Chương 7: Adapter Pattern và Facade Pattern – Trở nên thích nghi (P1)

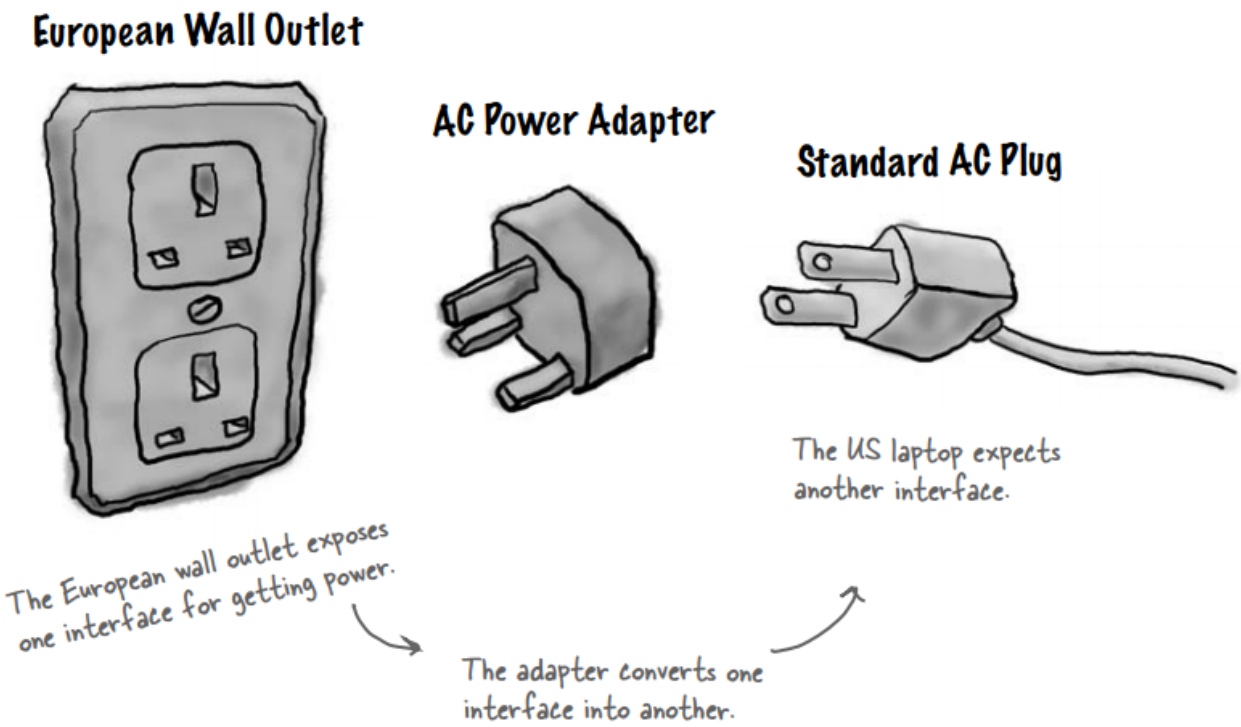
Adapter Pattern và Facade Pattern (https://toihocdesignpattern.com/chuong-7-adapter-va-facade-pattern-tro-nen-thich-nghi-phan-2.html) – trở nên thích nghi

Trong chương này, chúng tôi sẽ cố gắng thực hiện những chiến-công-bất-khả-thi như ghép một cái chốt vuông vào một cái lỗ tròn. Hoàn toàn có thể khi chúng ta có các mẫu thiết kế. Bạn nhớ Decorator Pattern (https://toihocdesignpattern.com/chuong-3-head-first-design-patterns-tieng-viet-decorator-pattern-doi-tuong-trang-tri.html) chứ? Chúng tôi bọc các đối tượng để cung cấp cho chúng trách nhiệm mới. Bây giờ, chúng tôi sẽ bọc một số đối tượng với một mục đích khác: để làm cho giao diện của chúng trở nên khác đi. Tại sao cần làm điều đó? Khi đó, chúng ta có thể điều chỉnh một thiết kế mong đợi một interface thành một lớp implements một interface khác (**Adapter Pattern**). Đó chưa phải là tất cả; Trong khi chúng tôi

làm việc với nó, chúng tôi sẽ xem xét một mẫu khác bao bọc các đối tượng để đơn giản hóa interface của đối tượng đó (Facade Pattern (https://toihocdesignpattern.com/chuong-7-adapter-va-facade-pattern-tro-nen-thich-nghi-phan-2.html)).

