DP4Dummies – Chương 3: Decorator, Factory

CHƯƠNG 3: TẠO VÀ MỞ RỘNG MỘT ĐỐI TƯỢNG VỚI MẪU DECORATOR VÀ FACTORY

Trong chương này, chúng ta sẽ thảo luận về một số nội dung sau:

- Giữ vững nguyên tắc viết mã "Open-Close" hay "Luôn mở cho việc mở rộng, nhưng đóng cho việc sửa đổi"
- Giới thiệu về mẫu trang trí Decorator
- Các ví dụ về mẫu trang trí Decorator
- Xây dựng các đối tượng với mẫu nhà máy Factory
- Đóng gói việc khởi tạo đối tượng bằng mẫu nhà máy Factory
- Sử dụng các phương thức khởi tạo nhà máy Factory Method

Bạn đang làm nhân viên tư vấn Thiết Kế Mẫu tại công ty GigantoComputer, với mức lương khá cao và bạn đang ở trong căn tin công ty.

"Hôm nay có món gì?" bạn hỏi tay đầu bếp khó chịu đang đứng sau bếp nướng.

"Cho một cái hamburger," bạn nói và xoay xoay cái khay trong tay.

Người đầu bếp mang cái hamburger đến bàn tính tiền, không quên hỏi lại "Có thêm thịt rán không?"

"Chắc chắn rồi", bạn nói.

Người đầu bếp xóa phiếu ăn cũ trên máy tính tiền và làm lại phiếu ăn.

"Hamburger và thịt rán". Vừa nói anh ta vừa gõ vào máy tính tiền.

"Cho thêm một ít pho mát" Bạn nói.

Người đầu bếp ném một ánh nhìn khó chịu , xóa cái phiếu ăn, mổ mổ cái bàn phím và nói "Hamburger với pho mát và thịt nướng. Ok. Đủ rồi chứ?"

“Hmm”, bạn nói, nhìn quét qua cái thực đơn “Hay là thêm một chút thịt xông khói?”

Người đầu bếp nhìn chằm chằm vào bạn và dường như định văng ra một vài câu khó chịu gì đó nhưng vẫn nhập phiếu ăn vào máy.

“Hey”, bạn nói. “Anh chắc chắn là được lợi nhiều hơn từ việc sử dụng mẫu thiết kế trang trí Decorator chứ hả?”

“Vâng”, anh đầu bếp trả lời, tự hỏi về những gì bạn nói “Tôi đã nói điều này cả ngàn lần rồi”

Bạn cầm cái Hamburger pho mát thịt xông khói với vẻ hạnh phúc và nói “Thêm một vài lát cà chua nữa thì tuyệt!”

Chương này nói về hai mẫu thiết kế quan trọng, nó sẽ lắp đầy những thiếu sót trong việc lập trình hướng đối tượng cơ bản, đặc biệt là ở khả năng kế thừa. Đây là hai mẫu trang trí Decorator và mẫu nhà máy Factory.

Mẫu trang trí Decorator là lựa chọn hoàn hảo cho tình huống tôi vừa nêu ở trên bởi vì ta đang nói về khả năng mở rộng chức năng cho một lớp có sẵn. Sau khi viết một lớp, bạn có thể thêm phần trang trí Decorator (các lớp mở rộng) để mở rộng lớp này. Khi đó bạn không phải sửa đổi lên lớp gốc. Kết quả là cái Hamburger của bạn trở thành Hamburger pho mát, rồi Hamburger pho mát thịt xông khói, mọi thứ thật dễ dàng.

Nguyên lý Open-Close – “Luôn Open cho việc mở rộng và Closed cho việc sửa đổi”

Một trong những khía cạnh quan trọng nhất trong quá trình phát triển một ứng dụng là các nhà phát triển và lập trình viên phải đối đầu với sự thay đổi, và đó là lý do vì sao các mẫu thiết kế này lại được giới thiệu trước tiên. Có thể nói các Mẫu Thiết Kế sẽ giúp bạn giải quyết được các sự thay đổi, và bạn có thể dễ dàng chuyển đổi mã nguồn của mình cho các trường hợp mới và bất khả kháng. Như tôi đã nói qua trong suốt cuốn sách này,lập trình viên thường tiêu tốn thời gian cho việc mở rộng và thay đổi mã nguồn hơn là phát triển mã nguồn gốc.

Mẫu chiến lược Strategy đã được giới thiệu trước đây trong chương II, giúp bạn xử lý những sự thay đổi bằng cách cho phép bạn chọn lựa một thuật toán thích hợp từ một tập hợp thuật toán bên ngoài hơn là phải viết lại mã nguồn. Mẫu trang trí Decorator cũng tương tự vậy, nó cho phép bạn viết tiếp mã nguồn, tránh việc sửa đổi lên mã nguồn gốc, trong khi vẫn đáp ứng được yêu cầu thay đổi. Đó là điểm chính yếu tôi muốn nhấn mạnh.

Ghi nhớ:Hãy làm cho mã nguồn của bạn đáp ứng được nguyên tắc "Luôn đóng cho sự chỉnh sửa, và luôn mở cho việc mở rộng" càng nhiều càng tốt. Nói cách khác, hãy thiết kế mã nguồn sao cho không cần phải thay đổi gì nhiều nhưng luôn có thể mở rộng khi cần.

