6.11 Mẫu thiết kế FLYWEIGHT

6.11.1 Mẫu thiết kế Flyweight

Lập trình hướng đối tượng làm cho lập trình dễ dàng và thú vị. Nó làm cho công việc lập trình dễ dàng hơn bởi mô hình đưa các thực thể thế giới thực vào thế giới lập trình. Lập trình viên tạo ra một lớp và khởi tạo nó bằng việc tạo ra đối tượng của nó. Đối tượng này mô hình một thực thể của thế giới thực và các đối tượng bên trong ứng dụng tương tác với nhau để hoàn thành một công việc được yêu cầu.

Nhưng đôi khi quá nhiều đối tượng có thể làm chậm mọi việc. Nhiều đối tượng có thể chiếm bộ nhớ lớn và có thể làm chậm ứng dụng hoặc ngay cả gặp vấn đề về vượt quá bộ nhớ. Như một lập trình viên tốt, bạn cần theo dõi các đối tượng khởi tạo và kiểm soát việc tạo đối tượng trong ứng dụng. Điều này đặc biệt đúng, khi ta có nhiều đối tượng tương tự và hai đối tượng cùng loại không có sự khác biệt nhiều lắm giữa chúng.

Đôi khi các đối tượng trong ứng dụng có thể có nhiều đặc tính giống nhau và là kiểu tương tự (kiểu tương tự ở đây nghĩa là hầu hết các tính chất của chúng có cùng giá trị và chỉ có một số ít khác nhau về giá trị). Có lúc, chúng cũng là các đối tượng nặng để tạo ra, chúng cần được kiểm soát bởi người phát triển ứng dụng. Ngược lại, chúng có thể chiếm nhiều bộ nhớ và tất nhiên làm chậm toàn bộ ứng dụng.

Mẫu Flyweight được thiết kế để kiểm soát việc tạo đối tượng kiểu như vậy và cung cấp cho bạn một cơ chế lưu giữ cơ bản. Nó cho phép bạn tạo một đối tượng cho mỗi kiểu (kiểu ở đây phân biệt bởi tính chất của đối tượng đó) và nếu bạn yêu cầu một đối tượng với tính chất giống đã có, thì nó sẽ trả về cùng đối tượng đó thay vì tạo đối tượng mới.

Trước khi đi sâu tìm hiểu mẫu Flyweight, chúng ta sẽ xét kịch bản sau: một trang web cho phép người sử dụng lập trình và chạy chương trình trực tuyến trên nhiều ngôn ngữ khác nhau. Chúng ta sẽ bàn về kịch bản này bây giờ và sẽ giải quyết vấn đề sử dụng mẫu Flyweight.

Trang lập trình X-programming cho phép người sử dụng tạo và chạy các chương trình sử dụng ngôn ngữ ưa thích của mình. Nó cung cấp cho bạn nhiều lựa chọn ngôn ngữ lập trình khác nhau. Bạn chọn một, viết chương trình với nó và chạy nó để xem kết quả.

Nhưng bây giờ Trang này bắt đầu mất dần người sử dụng của nó, nguyên nhân là sự chậm trễ của Trang. Người sử dụng không quan tâm đến nó nữa. Trang rất nổi tiếng và đôi khi ở đây có hàng ngàn lập trình viên sử dụng nó. Vì vậy, Trang chạy như bò. Nhưng việc sử dụng quá tải không phải là vấn đề đằng sau sự chậm trễ của Trang. Chúng ta sẽ xem code lập trình cốt lõi của Trang mà cho phép người sử dụng chạy và thực thi các chương trình của họ và vấn đề ở đâu sẽ được nhận biết.

```
package com.javacodegeeks.patterns.flyweightpattern;

public class Code {

        private String code;

        public String getCode() {
                return code;
        }

        public void setCode(String code) {

                this.code = code;
        }

}
```

Lớp trên được sử dụng để đặt *code* được tạo bởi lập trình viên để làm cho nó chạy. Đối tượng *code* là đối tượng nhẹ có tính chất *code* với các phương thức *set* và *get*.

```
package com.javacodegeeks.patterns.flyweightpattern;

public interface Platform {

        public void execute(Code code);
}
```

Giao diện *Platform* được cài đặt bởi nền tảng chuyên biệt của một ngôn ngữ để thực thi *code*. Nó có một phương thức *execute*, mà dùng đối tượng *code* như tham số.

```
package com.javacodegeeks.patterns.flyweightpattern;

public class JavaPlatform implements Platform {

        public JavaPlatform(){
                System.out.println("JavaPlatform object created");
        }

        @Override
        public void execute(Code code) {
                System.out.println("Compiling and executing Java code.");
        }

}
```

Lớp trên cài đặt giao diện *Platform* và cung cấp cài đặt cho phương thức *excecute* để thực thi *code* trong Java.

