Trong bài viết này, chúng tôi sẽ tiếp tục thảo luận của chúng tôi về các mẫu thiết kế bằng cách nhìn vào các **mẫu thiết kế bộ chuyển đổi** . Khuôn mẫu đặc biệt này có thể được sử dụng khi mã của bạn phụ thuộc vào một số API bên ngoài, hoặc bất kỳ lớp khác đó là dễ bị thay đổi thường xuyên. Mô hình này thuộc loại "mô hình cấu trúc" bởi vì nó dạy cho chúng ta cách mã của chúng tôi và các lớp học của chúng tôi cần được cấu trúc để quản lý và / hoặc mở rộng cho họ một cách dễ dàng.

Một lần nữa, tôi muốn nhắc lại rằng các mẫu thiết kế không có gì mới so với các lớp học truyền thống. Thay vào đó, họ chỉ cho chúng ta một cách tốt hơn để cơ cấu các lớp học của chúng tôi, xử lý các hành vi của họ, và quản lý sáng tạo của họ.

**Vấn đề**

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15 | <?php  class PayPal {        public function \_\_construct() {          // Your Code here //      }        public function sendPayment($amount) {          // Paying via Paypal //          echo "Paying via PayPal: ". $amount;      }  }    $paypal = new PayPal();  $paypal->sendPayment('2629'); |

Trong đoạn mã trên, bạn có thể thấy rằng chúng tôi đang sử dụng một lớp PayPal chỉ đơn giản là trả số tiền. Ở đây, chúng tôi đang trực tiếp tạo ra các đối tượng của lớp PayPal và trả tiền qua PayPal. Bạn có mã này nằm rải rác ở nhiều nơi. Vì vậy, chúng ta có thể thấy rằng các mã được sử dụng các  $paypal->sendPayment('amount here'); phương pháp để trả.

Một số thời gian trước đây, PayPal đã thay đổi tên method API từ sendPaymentđến payAmount. Điều này sẽ cho thấy rõ ràng là một vấn đề đối với những người trong chúng ta đã từng sử dụng các sendPaymentphương pháp. Cụ thể, chúng ta cần phải thay đổi tất cả các  sendPayment phương pháp các cuộc gọi đến payAmount. Hãy tưởng tượng số lượng mã chúng ta cần phải thay đổi và thời điểm chúng tôi cần phải chi tiêu vào các thử nghiệm từng tính năng một lần nữa.

**Giải pháp**

Một giải pháp cho vấn đề này là sử dụng các mẫu thiết kế bộ chuyển đổi.

Theo [Wikipedia](http://en.wikipedia.org/wiki/Adapter_pattern) :

*Trong kỹ thuật phần mềm, mô hình bộ chuyển đổi là một mẫu thiết kế phần mềm cho phép giao diện của một lớp học hiện có để được sử dụng từ giao diện khác. Nó thường được dùng để làm các lớp học hiện làm việc với những người khác mà không cần thay đổi mã nguồn của họ.*

Trong trường hợp này, chúng ta nên tạo một giao diện wrapper mà làm cho điều này có thể. Chúng tôi sẽ không thực hiện bất kỳ thay đổi trong thư viện lớp bên ngoài vì chúng ta không có quyền kiểm soát trên nó và nó có thể thay đổi bất cứ lúc nào.

Hãy đào vào mã bây giờ, trong đó cho thấy các mô hình chuyển đổi trong hành động:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | // Concrete Implementation of PayPal Class  class PayPal {        public function \_\_construct() {          // Your Code here //      }        public function sendPayment($amount) {          // Paying via Paypal //          echo "Paying via PayPal: ". $amount;      }  }    // Simple Interface for each Adapter we create  interface paymentAdapter {      public function pay($amount);  }    class paypalAdapter implements paymentAdapter {        private $paypal;        public function \_\_construct(PayPal $paypal) {          $this->paypal = $paypal;      }        public function pay($amount) {          $this->paypal->sendPayment($amount);      }  } |

Nghiên cứu code trên và bạn sẽ có thể nói rằng chúng tôi đã không giới thiệu bất kỳ thay đổi vào chính PayPallớp. Thay vào đó, chúng tôi đã tạo ra một giao diện cho bộ chuyển đổi thanh toán của chúng tôi và một lớp adapter cho PayPal.