Adapter Pattern xung quanh chúng ta

Bạn không gặp khó khăn để hiểu OO adapter là gì, vì thế giới thực có rất nhiều. Cho một ví dụ: Bạn đã bao giờ cần sử dụng máy tính xách tay do Mỹ sản xuất ở một quốc gia châu Âu chưa? Khi đó, bạn có thể cần một bộ chuyển đổi nguồn AC…



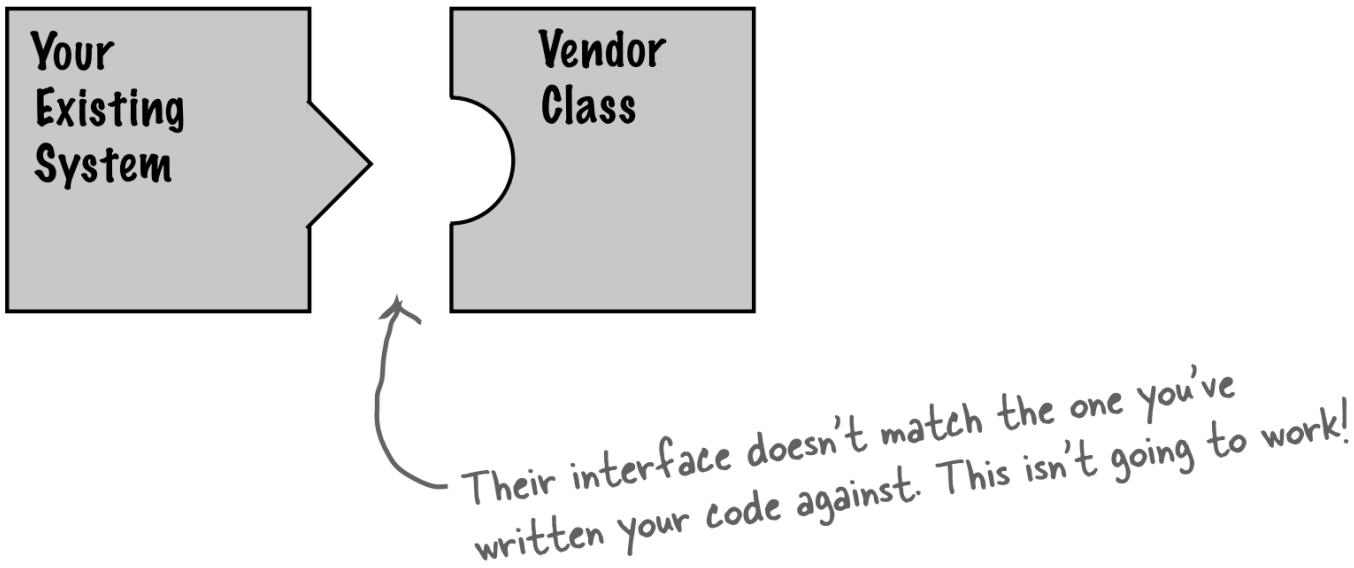
Bạn biết adapter làm gì: nó nằm ở giữa phích cắm của máy tính và ổ cắm AC châu Âu; công việc của nó là điều chỉnh ổ cắm châu Âu để bạn có thể cắm máy tính xách tay của mình vào đó và nhận nguồn điện. Hoặc nhìn vào nó theo cách này: *Adapter thay đổi interface của ổ cắm thành một cái mà máy tính xách tay của bạn mong đợi*.

Một số AC adapters rất đơn giản – chúng chỉ thay đổi hình dạng của ổ cắm sao cho phù hợp với phích cắm của bạn và chúng truyền thẳng dòng điện AC – nhưng các adapter khác có bên trong phức tạp hơn và có thể cần phải tăng hoặc giảm nguồn để phù hợp với nhu cầu của thiết bị.

