6.14 Mẫu thiết kế ABSTRACT FACTORY METHOD

6.14.1 Mở đầu

Trong bài trước, chúng ta đã phát triển ứng dụng cho công ty sản xuất các bộ phân tích thông điệp XML và hiển thị kết quả cho họ. Chúng ta làm điều này bằng cách tạo các bộ phân tích khác nhau cho các kiểu gói tin khác nhau giữa công ty và khách hàng của nó. Chúng ta sử dụng mẫu thiết kế Factory Method để giải quyết bài toán của họ.

Ứng dụng làm việc tốt cho họ. Nhưng bây giờ khách hàng không muốn tuân theo các qui tắc đặc tả XML của công ty. Các khách hàng muốn sử dụng các qui tắc XML của riêng họ để trao đổi với công ty sản xuất. Điều đó có nghĩa là với mỗi loại khách hàng, công ty cần phải có bộ phân tích XML chuyên biệt của khách hàng. Chẳng hạn, đối với khách hàng NY cần có bốn kiểu chuyên biệt của bộ phân tích XML, tức là *NYErrorXMLParser, NYFeedbackXMLParser, NYOrderXMLParser, NYResponseXMLParser* và bốn bộ phân tích cho khách hàng TW.

Công ty yêu cầu bạn thay đổi ứng dụng để đáp ứng yêu cầu mới. Để phát triển ứng dụng bộ phân tích chúng ta cần sử dụng mẫu thiết kế *Factory Method* ở đó một đối tượng chính xác để sử dụng được quyết định bởi lớp tuân theo kiểu của bộ phân tích. Bây giờ, để cài đặt yêu cầu mới này, chúng ta sẽ sử dụng nhà máy của các nhà máy (*factory of factories*), tức là *Abstract Factory.*

Bây giờ chúng ta cần các bộ phân tích tùy thuộc XML chuyên biệt của khách hàng, như vậy chúng ta sẽ tạo các *factories* khác nhau cho các khách hàng khác nhau mà sẽ cung cấp cho chúng ta XML chuyên biệt của khách hàng để phân tích. Chúng ta sẽ làm điều đó bằng cách tạo ra *Abstract Factory* và sau đó cài đặt một *factory* để cung cấp *factory XML* chuyên biệt của *client*. Và chúng ta sẽ sử dụng *factory* này để nhận được đối tượng phân tích XML chuyên biệt khách hàng mong muốn.

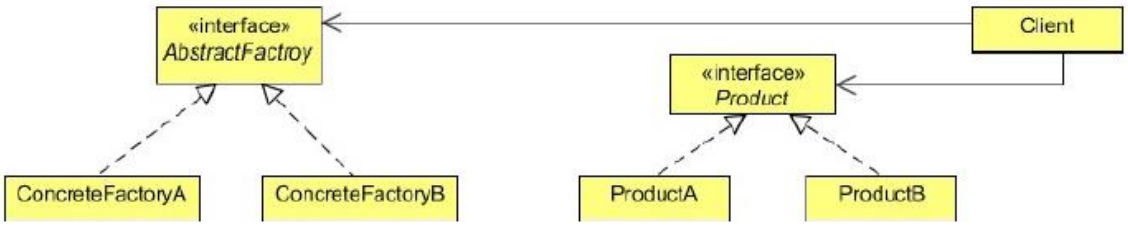
Abtract Factory là mẫu thiết kế lựa chọn của chúng ta và trước khi cài đặt để giải quyết bài toán chúng ta sẽ tìm hiểu thêm về mẫu này.

6.14.2 Mẫu Abstract Factory Method là gì

Abstract Factory là mẫu thiết kế mà cung cấp giao diện để tạo họ (nhóm các nhóm) các đối tượng liên quan hoặc phụ thuộc nhau mà không cần đặc tả các lớp cụ thể của chúng. Mẫu *Abtract Factory* dùng khái niệm mẫu *Factory Method* như mức tiếp theo. *Abstract Factory* là một lớp mà cung cấp giao diện cho họ các đối tượng. Trong Java, nó có thể được cài đặt sử dụng giao diện lớp trừu tượng.

