hệ thống bị sập. Bằng cách làm gia tăng giao diện *Command* với việc tải và lưu giữ các thao tác, bạn có thể giữ log thay đổi một cách bền vững. Khôi phục từ lỗi sập bao gồm tải lại các *logged command* từ đĩa và *re-executing* chúng với thao tác *Execute*.
- Cấu trúc một hệ thống quanh các thao tác bậc cao được xây dựng trên các thao tác nguyên thủy. Một cấu trúc như vậy là phổ cập trong các hệ thống thông tin mà hỗ trợ các giao dịch. Một giao dịch đóng gói một tập các thay đổi đến dữ liệu. Mẫu *Command* đưa ra cách để mô hình các giao dịch. *Commands* có giao diện chung, cho phép bạn triệu gọi mọi giao dịch theo cách như nhau. Mẫu này cũng làm cho nó dễ mở rộng hệ thống với các giao dịch mới.