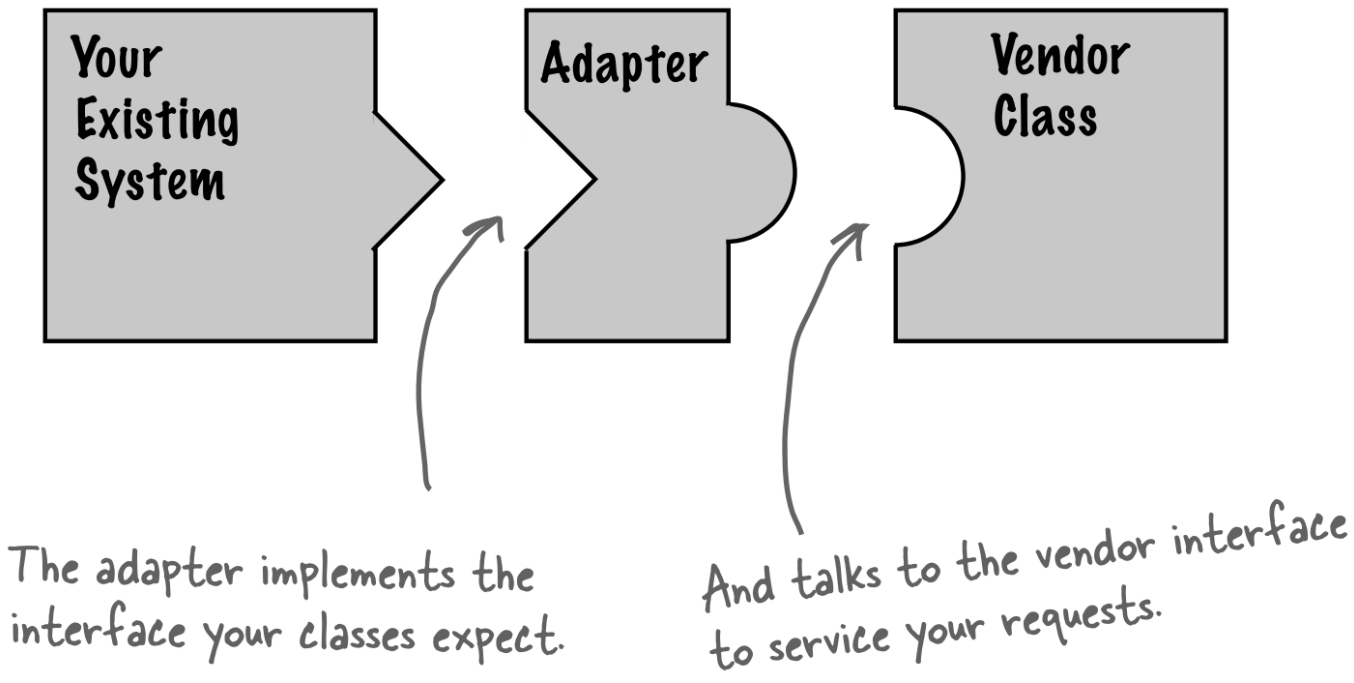
Được rồi, đó là thế giới thực, còn bộ OO adapter thì sao? Chà, OO adapter của chúng tôi đóng vai trò giống như các Adapter trong thế giới thực: nó có một interface và điều chỉnh nó theo interface mà client đang mong đợi.

Adapter pattern trong hướng đối tượng

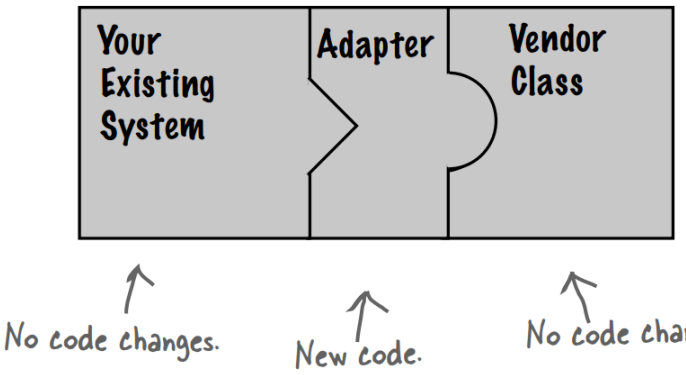
Giả sử bạn **ĐÃ CÓ** một hệ thống phần mềm mà bạn cần làm việc với framework từ một nhà cung cấp mới, nhưng nhà cung cấp mới đã thiết kế giao diện của họ khác với nhà cung cấp trước đó (Hệ thống phần mềm của bạn chỉ làm việc được với giao diện của nhà cung cấp trước đó):



Được rồi, bạn không muốn giải quyết vấn đề bằng cách thay đổi code hiện tại của mình (và bạn cũng không thể thay đổi code của nhà cung cấp mới) (điểm mấu chốt của vấn đề là hệ thống của bạn và các lớp từ nhà cung cấp mới đều không thể thay đổi, khi đó sẽ cần đến Adapter Pattern). Vậy bạn làm gì? Chà, bạn có thể viết một lớp chuyển interface nhà cung cấp mới thành interface mà bạn mong đợi.