Đây là một ví dụ cho việc viết mã nguồn luôn đóng cho sự thay đổi.

Công ty mà bạn đang làm tư vấn, công ty GigantoComputer, quyết định làm một cái máy vi tính mới. Đây là mã nguồn của lớp *Computer*:

```
public class Computer
{
  public Computer()
  {
  }

  public String description()
  {
    return "You're getting a computer.";
  }
}
```

Khi một đối tượng computer được khởi tạo. Phương thức *description* sẽ trả về văn bản *"You're getting a computer."* . Tới giờ mọi việc vẫn tốt đẹp. Nhưng một số khách hàng quyết định rằng họ muốn có một cái đĩa cứng trong máy tính. "Không vấn đề gì cả" Các lập viên trong công ty đáp. " Chúng ta chỉ cần chỉnh sửa mã nguồn lại một chút như sau:"

```
public class Computer
{
  public Computer()
  {
  }

  public String description()
  {
    return "You're getting a computer and a disk.";
  }
}
```

Bây giờ khi một đối tượng computer được tạo và bạn gọi phương thức *description*, bạn sẽ nhận được văn bản *"You're getting a computer and a disk."* Nhưng một vài khách hàng vẫn chưa hài lòng. Họ muốn thêm một cái màn hình nữa. Và thế là các lập trình viên phải chỉnh sửa tiếp như sau:

```
public class Computer
{
  public Computer()
  {
  }

  public String description()
  {
    return "You're getting a computer and a disk and a monitor.";
  }
}
```

Bây giờ, khi bạn tạo một computer và gọi phương thức *description* bạn sẽ thấy

```
You're getting a computer and a disk and a monitor.
```

Bạn có thể thấy vấn đề ở đây: Các lập trình viên phải thay đổi mã nguồn mỗi khi khách hàng thay đổi yêu cầu của họ. Rõ ràng, đó là vấn đề chính.

Và bạn, với cương vị là tư vấn mẫu thiết kế, sẽ chỉnh sửa nó.

MẪU TRANG TRÍ DECORATOR?

Tôi phải nhắc lại một lần nữa: càng nhiều càng tốt, hãy viết mã nguồn của bạn đóng cho việc sửa đổi, nhưng mở cho việc mở rộng. Trong chương II, bạn đã biết cách làm việc với mẫu chiến lược Strategy. Đó là, bạn đóng gói mã nguồn vào các thuật toán riêng biệt để sử dụng dễ dàng, hơn là việc xử lý chúng thông qua các lớp con.

Mẫu trang trí Decorator có một cách tiếp cận khác. Thay vì sử dụng một thuật toán bên ngoài, mẫu thiết kế này sử dụng một phương pháp "bao bọc" mã nguồn của bạn để mở rộng chúng.

Ghi nhớ: Định nghĩa chính thức của mẫu trang trí Decorator trong sách của GoF có viết: "Gắn kết thêm một số tính năng cho đối tượng một cách linh động. Mẫu trang trí Decorator cung cấp một phương pháp linh hoạt hơn là sử dụng lớp con để mở rộng chức năng cho đối tượng"

Mẫu thiết kế này được gọi là "Người trang trí" Decorator nhưng dường như đó là tên gọi rườm rà. Một cái tên tốt hơn cho mẫu này có thể là "Người tăng thêm" Augmentor hay "Người mở rộng" Extender bởi vì nó cho phép bạn: tăng thêm hay mở rộng một lớp một cách linh động khi chương trình được thực thi. Tuy nhiên, như bạn thấy trong chương này, thuật ngữ "Người trang trí" Decorator còn giúp bạn hiểu rõ hơn khái niệm "đóng cho việc

chỉnh sửa, mở cho việc mở rộng". Khi bạn làm hành động bao bọc mã nguồn để mở rộng thêm chức năng, bạn không cần thiết chỉnh sửa lại mã nguồn cũ, bạn chủ yếu tập trung vào việc trang trí nó.

Và đây là cách mà nó làm việc. Bạn bắt đầu với một cái máy tính computer đơn giản sau:

Figure 3-1: You start with a core computer.

Khi bạn gọi phương thức *description*, bạn nhận được kết quả "*You're getting a computer*". Bây giờ bạn muốn thêm ít phần cứng, một ổ cứng mới chẳng hạn. Trong trường hợp này, bạn có thể thêm một lớp bao bọc wrapper như sau:

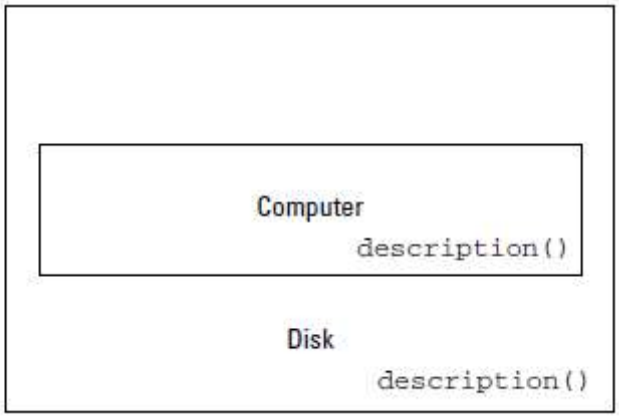

Figure 3-2: Next, you add a disk drive.