Và do đó, sau đó chúng tôi đã thực hiện các đối tượng của lớp chuyển đổi thay vì chính PayPallớp. Trong khi tạo một đối tượng của lớp học chuyển đổi, chúng tôi sẽ vượt qua các đối tượng của chính PayPallớp học như một tham số, do đó lớp adapter có thể có một tham chiếu đến các lớp học chính và nó có thể gọi các phương pháp cần thiết của chính PayPallớp.

Hãy tìm hiểu làm thế nào chúng ta có thể sử dụng phương pháp này trực tiếp:

|  |  |
| --- | --- |
| 1  2  3 | // Client Code  $paypal = new paypalAdapter(new PayPal());  $paypal->pay('2629'); |

Bây giờ tưởng tượng PayPal thay đổi tên phương thức của nó từ sendPaymentđể payAmount. Sau đó, chúng ta chỉ cần thực hiện thay đổi trong paypalAdapter. Chỉ cần có một cái nhìn tại các mã bộ chuyển đổi sửa đổi, trong đó chỉ có một sự thay đổi.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12 | class paypalAdapter implements paymentAdapter {        private $paypal;        public function \_\_construct(PayPal $paypal) {          $this->paypal = $paypal;      }        public function pay($amount) {          $this->paypal->payAmount($amount);      }  } |

Vì vậy, chỉ cần một sự thay đổi và chúng tôi đang có.

**Thêm một Adapter mới**

Tại thời điểm này, chúng tôi đã nhìn thấy như thế nào chúng ta có thể sử dụng các thiết kế guốc adapter để vượt qua những tình huống nói trên. Bây giờ, nó rất dễ dàng để thêm một lớp mới phụ thuộc vào các bộ chuyển đổi hiện tại.Hãy nói rằng các API moneybooker là có cho thanh toán.

Sau đó, thay vì sử dụng các lớp moneybooker trực tiếp, chúng ta nên áp dụng các mô hình chuyển đổi cùng chúng tôi chỉ sử dụng cho PayPal.

|  |  |
| --- | --- |
|  | // Concrete Implementation of MoneyBooker Class  class MoneyBooker {        public function \_\_construct() {          // Your Code here //      }        public function doPayment($amount) {          // Paying via MoneyBooker //          echo "Paying via MoneyBooker: ".  $amount;      }  }    // MoneyBooker Adapter  class moneybookerAdapter implements paymentAdapter {        private $moneybooker;        public function \_\_construct(MoneyBooker $moneybooker) {          $this->moneybooker = $moneybooker;      }        public function pay($amount) {          $this->moneybooker->doPayment($amount);      }  }    // Client Code  $moneybooker = new moneybookerAdapter(new MoneyBooker());  $moneybooker->pay('2629'); |

Như bạn có thể thấy, các nguyên tắc tương tự áp dụng. Bạn định nghĩa một phương pháp nào có sẵn cho các lớp học của bên thứ ba và sau đó, nếu một phụ thuộc thay đổi API của nó, bạn chỉ cần thay đổi các lớp phụ thuộc mà không lộ giao diện bên ngoài của nó.

**Phần kết luận**

Một ứng dụng tuyệt vời được liên tục nối vào các thư viện và các API khác, vì vậy tôi sẽ đề xuất rằng chúng tôi thực hiện phương pháp chuyển đổi, do đó chúng tôi không gặp bất kỳ rắc rối khi một API của bên thứ ba hoặc thư viện thay đổi cơ sở mã của nó.