Adapter hoạt động như người trung gian bằng cách nhận các yêu cầu từ client (hệ thống của bạn) và chuyển đổi chúng thành các yêu cầu có ý nghĩa đối với các lớp của nhà cung cấp mới.



Can you think of a solution that doesn't require YOU to write ANY additional code to integrate the new vendor classes? How about making the vendor supply the adapter class.

Nếu đó đi giống vịt và kêu giống vịt, khi đó nó ~~phải~~ có thể là một con ~~vịt~~ gà tây được bọc bởi Duck adapter...

Đây là thời gian để xem cách một adapter hoạt động. Nhớ con vịt của chúng ta từ Chương 1 (https://toihocdesignpattern.com/chuong-1-strategy-pattern-chao-mung-den-voi-design-patterns.html) không? Hãy cùng xem lại một phiên bản đơn giản hóa của các interface **Duck**:

```
public interface Duck {
    public void quack();
    public void fly();
}
```

This time around, our ducks implement a Duck interface that allows Ducks to quack and fly.

Ở đây, một lớp con của Duck, **MallardDuck**.

```
public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }

    public void fly() {
        System.out.println("I'm flying");
    }
}
```

Simple implementations: the duck just prints out what it is doing.

Bây giờ, thời gian để gặp gỡ những chú gà tây mới:

```
public interface Turkey {
    public void gobble();
    public void fly();
}
```

Turkeys don't quack, they gobble.

Turkeys can fly, although they can only fly short distances.

```
public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }

    public void fly() {
        System.out.println("I'm flying a short distance");
    }
}
```

Here's a concrete implementation of Turkey; like Duck, it just prints out its actions.

Bây giờ, giả sử bạn thiếu các đối tượng **Duck** và bạn muốn sử dụng một số đối tượng **Turkey** (Gà tây) thay vào đó. Rõ ràng là chúng ta không thể sử dụng **Turkey** hoàn toàn vì chúng có interface khác.

Vì vậy, hãy viết một bộ chuyển đổi Adapter:

Code Up Close

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see.

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts – they can't do long-distance flying like ducks. To map between a Duck's fly() method and a Turkey's, we need to call the Turkey's fly() method five times to make up for it.

Tạo test drive cho Adapter pattern

Bây giờ chúng ta chỉ cần một số dòng code để test adapter của chúng ta:

```
public class DuckTestDrive {
    public static void main(String[] args) {
        MallardDuck duck = new MallardDuck();

        WildTurkey turkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

        System.out.println("The Turkey says...");
        turkey.gobble();
        turkey.fly();

        System.out.println("\nThe Duck says...");
        testDuck(duck);

        System.out.println("\nThe TurkeyAdapter says...");
        testDuck(turkeyAdapter);
    }

    static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

Let's create a Duck...

and a Turkey.

And then wrap the turkey in a TurkeyAdapter, which makes it look like a Duck.

Then, let's test the Turkey: make it gobble, make it fly.

Now let's test the duck by calling the testDuck() method, which expects a Duck object.

Now the big test: we try to pass off the turkey as a duck...

Here's our testDuck() method; it gets a duck and calls its quack() and fly() methods.

Test run

```
File Edit Window Help Don'tForgetToDuck
%java RemoteControlTest
The Turkey says...
Gobble gobble
I'm flying a short distance