Bây giờ khi bạn gọi phương thức *description* của lớp bao bọc wrapper, nó sẽ gọi phương thức *description* của đối tượng computer để nhận được kết quả "*You're getting a computer* " và đối tượng ổ cứng *disk* sẽ trả về kết quả "*and a disk*". Kết quả bạn nhận được *"You're getting a computer and a disk"*

Nếu bạn muốn thêm vài thứ nữa vào lớp máy tính *Computer*, bạn hãy đặt nó vào lớp bao bọc *wrapper*, ví dụ như thêm vào cái màn hình *Monitor*:

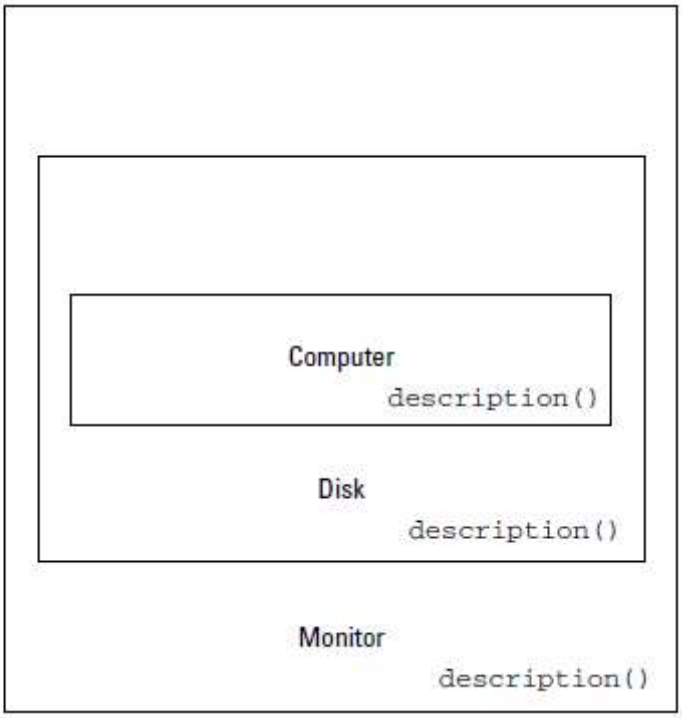

Figure 3-3: Finally, you add a monitor to the computer.

Bây giờ khi bạn gọi phương thức *description*, mọi việc sẽ xảy ra như sau:

– Đối tượng computer, sẽ thực hiện phương thức *description* để tạo ra kết quả *"You're getting a computer"*

– Đối tượng disk, sẽ thực hiện tiếp phương thức trên để thêm vào *"and a disk"*

– Đối tượng monitor, tiếp tục thực hiện phương thức *description* để thêm vào *"and a monitor"*

– Kết quả là bạn nhận được *"You're getting a computer and a disk and a monitor"*

VÍ DỤ VỀ MẪU TRANG TRÍ DECORATOR

Bạn bắt đầu viết một lớp máy tính Computer đơn giản, với một phương thức *description* trả về kết quả "*computer*" như sau:

```
public class Computer
{
  public Computer()
  {
  }

  public String description()
  {
    return "computer";
  }
}
```

OK. Bạn đã hoàn thành cái máy tính đơn giản. Bây giờ làm sao để tạo một lớp trang trí? Những lớp này hoạt động như là một lớp bao bọc cho lớp *Computer*, điều này có nghĩa là phải có một biến để lưu trữ một đối tượng *computer*. Một cách đơn giản để tạo lớp bao bọc wrapper là mở rộng lớp *Computer*.

Tạo dựng một Lớp trang trí Decorator

Bạn có thể bắt đầu bằng việc tạo một lớp trừu tượng được mở rộng từ lớp *Computer* (nhớ rằng lớp trừu tượng sẽ không thể sử dụng trực tiếp được, bạn phải kế thừa từ lớp này, và tạo ra lớp mới để sử dụng). Đây là mã nguồn:

```
public abstract class ComponentDecorator extends Computer
{
  public abstract String description();
}
```

Lớp mới này, *ComponentDecorator* , có một phương thức trừu tượng tên *description*. Bởi vì lớp này là trừu tượng nên bạn không thể tạo đối tượng từ nó. Điều đó có nghĩa là bạn đã chắn chắn mọi lớp bao bọc wrapper kế thừa từ lớp này phải nhất quán, và khi đó mọi lớp kế thừa sẽ có một phương thức description riêng khác nhau.