Để thực thi *code*, đối tượng Code mà chứa *code* và đối tượng *Platform* sẽ thực thi *code* được tạo. Code để chạy trông như sau:

```
Platform platform = new JavaPlatform();
platform.execute(code);
```

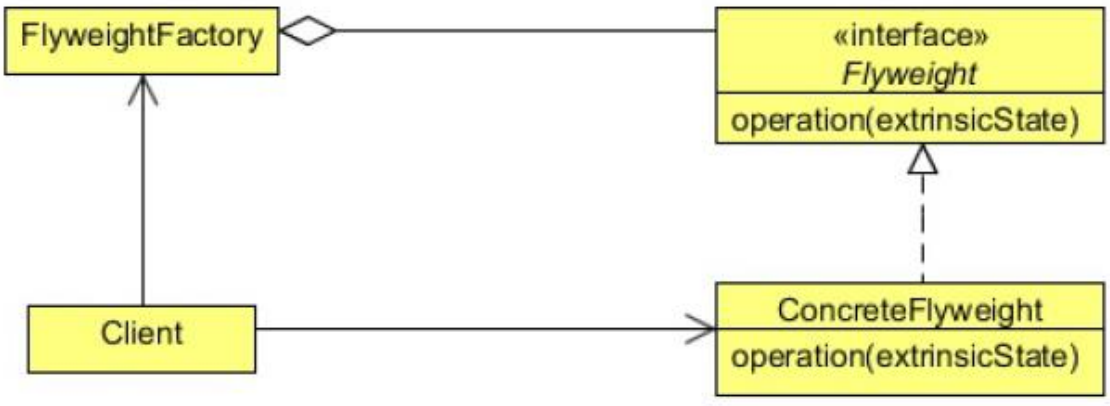
Bây giờ giả sử, có khoảng 2k người sử dụng trực tuyến và chạy *code* của họ mà theo đó có 2k đối tượng *code* và 2k đối tượng *platform*. Các đối tượng *code* là nhẹ và cần phải có mỗi đối tượng *code* cho mỗi người sử dụng. Nhưng, *Platform* là đối tượng nặng mà được sử dụng để tạo môi trường thực thi. Tạo quá nhiều đối tượng *Platfrom* là tốn thời gian và là nhiệm vụ khó khăn. Chúng ta cần kiểm soát việc tạo đối tượng *Platform* mà có thể được thực hiện sử dụng mẫu *Flyweight*, nhưng trước hết chúng ta tìm hiểu mẫu *Flyweight*.

6.11.2 Mẫu Flyweight là gì

Mục đích của mẫu *Flyweight* là để sử dụng các đối tượng chia sẻ mà hỗ trợ số lượng lớn các đối tượng phân loại nhỏ một cách hiệu quả. *Flyweight* là đối tượng chia sẻ mà có thể được sử dụng trong nhiều ngữ cảnh một cách đồng thời. *Flyweight* hoạt động như đối tượng độc lập trong mỗi ngữ cảnh – không phân biệt được việc tạo ra đối tượng mà nó không chia sẻ. *Flyweight* không tạo ra các điều kiện về ngữ cảnh mà ở đó chúng vận hành. Khái niệm chính ở đây là sự phân biệt giữa trạng thái bên trong và bên ngoài. Trạng thái bên trong là được lưu giữ trong *Flyweight*, nó chứa các thông tin mà độc lập với ngữ cảnh của *flyweight*, do đó tạo nên nó chia sẻ được. Trạng thái bên ngoài (*extrinsic state*) phụ thuộc vào và đa dạng với ngữ cảnh *flyweight* và do đó không chia sẻ được. *Client object* có trách nhiệm truyền trạng thái bên ngoài cho *flyweight* khi nó cần đến nó.