Mẫu *Abstract Factory* là hữu ích khi đối tượng *client* muốn tạo một khởi tạo của một trong một bộ các đối tượng liên quan và phụ thuộc.mà không cần biết lớp cụ thể chuyên biệt nào là được khởi tạo. Các f*actories* cụ thể khác nhau cài đặt giao diện *abstract factory*. Các đối tượng

client sử dụng một trong các *factory* cụ thể để tạo đối tượng và do đó không cần biết lớp cụ thể nào thực tế đã được khởi tạo.
Abstract Factory là hữu ích gắn với một nhóm các đối tượng để theo dõi hành vi của hệ thống. Đối với mỗi nhóm hoặc họ, một *factory* cụ thể được cài đặt để quản trị việc tạo ra các đối tượng và các sự phụ thuộc bên trong và các yêu cầu đồng nhất giữa chúng. Mội *factory* cụ thể cài đặt giao diện của *abstract factory*.



AbstractFactory

- Khai báo giao diện cho các thao tác mà tạo các đối tượng trừu tượng.

ConcreteFactory.

- Cài đặt các thao tác tạo các đối tượng *Product* cụ thể.

AbstractProduct

- Khai báo giao diện cho một kiểu đối tượng *Product*

ConcreteProduct

- Định nghĩa đối tượng product mà được tạo bởi factory cụ thể tương ứng.

Client

- Sử dụng chỉ giao diện được khai báo bởi các lớp *AbstractFactory* và *AbstractProduct*

6.14.3 Cài đặt mẫu Abstract Factory

Để cài đặt mẫu thiết kế Abstract Factory trước hết chúng ta sẽ tạo giao diện mà sẽ được cài đặt bởi *concrete factory.*

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public interface AbstractParserFactory {

        public XMLParser getParserInstance(String parserType);
}
```

Giao diện trên được cài đặt bởi *factory* cụ thể chuyên biệt cho *client* mà sẽ cung cấp đối tượng phân tích cho đối tượng của *client.* Phương thức *getParserInstance* lấy *parserType* làm đối số mà sẽ được sử dụng để nhận đối tượng phân tích chuyên biệt (*error parser, order parser, ...*).

Có hai *factory parser* cụ thể chuyên biệt cho *client* như sau:

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class NYParserFactory implements AbstractParserFactory {

        @Override
        public XMLParser getParserInstance(String parserType) {

                switch(parserType){
                        case "NYERROR": return new NYErrorXMLParser();
                        case "NYFEEDBACK": return new NYFeedbackXMLParser ();
                        case "NYORDER": return new NYOrderXMLParser();
                        case "NYRESPONSE": return new NYResponseXMLParser();
                }

                return null;
        }

}
```

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class TWParserFactory implements AbstractParserFactory {

        @Override
        public XMLParser getParserInstance(String parserType) {

                switch(parserType){
                        case "TWERROR": return new TWErrorXMLParser();
                        case "TWFEEDBACK": return new TWFeedbackXMLParser ();
                        case "TWORDER": return new TWOrderXMLParser();
                        case "TWRESPONSE": return new TWResponseXMLParser();
                }

                return null;
        }

}
```

Hai *factoty* trên cài đặt giao diện *AbstractParserFactory* và ghi đè phương thức *getParserInstance*. Nó trả về đối tượng *parser* chuyên biệt cho client tùy thuộc kiểu *parser* được yêu cầu trong đối số.

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public interface XMLParser {

        public String parse();

}
```

Giao diện trên được cài đặt bởi các lớp *parser* cụ thể để phân tích XML và trả về thông điệp *string.*

Có hai client và bốn kiểu thông điệp khác nhau trao đổi giữa công ty và các khách hàng của nó. Vì vậy, cần có sáu kiểu XML parser cụ thể khác nhau mà chuyên biệt cho client.