Thêm vào một đĩa cứng *Disk*

Đây là lớp bao bọc *Disk* , sẽ thêm một ổ cứng vào máy tính. Lớp này sẽ mở rộng từ lớp trừu tượng *ComponentDecorator*

```
public class Disk extends ComponentDecorator
{
   .
   .
   .
}
```

Bởi vì đây là một lớp bao bọc, nó cần phải biết đang bao bọc thứ gì. Vì vậy bạn đưa cho nó một đối tượng *computer* ngay khi nó khởi tạo. Lớp bao bọc *Disk* sẽ lưu trữ một đối tượng tên *computer*

```
public class Disk extends ComponentDecorator
{
  Computer computer;

  public Disk(Computer c)
  {
    computer = c;
  }
   .
   .
   .
}
```

Bây giờ bạn cần hiện thực phương thức Description. (Lưu ý: khi bạn kế thừa một lớp trừu tượng trong Java, bạn cần hiện thức tất cả các phương thức trừu tượng của lớp đó). Phương thức mới này sẽ gọi phương thức *description* của lớp *computer* và thêm vào dòng chữ *"and a disk"* như sau:

```
public class Disk extends ComponentDecorator
{
  Computer computer;

  public Disk(Computer c)
  {
    computer = c;
  }

  public String description()
  {
    return computer.description() + " and a disk";
  }
}
```

Vậy là bạn đã bao bọc đối tượng *computer,* và khi bạn gọi phương thức *description* của đối tượng *disk* này, nó sẽ gọi phương thức *description* của lớp computer, đồng thời thêm vào dòng chữ *"and a disk"*. Kết quả bạn sẽ có *"computer and a disk"*

Thêm vào một ổ CD

Bạn cũng có thể thêm vào một ổ CD theo cùng cách trên. Đây là mã nguồn

```
public class CD extends ComponentDecorator
{
  Computer computer;

  public CD(Computer c)
  {
    computer = c;
  }

  public String description()
  {
    return computer.description() + " and a CD";
  }
}
```

Thêm vào một màn hình monitor

Tất nhiên bạn cũng có thêm vào một màn hình theo cùng một cách như sau:

```
public class Monitor extends ComponentDecorator
{
  Computer computer;

  public Monitor(Computer c)
  {
    computer = c;
  }

  public String description()
  {
    return computer.description() + " and a monitor";
  }
}
```

OK. Bạn đã có đầy đủ các lớp. Giờ là lúc chạy thử nghiệm chương trình.

Đầu tiên bạn tạo đối tượng c*omputer* như sau:

```
public class Test
{
  public static void main(String args[])
  {
    Computer computer = new Computer();
        .
        .
        .
  }
}
```

Sau đó bạn bao bọc đối tượng *computer* để thêm vào một đĩa cứng

```
public class Test
{
  public static void main(String args[])
  {
    Computer computer = new Computer();

    computer = new Disk(computer);
        .
        .
        .
  }
}
```

Bây giờ hãy thêm vào một monitor:

```
public class Test
{
  public static void main(String args[])
  {
    Computer computer = new Computer();

    computer = new Disk(computer);
    computer = new Monitor(computer);
        .
        .
        .
  }
}
```

Sau đó, bạn có thêm vào không chỉ một ổ CD, mà là hai ổ CD chẳng hạn. Không vấn đề gì lớn lao cả. Cuối cùng bạn gọi phương thức *description* của lớp bao bọc để xem kết quả:

```
public class Test
{
  public static void main(String args[])
  {
    Computer computer = new Computer();

    computer = new Disk(computer);
    computer = new Monitor(computer);
    computer = new CD(computer);
    computer = new CD(computer);

    System.out.println("You're getting a " + computer.description()
      + ".");
  }
}
```

OK. Khi chạy chương trình bạn nhận được kết quả.

```
You're getting a computer and a disk and a monitor and a CD and a CD.
```

Không tồi. Bạn đã mở rộng một đối tượng gốc thật đơn giản bằng cách bao bọc nó trong nhiều lớp trang trí decorator khác nhau, tránh việc phải chỉnh sửa trong mã nguồn gốc. Và đó là Mẫu Thiết Kế Trang Trí Decorator.

CẢI TIẾN TOÁN TỬ NEW VỚI MẪU THIẾT KẾ NHÀ MÁY FACTORY

Tại đây, công ty MegaGigaCo, bạn được trả lương cao cho kỹ năng thiết kế mẫu chuyên nghiệp của mình, bạn đang tạo một đối tượng kết nối cơ sở dữ liệu mới. Hãy xem toán tử new trong Java làm việc này như thế nào?

```
Connection connection = new OracleConnection();
```

“Không tồi,” bạn nghĩ, sau khi hoàn thành đoạn mã việc lớp OracleConnection. Bây giờ bạn đã có thể kết nối với cơ sở dữ liệu Oracle.

“Nhưng,” Giám đốc điều hành la lên ,”làm thế nào để kết với máy chủ cơ sở dữ liệu Microsof SQL Server?”

“Được”, bạn nói “Bình tĩnh, để tôi suy nghĩ một lát”. Bạn ra khỏi phòng để ăn trưa và sau đó quay lại tìm giám đốc và ban quản trị. Mọi người nóng lòng chờ đợi và hỏi “Mọi việc đã xong chưa?”