The Duck says...
Quack
I'm flying

The TurkeyAdapter says...
Gobble gobble
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
```

The Turkey gobbles and flies a short distance.

The Duck quacks and flies just like you'd expect.

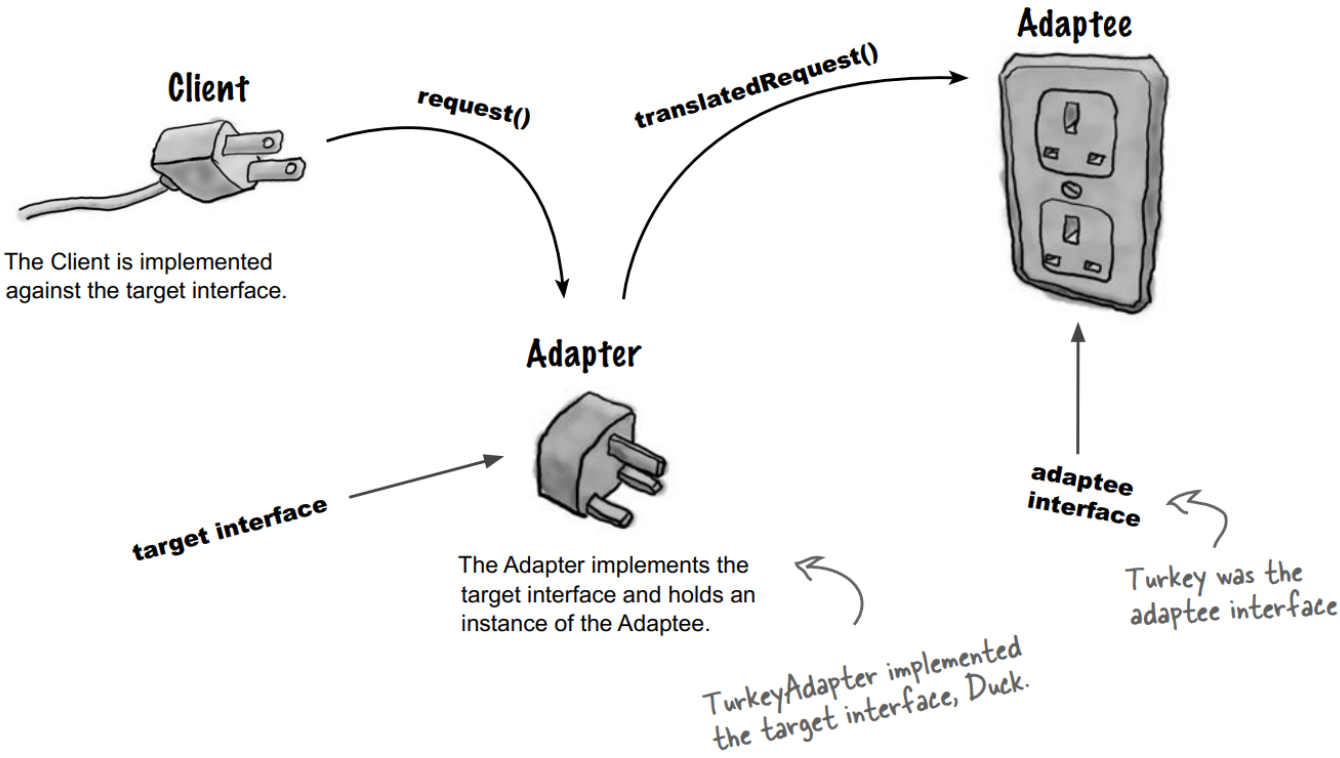
And the adapter gobbles when quack() is called and flies a few times when fly() is called. The testDuck() method never knows it has a turkey disguised as a duck!

Giải thích Adapter Pattern

Để hiểu về Adapter Pattern thì trước hết bạn phải hiểu về 3 khái niệm:

- Client: Đây là lớp sẽ sử dụng đối tượng của bạn (hệ thống của chúng ta).
- Adaptee: Đây là những lớp bạn muốn lớp Client sử dụng, nhưng hiện thời giao diện của nó không phù hợp (hệ thống mới).
- Adapter: Đây là lớp trung gian, thực hiện việc chuyển đổi giao diện cho Adaptee và kết nối Adaptee với Client.

Bây giờ chúng ta đã có ý tưởng về Adapter là gì, hãy xem lại tất cả các phần.



Ở đây, cách Client sử dụng Adapter:

1. Client đưa ra yêu cầu cho adapter bằng cách gọi một phương thức trên nó.

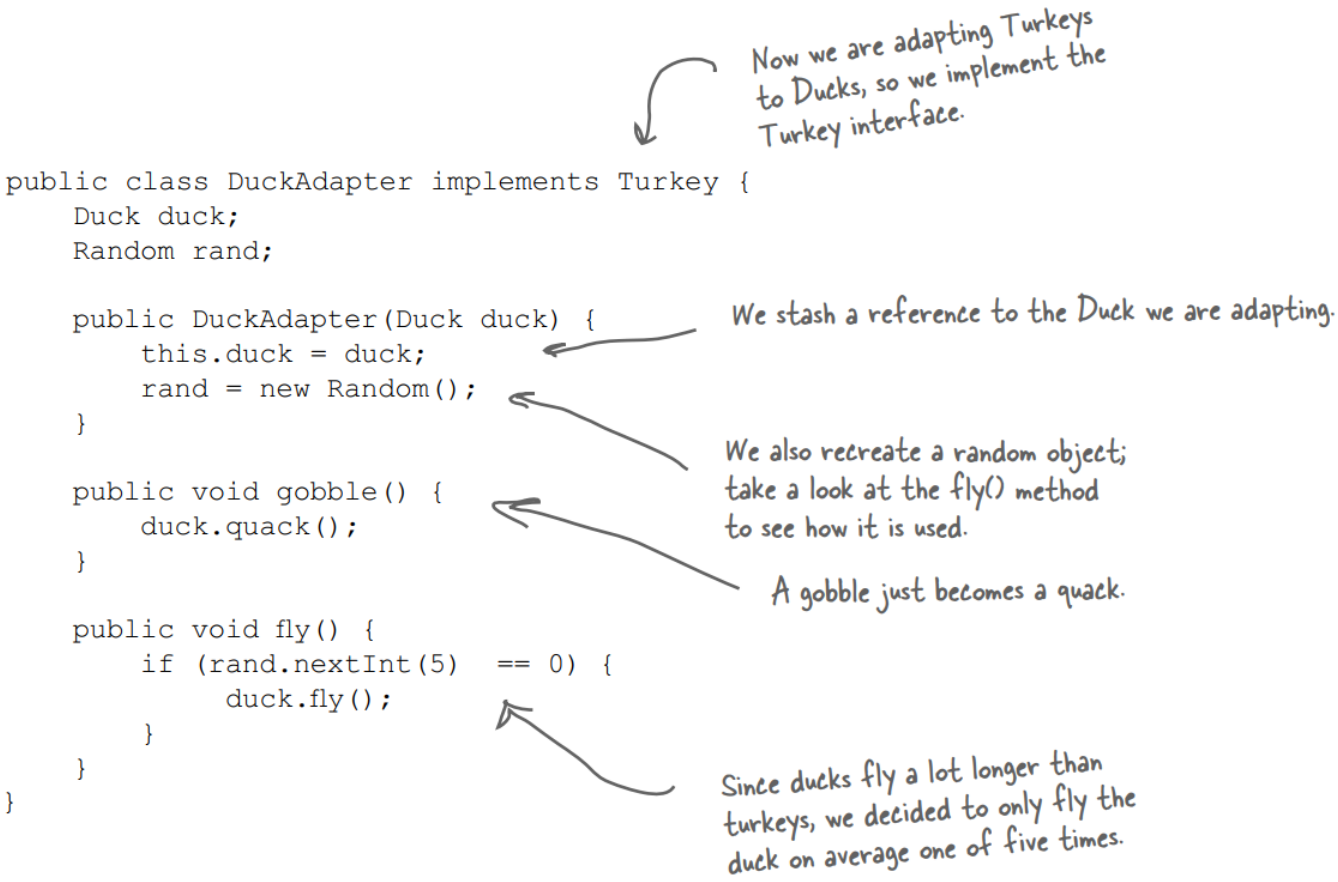
2. Adapter chuyển yêu cầu thành một hoặc nhiều cuộc gọi đến adaptee bằng adaptee interface.
3. Client nhận được kết quả của cuộc gọi và không bao giờ biết có một Adapter đang thực hiện chuyển đổi.

Lưu ý rằng **Client** và **Adaptee** đã được tách rời – chúng không biết về nhau.

BÀI TẬP:

Giả sử chúng ta cũng cần một **Adapter** chuyển đổi **Duck** thành **Turkey** *(ví dụ lúc nãy là **Turkey** chuyển thành **Duck**)*. Hãy gọi nó là **DuckAdapter**. Hãy viết lớp đó.

ĐÁP ÁN:



BÀI TẬP 2:

Bạn xử lý phương thứ **fly** như thế nào?

Không có câu hỏi ngớ ngẩn

Hỏi: Adapter cần bao nhiêu "sự chuyển đổi"? Nó có vẻ như nếu tôi cần chuyển đổi target interface lớn, tôi có thể có RẤT NHIỀU công việc trong tay.

Trả lời: Công việc implement một adapter thực sự tỷ lệ với kích thước của interface bạn cần hỗ trợ giống như target interface của bạn. Hãy suy nghĩ về các lựa chọn của bạn. Bạn có thể phải viết lại tất cả các cuộc gọi phía client của mình đến interface, điều này sẽ dẫn đến rất nhiều công việc điều tra và thay đổi code. Hoặc, bạn có thể cung cấp rõ ràng một lớp, gói gọn tất cả các thay đổi trong một lớp.

Hỏi: Có phải một adapter luôn bao bọc một và chỉ một lớp không?

Trả lời: Vai trò của **Adapter Pattern** là chuyển đổi một interface này sang interface khác. Trong khi hầu hết các ví dụ về adapter pattern chỉ nói đến một adapter bao bọc một lớp adaptee, cả hai chúng ta đều biết thế giới thực thường lộn xộn hơn một chút. Vì vậy, bạn cũng có thể có các tình huống trong đó một adapter giữ hai hoặc nhiều adaptee cần thiết để cài đặt target interface.

Điều này liên quan đến một mẫu khác gọi là Facade Pattern (https://toihocdesignpattern.com/chuong-7-adapter-va-facade-pattern-tro-nen-thich-nghi-phan-2.html); mọi người thường nhầm lẫn hai cái này. Chúng ta sẽ xem lại điều này khi chúng ta nói về facade ở phần sau trong chương này.