Xét kịch bản ứng dụng mà bao gồm tạo ra số lượng lớn các đối tượng mà là duy nhất chỉ theo giá trị một số ít tham số. Nói cách khác, các đối tượng này chứa dữ liệu bên trong và bất biến mà là chung giữa mọi đối tượng. Dữ liệu bên trong cần được tạo và duy trì như một phần của mỗi đối tượng mà được tạo ra. Việc tạo tổng thể và duy trì nhóm lớn các đối tượng như vậy có thể là rất đắt theo chi phí bộ nhớ và hiệu năng. Mẫu *Flyweight* có thể được sử dụng trong mỗi kich bản để thiết kế cách tạo đối tượng hiệu quả hơn.

Đây là sơ đồ lớp của mẫu *Flyweight*:



Flyweight

- Khai báo giao diện qua đó các *flyweight* có thể nhận được và hoạt động trên trạng thái ngoài.

Concrete Flyweight

- Cài đặt giao diện *Flyweight* và bổ sung lưu giữ cho trạng thái bên trong. Một đối tượng *ConcreteFlyweight* cần phải là chia sẻ được. Bất cứ trạng thái mà nó lưu giữ cần là bên trong, mà là độc lập với ngữ cảnh của đối tượng *ConcreteFlyweight*.

FlyweightFactory

- Tạo và quản lý các đối tượng *flyweight*
- Tin tưởng rằng *flyweight* là chia sẻ đúng đắn. Khi một *client* yêu cầu một *flyweight*, đối tượng *FlyweightFactory* cung cấp khởi tạo đã có hoặc tạo ra cái mới, nếu nó chưa có.

Client

- Duy trì tham chiếu đến *flyweight*
- Tính hoặc lưu trạng thái ngoài của *flyweight*

6.11.3 Giải pháp cho bài toán

Để giải bài toán trên, chúng ta sẽ tạo lớp Platform Factory mà sẽ kiểm soát việc tạo các đối tượng *Platform.*

```
package com.javacodegeeks.patterns.flyweightpattern;

import java.util.HashMap;
import java.util.Map;

public final class PlatformFactory {
```

```
        private static Map<String, Platform> map = new HashMap<>();
        private PlatformFactory(){
                throw new AssertionError("Cannot instantiate the class");
        }

        public static synchronized Platform getPlatformInstance(String platformType){
                Platform platform = map.get(platformType);
                if(platform==null){
                        switch(platformType){
                                case "C" : platform = new CPlatform();
                                                break;
                                case "CPP" : platform = new CPPPlatform();
                                                break;
                                case "JAVA" : platform = new JavaPlatform();
                                                break;
                                case "RUBY" : platform = new RubyPlatform();
                                                break;
                        }
                        map.put(platformType, platform);
                }
                return platform;
        }

}
```

Lớp trên chứa *map* tĩnh mà giữ đối tượng *String* như khóa và đối tượng *Platform* như giá trị của nó. Chúng ta không muốn tạo khởi tạo của lớp này nên chỉ giữ hàm tạo của nó là *private* và đưa ra báo lỗi *AssertionError* chỉ để tránh bất cứ việc tạo lỗi nào của đối tượng đó ngay cả bên trong lớp.

Phương thức chính và chỉ một phương thức của lớp này là *getPlatformInstance*. Nó là phương thức tĩnh mà có *PlatformType* là tham số. *PlatformType* này được sử dụng như khóa của *map*, trước hết nó kiểm tra *map* xem có đối tượng platform nào mà có khóa *key* đã tồn tại hay chưa. Nếu không có đối tượng như vậy được tìm thấy, *platform* thích hợp sẽ được tạo, nó sẽ đặt vào trong *map* và sau đó phương thức này sẽ trả về đối tượng đó. Lần sau, khi cùng một đối tượng kiểu *Platform* được yêu cầu, đối tượng tồn tại sẽ được trả về, thay vì tạo mới.

Cũng lưu ý rằng phương thức *getPlatformInstance* là *synchronized* để cung cấp an toàn luồng khi kiểm tra và tạo khởi tạo của đối tượng. Trong ví dụ trên, không có tính chất bên trong nào của đối tượng này mà được chia sẻ, nhưng chỉ có tính chất bên ngoài mà là đối tượng code được cung cấp bởi *client code*.