Bạn trở lại làm việc và tạo ra một lớp mới dùng để kết nối cơ sở dữ liệu, lớp SQLServerConnection.

```
Connection connection = new SqlServerConnection();
```

“Tốt lắm” Vị giám đốc nói. “Umh, vậy làm sao để kết nối với MySQL? Chúng ta muốn nó là kết nối mặc định”. “Woa”, bạn hơi bối rối. Tuy nhiên bạn vẫn làm thêm một kết nối với MySQL như sau:

```
Connection connection = new MySqlConnection();
```

Hiện tại bạn đã có ba loại kết nối cơ sở dữ liệu như sau: Oracle, SQL Server và MySQL. Vì vậy bạn chỉnh sửa mã nguồn cho phù hợp với các biến như “Oracle”, “SQL Server” hay bất cứ biến nào khác như sau:

```
Connection connection;

if (type.equals("Oracle")){
  connection= new OracleConnection();
}
else if (type.equals("SQL Server")){
  connection = new SqlServerConnection();
}
else {
  connection = new MySqlConnection();
}
```

Mọi việc đều ổn, bạn nghĩ. Tuy nhiên có tới 200 chỗ trong mã nguồn cần phải tạo kết nối cơ sở dữ liệu. Vì vậy đã tới lúc đưa đoạn mã này vào một phương thức riêng, phương thức createConnection, qua truyền cho nó loại kết nối mà bạn muốn như sau:

```
public Connection createConnection(String type)
{
  .
  .
  .
}
```

Phương thức có thể trả về loại kết nối mong muốn, tùy thuộc vào giá trị tham số truyền vào:

```
public Connection createConnection(String type)
{
  if (type.equals("Oracle")){
    return new OracleConnection();
  }
  else if (type.equals("SQL Server")){
    return new SqlServerConnection();
  }
  else {
    return new MySqlConnection();
  }
}
```

Tuyệt, bạn nghĩ. Không có gì khó khăn ở đây.

“Tin xấu” Vị giám đốc nói lớn trong khi chạy ào vào phòng làm việc của bạn. “Chúng ta cần phải chỉnh sửa lại mã nguồn để xử lý các kết nối an toàn cho tất cả máy chủ cơ sở dữ liệu. Hội đồng quản trị của khu vực Western yêu cầu như vậy”

Bạn đưa vị giám đốc ra khỏi phòng và ngồi suy nghĩ. Tất cả mã nguồn chắc phải chỉnh sửa lại. Phương thức mới createConnection, phần chính của mã nguồn, sẽ phải chỉnh sửa lại.

Trong chương II của quyển sách này. Bạn đã được biết dấu hiệu phải sử dụng mẫu thiết kế: "Đó là tách rời phần mã nguồn dễ thay đổi nhất ra khỏi phần mã chính của bạn. Và cố gắng sử dụng lại những phần này càng nhiều càng tốt."

Có lẽ đây là lúc nghĩ về việc tách rời phần mã nguồn dễ thay đổi ra khỏi chương trình chính, phần tạo kết nối cơ sở dữ liệu connection, và đóng gói nó vào một đối tượng. Và đối tượng đó chính là mẫu nhà máy Factory. Đối tượng là một nhà máy, được viết trong mã nguồn, nhằm tạo ra các đối tượng kết nối connection.

Vì sao bạn nghĩ tới mẫu thiết kế nhà máy Factory. Đây là những gợi ý:

- Bạn sử dụng toán tử new để tạo đối tượng OracleConnection
- Sau đó lại sử dụng tiếp toán tử new để tạo đối tượng SQLServerConnection, và sau đó là MySQLConnection. Nói cách khác, bạn đã sử dụng toán tử new để tạo nhiều đối tượng thuộc các lớp khác nhau, điều này làm mã nguồn của bạn trở nên lớn hơn và bạn buộc phải lặp lại điều này nhiều lần trong toàn bộ mã nguồn.
- Sau đó bạn đưa đoạn mã đó vào trong một phương thức
- Bởi vì yêu cầu vẫn còn có thể thay đổi nhanh chóng, nên cách tốt nhất là đóng gói chúng vào một đối tượng nhà máy factory. Theo cách làm này, bạn đã tách phần mã dễ thay đổi riêng biệt ra và giúp phần mã nguồn còn lại giữ vững nguyên tắc "đóng cho việc sửa đổi"

Chúng ta có thể nói rằng, toán tử new vẫn tốt trong mọi trường hợp, nhưng khi mã tạo dựng đối tượng bị liên tục thay đổi, ta nên nghĩ đến việc đóng gói chúng bằng mẫu thiết kế nhà máy factory.

XÂY DỰNG MẪU NHÀ MÁY FACTORY ĐẦU TIÊN

Nhiều lập trình viên biết cách thức mà đối tượng nhà máy factory làm việc. Họ nghĩ đơn giản rằng, bạn có một đối tượng làm nhiệm vụ tạo ra đối tượng khác. Đó là cách mà đối tượng factory thường được tạo ra và sử dụng, tuy nhiên nó còn làm được nhiều hơn thế. Chúng ta hãy nhìn vào cách thông thường khi tạo một đối tượng nhà máy factory trước, sau đó xem xét định nghĩa chính xác từ sách của GOF, theo định nghĩa mà đối tượng nhà máy factory sẽ có nhiều điểm khác, nhiều sự uyển chuyển hơn.