Hỏi: Điều gì sẽ xảy ra nếu tôi có cả các phần cũ và mới trong hệ thống của mình, các phần cũ mong đợi interface từ nhà cung cấp cũ, nhưng chúng tôi đã viết các phần mới để sử dụng interface nhà cung cấp mới? Nó sẽ gây nhầm lẫn khi sử dụng một adapter ở đây và interface chưa được mở ở đó. Sẽ tốt hơn hay không nếu tôi chỉ viết code cũ của mình và quên adapter?

Trả lời: Không nhất thiết. Một điều bạn có thể làm là tạo Two Way Adapter hỗ trợ cả hai interface. Để tạo Two Way Adapter, chỉ cần implement cả hai interface liên quan, do đó adapter có thể hoạt động như một interface cũ hoặc interface mới.

Định nghĩa Adapter Pattern

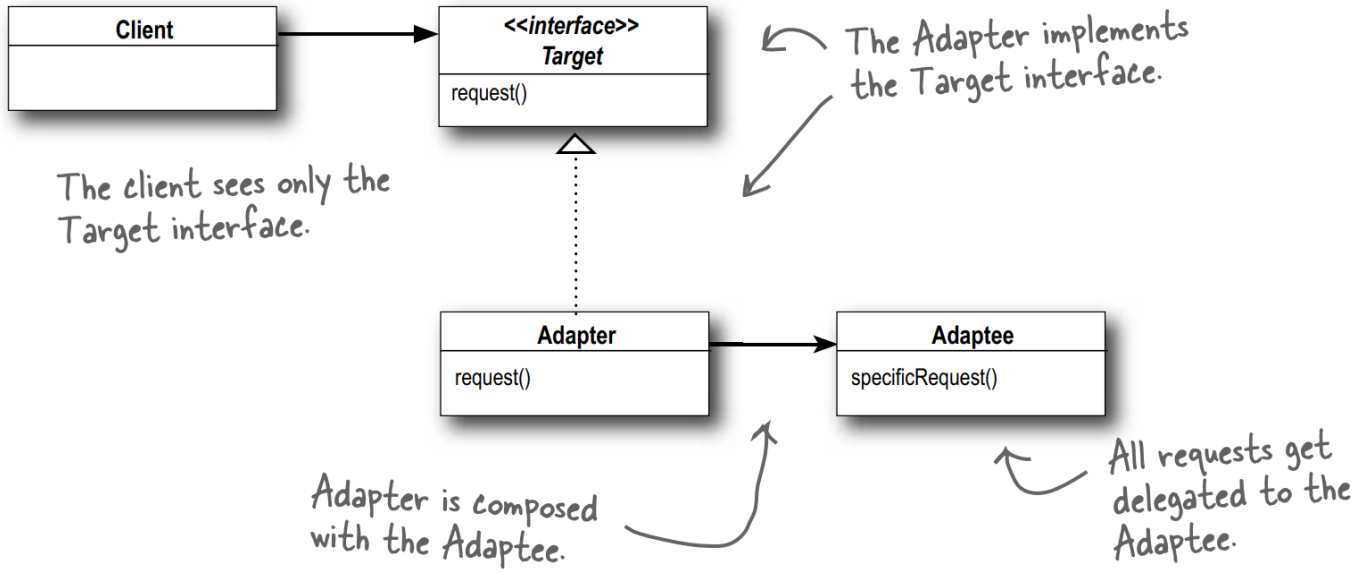
Đã đủ adapter cho vịt, gà tây và bộ nguồn AC; hãy thực tế và nhìn vào định nghĩa chính thức của Adapter Pattern:

> **The Adapter Pattern** converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

(Adapter Pattern chuyển đổi giao diện của một lớp thành một giao diện khác mà client mong đợi. Adapter cho phép các lớp hoạt động cùng nhau mà bình thường là không thể bởi vì sự không tương thích về interface)

Bây giờ, chúng ta biết mẫu này cho phép client sử dụng một interface không tương thích bằng cách tạo Adapter thực hiện chuyển đổi. Điều này có tác dụng tách rời client khỏi interface đã triển khai và nếu chúng ta mong đợi interface thay đổi theo thời gian, adapter sẽ đóng gói thay đổi đó để client không phải sửa đổi mỗi khi cần hoạt động với interface mới.