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class NYErrorXMLParser implements XMLParser{

        @Override
        public String parse() {
                System.out.println("NY Parsing error XML...");

                return "NY Error XML Message";
        }

}
```

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class NYFeedbackXMLParser implements XMLParser{

        @Override
        public String parse() {
                System.out.println("NY Parsing feedback XML...");
                return "NY Feedback XML Message";
        }

}
```

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class NYOrderXMLParser implements XMLParser{

        @Override
        public String parse() {
                System.out.println("NY Parsing order XML...");
                return "NY Order XML Message";
        }

}
```

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class NYResponseXMLParser implements XMLParser{

        @Override
        public String parse() {
                System.out.println("NY Parsing response XML...");
                return "NY Response XML Message";
        }

}
```

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class TWErrorXMLParser implements XMLParser{

        @Override
        public String parse() {
                System.out.println("TW Parsing error XML...");
                return "TW Error XML Message";
        }

}
```

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class TWFeedbackXMLParser implements XMLParser{

        @Override
        public String parse() {
                System.out.println("TW Parsing feedback XML...");

                return "TW Feedback XML Message";
        }

}
```

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class TWOrderXMLParser implements XMLParser{

        @Override
        public String parse() {
                System.out.println("TW Parsing order XML...");
                return "TW Order XML Message";
        }

}
```

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class TWResponseXMLParser implements XMLParser{

        @Override
        public String parse() {
                System.out.println("TW Parsing response XML...");
                return "TW Response XML Message";
        }

}
```

Để tránh sự phụ thuộc giữa *code client* và *factories*, tùy chọn chúng ta cài đặt *factory producer* mà có một phương thức tĩnh và có trách nhiệm cung cấp đối tượng *factory* mong muốn cho đối tượng *client*.

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public final class ParserFactoryProducer {

        private ParserFactoryProducer(){
                throw new AssertionError();
        }

        public static AbstractParserFactory getFactory(String factoryType){

                switch(factoryType)
                {
                        case "NYFactory": return new NYParserFactory();
                        case "TWFactory": return new TWParserFactory();
                }

                return null;
        }

}
```

Bây giờ chúng ta kiểm tra mẫu này.

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class TestAbstractFactoryPattern {

        public static void main(String[] args) {

                AbstractParserFactory parserFactory = ParserFactoryProducer.getFactory(" ↩
                    NYFactory");
                XMLParser parser = parserFactory.getParserInstance("NYORDER");
                String msg="";
                msg = parser.parse();
                System.out.println(msg);

                System.out.println("***********************************");

                parserFactory = ParserFactoryProducer.getFactory("TWFactory");
                parser = parserFactory.getParserInstance("TWFEEDBACK");
                msg = parser.parse();
                System.out.println(msg);
        }

}
```

Code trên cho kết quả đầu ra là:

```
NY Parsing order XML...
NY Order XML Message
***********************************
TW Parsing feedback XML...
TW Feedback XML Message
```

Trong lớp trên, trước hết chúng ta nhận được *NY factory* từ *factory producer*, và sau đó nhận được *parser Order XML* từ *NY parser factory*. Và chúng ta gọi phương thức *parser* trên đối tượng *parser* và hiển thị thông điệp trả về. Chúng ta làm tương tự đối với khách hàng TW như được chỉ rõ ở đầu ra.

6.14.4 Khi nào sử dụng mẫu thiết kế Abstract Factory

Sử dụng mẫu *Abstract Factory* khi

- Một hệ thống cần là độc lập với việc các đối tượng của nó được tạo, kết hợp và thể hiện như thế nào.
- Một hệ thống cần được cấu hình với một trong nhiều họ các sản phẩm.
- Một họ các đối tượng sản phẩm liên quan được thiết kế để sử dụng cùng nhau và bạn cần đáp ứng ràng buộc này.
- Bạn muốn cung cấp thư viện lớp sản phẩm và bạn muốn chỉ để lộ ra giao diện, chứ không phải cài đặt của chúng.