Tạo dựng đối tượng nhà máy Factory

Ví dụ đầu tiên, FirstFactory, sẽ làm việc theo cách hiểu thông thường nhất. Lớp FirstFactory đóng gói đối tượng xây dựng connection, và bạn truyền giá trị tham số theo đúng loại muốn

tạo đó là "Oracle" hay "SQL Server" hay loại gì khác. Đây là cách bạn tạo một đối tượng sử dụng nhà máy factory :

```
FirstFactory factory;

factory = new FirstFactory("Oracle");
```

Bây giờ, bạn có thể sử dụng đối tượng nhà máy factory mới tạo này, để tạo đối tượng kết nối connection, bằng cách gọi phương thức tên createConnection như sau:

```
FirstFactory factory;

factory = new FirstFactory("Oracle");

Connection connection = factory.createConnection();
```

Vậy bạn đã tạo lớp nhà máy FirstFactory như thế nào? Hãy xem mã sau:

```
public class FirstFactory
{
  protected String type;

  public FirstFactory(String t)
  {
    type = t;
  }
    .
    .
    .
}
```

Đầu tiên bạn truyền kiểu kết nối vào phương thức khởi tạo của lớp FirstFactory.

Lớp FirstFactory chứa đựng một phương thức createConnection dùng để tạo ra một đối tượng kết nối connection thật sự. Đây là nơi bạn phải chỉnh sửa mã nguồn nhiều nhất tùy theo loại kết nối muốn tạo, mã như sau:

```
public class FirstFactory
{
  protected String type;

  public FirstFactory(String t)
  {
    type = t;
  }

  public Connection createConnection()
  {
    if (type.equals("Oracle")){
      return new OracleConnection();
    }
    else if (type.equals("SQL Server")){
      return new SqlServerConnection();
    }
    else {
      return new MySqlConnection();
    }
  }
}
```

Kết quả, bạn đã có một lớp nhà máy factory.

Tạo một lớp kết nối Connection trừu tượng

Hãy nhớ rằng một trong những mục tiêu của chúng ta khi viết mã, là làm sao việc thay đổi phần chính của mã nguồn càng ít càng tốt. Với mục tiêu đó, hãy nhìn đoạn mã sau làm việc, khi ta sử dụng một đối tượng connection được tạo bởi đối tượng nhà máy factory:

```
FirstFactory factory;

factory = new FirstFactory("Oracle");

Connection connection = factory.createConnection();

connection.setParams("username", "Steve");

connection.setParams("password", "Open the door!!!)";

connection.initialize();

connection.open();
        .
        .
```

Bạn có thể thấy rằng, đối tượng kết nối connection được tạo bởi nhà máy factory, được sử dụng khắp nơi trong mã nguồn. Để sử dụng cùng một đoạn mã cho tất cả các loại kết nối khác nhau (Oracle,MySQL...), đoạn mã cần phải được viết theo tính "đa hình", có nghĩa là tất cả các đối tượng connection, đều có cùng một giao diện interface, hay cùng kế thừa từ

một lớp cơ sở. Theo cách đó, bạn có thể sử dụng cùng một biến cho mọi loại đối tượng kết nối.

Trong ví dụ, tôi tạo một lớp trừu tượng connection, để các lớp khác kế thừa nó. Lớp này gồm một phương thức khởi dựng, và một phương thức description (trả về mô tả của loại đối tượng). Mã như sau:

```
public abstract class Connection
{
  public Connection()
  {
  }

  public String description()
  {
    return "Generic";
  }
}
```

OK. Mọi việc có vẻ tốt đẹp. Bây giờ bạn đã tạo một lớp trừu tượng cơ sở cho các lớp kết nối khác kế thừa. Bạn cần phải kế thừa tất cả các đối tượng kết nối connection tạo ra từ lớp nhà máy factory.

Tạo lớp kế nối connection

Có ba lớp kết nối connection mà nhà máy Factory có thể tạo ra, phù hợp với loại kết nối mà Vị giám đốc mong muốn: OracleConnection,SqlServerConnection,MySQLConnection. Như chúng ta vừa nói, cần phải kế thừa từ lớp trừu tượng vừa tạo. Và mỗi loại trong chúng đều có phương thức decription trả về mô tả của từng loại kết nối một. Đây là mã nguồn của lớp OracleConnection:

```
public class OracleConnection extends Connection
{
  public OracleConnection()
  {
  }

  public String description()
  {
    return "Oracle";
  }
}
```

Đây là lớp SqlServerConnection, cũng kế thừa từ lớp trừu tượng Connection:

```
public class SqlServerConnection extends Connection
{
  public SqlServerConnection()
  {
  }

  public String description()
  {
    return "SQL Server";
  }
}
```

Và lớp MySqlConnection cũng tương tự:

```
public class MySqlConnection extends Connection
{
  public MySqlConnection()
  {
  }

  public String description()
  {
    return "MySQL";
  }
}
```

Tuyệt với. Mọi việc hoàn tất. Giờ là lúc thử nghiệm chúng. Đầu tiên ta tạo lớp nhà máy, truyền tham số khởi dựng là Oracle:

```
public class TestConnection
{
  public static void main(String args[])
  {
    FirstFactory factory;

    factory = new FirstFactory("Oracle");

    Connection connection = factory.createConnection();
        .
        .
        .
  }
}
```

Để kiểm tra lại đối tượng connection được tạo có phải là Oracle không, ta gọi phương thức description như sau:

```
public class TestConnection
{
  public static void main(String args[])
  {
    FirstFactory factory;

    factory = new FirstFactory("Oracle");

    Connection connection = factory.createConnection();

    System.out.println("You're connecting with " +
      connection.description());
  }
}
```

Kết quả bạn nhận được

```
You're connecting with Oracle
```

Không tồi. Đó là những gì bạn mong đợi.