Bây giờ ta sẽ kiểm tra *code* trên.

```
package com.javacodegeeks.patterns.flyweightpattern;

public class TestFlyweight {

        public static void main(String[] args) {

                Code code = new Code();
                code.setCode("C Code...");
                Platform platform = PlatformFactory.getPlatformInstance("C");
                platform.execute(code);
                System.out.println("*************************");
                code = new Code();
                code.setCode("C Code2...");
                platform = PlatformFactory.getPlatformInstance("C");
                platform.execute(code);
                System.out.println("*************************");
                code = new Code();
                code.setCode("JAVA Code...");
                platform = PlatformFactory.getPlatformInstance("JAVA");
                platform.execute(code);
                System.out.println("*************************");
                code = new Code();
                code.setCode("JAVA Code2...");
                platform = PlatformFactory.getPlatformInstance("JAVA");
                platform.execute(code);
                System.out.println("*************************");
                code = new Code();
                code.setCode("RUBY Code...");
                platform = PlatformFactory.getPlatformInstance("RUBY");
                platform.execute(code);
                System.out.println("*************************");
                code = new Code();
                code.setCode("RUBY Code2...");
                platform = PlatformFactory.getPlatformInstance("RUBY");
                platform.execute(code);
        }

}
```

Đoạn *code* trên sẽ cho kết quả đầu ra như sau:

```
CPlatform object created
Compiling and executing C code.
************************
Compiling and executing C code.
************************
JavaPlatform object created
Compiling and executing Java code.
************************
Compiling and executing Java code.
************************
RubyPlatform object created
Compiling and executing Ruby code.
************************
Compiling and executing Ruby code.
```

Trong lớp trên, trước hết ta tạo đối tượng code và đặt nó là *C code* trong đó. Sau đó ta yêu cầu *PlatformFactory* cung cấp platform để thực thi code này. Sau đó, ta gọi phương thức *excecute* trên đối tượng trả về, truyền đối tượng *code* cho nó.

Chúng ta thực hiện cùng thủ tục như vậy, tức là tạo và đặt đối tượng Code đó và sau đó yêu cầu đối tượng *Platform*, chuyên biệt cho code này. Đầu ra chỉ rõ rằng đối tượng *platform* chỉ được tạo lần đầu khi được yêu cầu, lần sau nó trả về cùng đối tượng đã có.

Các lớp chuyên biệt nền tảng khác cũng tương tự như *JavaPlatform* cũng được nêu ra.

```
package com.javacodegeeks.patterns.flyweightpattern;

public class CPlatform implements Platform {

        public CPlatform(){
                System.out.println("CPlatform object created");
        }

        @Override
        public void execute(Code code) {
                System.out.println("Compiling and executing C code.");
        }

}

package com.javacodegeeks.patterns.flyweightpattern;
```

```
public class CPPPlatform implements Platform{

        public CPPPlatform(){
                System.out.println("CPPPlatform object created");
        }

        @Override
        public void execute(Code code) {
                System.out.println("Compiling and executing CPP code.");
        }

}

package com.javacodegeeks.patterns.flyweightpattern;

public class RubyPlatform implements Platform{

        public RubyPlatform(){
                System.out.println("RubyPlatform object created");
        }

        @Override
        public void execute(Code code) {
                System.out.println("Compiling and executing Ruby code.");
        }

}
```

Nếu chúng ta xét có 2k người sử dụng đồng thời cùng sử dụng Trang này, chính xác 2k đối tượng *code* nhẹ sẽ được tạo nhưng chỉ 4 đối tượng *platform* nặng được khởi tạo. Lưu ý rằng, chúng ta nói bốn đối tượng platform bởi có tối thiểu một người nào đó sử dụng một ngôn ngữ. Nếu không có ai sử dụng *Ruby*, thì chỉ có 3 đối tượng nền tảng được tạo.

6.11.4 Khi nào sử dụng mẫu Flyweight

Tính hiệu quả của mẫu *Flyweight* phụ thuộc nhiều vào việc làm như thế nào và ở đâu chúng được sử dụng. Áp dụng mẫu *Flyweight* khi mọi điều sau đề đúng:

- Ứng dụng sử dụng số lượng lớn các đối tượng.
- Giá lưu trữ là cao vì số lượng lớn các đối tượng.
- Hầu hết trạng thái đối tượng có thể được tạo bên ngoài.
- Nhiều nhóm các đối tượng có thể được thay thế bằng một số ít các đối tượng chia sẻ khi trạng thái bên ngoài được hủy bỏ.
- Ứng dụng này không phụ thuộc vào định danh đối tượng. Vì vậy các đối tượng *flyweight* có thể được chia sẻ, kiểm tra định danh sẽ trả về *true* cho các đối tượng khác nhau về bản chất.