Ghi nhớ: Theo sách GoF, mẫu thiết kế phương thức nhà máy Factory Method được định nghĩa "Định nghĩa một giao diện để tạo một đối tượng, nhưng cho phép các lớp con quyết định cách thức thể hiện nó. Phương thức nhà máy Factory cho phép một lớp trì hoãn việc hiện thực của nó qua các lớp con"

Điểm mấu chốt ở đây là phần "để lớp con quyết định". Cho tới bây giờ, lớp nhà máy Factory mà bạn vừa tạo, vẫn chưa cho phép các lớp con quyết định cách thể hiện, trừ việc cho kế thừa và ghi đè lại phương thức của lớp Connection cơ sở.

Mẫu thiết kế phương thức nhà máy Factory Method của GoF đem đến cho bạn khả năng uyển chuyển hơn phương pháp truyền thống rất nhiều. Cách làm của GoF là: bạn định nghĩa cách phương thức nhà máy Factory làm việc, và cho phép các lớp con hiện thực implement một nhà máy factory thật sự.

Chúng ta đã nói rằng, Hội đồng quản trị khu vực Western bất ngờ gọi điện và yêu cầu họ không thích lớp nhà máy FirstFactory, họ muốn có thể tạo ra các kết nối bảo mật đến máy chủ cơ sở dữ liệu, không chỉ là một kết nối thông thường. Điều này có nghĩa là họ phải viết lại lớp nhà máy FirstFactory mỗi khi bạn thay đổi nó, để họ có thể tạo ra một kết nối bảo mật.

Đây là vấn đề của các lập trình viên. Mỗi khi bạn cập nhật lại lớp FirstFactory, các lập trình viên khác phải viết lại mã của họ để thích hợp với yêu cầu của họ. Họ đang gọi và yêu cầu

rằng họ muốn kiểm soát được quá trình nhiều hơn.

Tốt thôi, bạn nói. Đó chính là vấn đề mẫu thiết kế Factory áp dụng, giao quyền kiểm soát cho các lớp con. Để thấy cách mẫu này hoạt động, bạn thay đổi cách tạo đối tượng kết nối connection, sử dụng kỹ thuật của GoF, bạn sẽ làm cho khu vực Western của công ty MegaGigaCo hài lòng.

Gợi ý: Bạn vẫn còn băn khoăn về cách sử dụng của mẫu nhà máy Factory của GoF? Mẫu Factory được sử dụng khi bạn muốn chuyển giao toàn bộ quyền điều khiển các lớp con cho các lập trình viên khác.

TẠO MỘT NHÀ MÁY FACTORY THEO CÁCH CỦA GoF

Làm cách nào để "cho phép các lớp con toàn quyền hiện thực cách lớp con thể hiện" khi tạo một đối tượng nhà máy factory. Cách mà bạn phải làm là định nghĩa lớp nhà máy factory như là một lớp trừu tượng abstract hay giao diện interface, và để cho các lớp con hiện thực implement nó.

Nói cách khác, bạn tạo ra một khung sườn cho lớp nhà máy Factory tại trụ sở của MegaGigaCo, và việc hiện thực lớp này sẽ do các lớp con đảm nhiệm.

Tạo lớp nhà máy trừu tượng factory

Việc tạo lớp trừu tượng factory rất dễ dàng. Lớp này được gọi là ConnectionFactory

```
public abstract class ConnectionFactory
{
        .
        .
        .
}
```

Bên cạnh một phương thức khởi dựng rỗng, phương thức quan trọng nhất ở đây là phương thức nhà máy createConnection. Ta phải làm cho phương thức mang tính trừu tượng, để các lớp con hiện thực nó. Phương thức này nhận một đối số, đó là loại kết nối cần tạo:

```
public abstract class ConnectionFactory
{
  public ConnectionFactory()
  {
  }

  protected abstract Connection createConnection(String type);
}
```

Và đó là tất cả những gì bạn cần. Sự đặc tả cho đối tượng nhà máy factory. Bây giờ khu vựa Western sẽ hài lòng vì họ có thể hiện thực một đối tượng nhà máy cụ thể thích hợp với họ từ lớp trừu tượng trên.

Tạo một lớp nhà máy factory cụ thể

Bạn đã bay tới khu vực Western của công ty MegaGigaCo, để giúp họ xử lý vấn đề tạo đối tượng. Bạn giải thích "Tôi hiểu rằng các anh muốn được quyền điều khiển nhiều hơn đối với các đối tượng kết nối"

"Vâng" các lập trình viên của Western nói. "Chúng tôi muốn có thể làm việc với các kết nối bảo mật. Chúng tôi đã tạo một vài lớp mới, lớp SecureOracleConnection, SecureSqlServerConnection và SecureMySqlConnection để tạo ra các kết nối bảo mật.

"OK" bạn nói. "tất cả những gì các bạn phải làm là mở rộng lớp trừu tượng mới của tôi, tên là ConnectionFactory khi các bạn muốn tạo đối tượng nhà máy factory cho các bạn. Hãy chắc chắn là các bạn sẽ hiện thực phương thức createConnection. Sau đó bạn có thể tùy ý viết mã cho phương thức createConnection để tạo đối tượng theo đúng cách bảo mật mà bạn muốn"

Các lập trình viên của Western nói. "Wa, thật dễ dàng. Chúng tôi sẽ tạo lớp factory mới với tên SecureFactory, và nó sẽ kế thừa từ ConnectionFactory như sau:

```
public class SecureFactory extends ConnectionFactory
{
      .
      .
      .
}
```

"Tiếp theo," các lập viên của khu vực Western nói "Chúng chỉ cần hiện thực lớp createConnection mà lớp trừu tượng ConnectionFactory yêu cầu:

```
public class SecureFactory extends ConnectionFactory
{
  public Connection createConnection(String type)
  {
      .
      .
      .
  }
}
```

“Cuối cùng,” các lập trình viên nói “chúng ta chỉ cần tạo các đối tượng từ các lớp vừa tạo, lớp SecureOracleConnection, SecureSqlServerConnection và SecureMySqlConnection, tùy thuộc vào kiểu dữ liệu được truyền vào hàm createConnection:

```
public class SecureFactory extends ConnectionFactory
{
  public Connection createConnection(String type)
  {
    if (type.equals("Oracle")){
      return new SecureOracleConnection();
    }
    else if (type.equals("SQL Server")){
      return new SecureSqlServerConnection();
    }
    else {
      return new SecureMySqlConnection();
    }
  }
}
```

“Thật đơn giản” Họ nói.

Sự khác biệt giữa cách tạo mẫu nhà máy factory thông thường và cách của GoF là cách của GoF chỉ đặc tả lớp nhà máy factory và để cho các lớp con xử lý nội dung chi tiết.

Tạo các lớp kết nối bảo mật

Để hiểu rõ cách thức GoF tạo mẫu factory, bạn cần tạo các lớp cụ thể cho đối tượng nhà máy connect mới, lớp SecureOracleConnection, SecureSqlServerConnection và lớp SecureMySqlConnection. Thật dễ dàng để tạo chúng ,bắt đầu từ lớp SecureOracleConnection, với hàm description trả về văn bản “Oracle Secure”:

```
public class SecureOracleConnection extends Connection
{
  public SecureOracleConnection()
  {
  }

  public String description()
  {
    return "Oracle secure";
  }
}
```

Tiếp theo là lớp SecureSqlServerConnection, với hàm description trả về văn bản "SQL Server Secure"

```
public class SecureSqlServerConnection extends Connection
{
  public SecureSqlServerConnection()
  {
  }

  public String description()
  {
    return "SQL Server secure";
  }
}
```

Và lớp SecureMySqlConnection, với hàm description trả về văn bản "MySql Secure":

```
public class SecureMySqlConnection extends Connection
{
  public SecureMySqlConnection()
  {
  }

  public String description()
  {
    return "MySQL secure";
  }
}
```

Vậy là đã hoàn tất phần mã nguồn. Giờ là lúc cho chương trình chạy.

Thực thi chương trình

Để kiểm tra mã nguồn, hãy tạo đối tượng SecureFactory và sử dụng nó để tạo đối tượng SecureOracleConnection. Mã như sau:

```
public class TestFactory
{
  public static void main(String args[])
  {
    SecureFactory factory;

    factory = new SecureFactory();
```

Tất cả những gì bạn cần phải làm là sử dụng hàm createConnection của đối tượng nhà máy factory để tạo các kết nối an toàn. Mã như sau:

```
public class TestFactory
{
  public static void main(String args[])
  {
    SecureFactory factory;

    factory = new SecureFactory();

    Connection connection = factory.createConnection("Oracle");

    System.out.println("You're connecting with " +
      connection.description());
  }
}
```

Khi chạy chương trình, đúng như mong đợi, bạn nhận được văn bản sau, chứng tỏ rằng bạn đang sử dụng một kết nối Oracle bảo mật:

```
You're connecting with Oracle secure
```

Đây cũng là kết quả mà bạn nhận được từ ví dụ FirstFactory mà chúng ta đã nói trong phần trước, ngoại trừ một điều là bạn cho phép khu vựa Western tự mình hiện thực loại nhà máy factory mà họ mong muốn. Bạn đặc tả một lớp nhà máy bằng cách tạo ra một lớp trừu tượng hay một giao diện interface để các lớp con sử dụng, và người khác sẽ tự mình quyết định lớp đó thực hiện như thế nào. Không còn việc sử dụng một đối tượng nhà máy cụ thể, nay tập hợp các lớp con quyết định việc thể hiện chúng như thế nào.

Download source code C# tại đây

Design Patterns For Dummies

Design Patterns

< DP4Dummies – Chương 2: Strategy

> DP4Dummies – Chương 4: Observer, Chain of Responsibility