Chúng tôi đã xem xét hành vi runtime của mẫu Adapter Pattern; hãy xem sơ đồ lớp của nó:



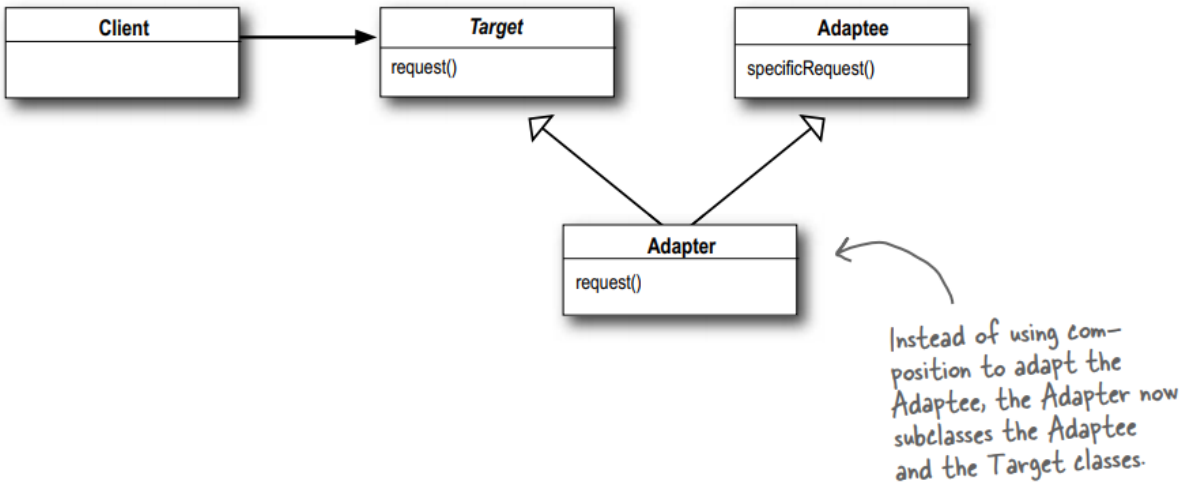
Adapter Pattern có đầy đủ các nguyên tắc thiết kế OO tốt: kiểm tra việc kết hợp các đối tượng (object composition) để bọc adaptee với interface được sửa đổi. Cách tiếp cận này có thêm lợi thế là chúng ta có thể sử dụng adapter với bất kỳ lớp con nào của adaptee.

Ngoài ra, kiểm tra làm thế nào mẫu này có thể liên kết client với một interface, không phải bằng một sự thừa kế; chúng ta có thể sử dụng một vài adapter, mỗi bộ chuyển đổi một tập các lớp backend khác nhau. Hoặc, chúng ta có thể thêm các implementation mới sau đó, miễn là chúng tuân thủ Target interface.

Object adapter và class adapter

Bây giờ mặc dù đã định nghĩa mẫu, chúng tôi vẫn chưa nói cho bạn toàn bộ câu chuyện. Thực tế có hai loại adapter: **object adapter** và **class adapter**. Chương này đã nói về các object adapter và sơ đồ lớp ở trên là sơ đồ của object adapter.

Vậy class adapter là gì và tại sao chúng tôi không nói với bạn về nó? Bởi vì bạn cần đa kế thừa để cài đặt nó, điều này không thể có trong Java (không hỗ trợ đa kế thừa). Nhưng, điều đó không có nghĩa là bạn không có nhu cầu về class adapter khi sử dụng đa kế thừa trong ngôn ngữ yêu thích của bạn! Hãy nhìn vào sơ đồ lớp cho đa kế thừa.



Nhìn có quen không? Đúng vậy – khác biệt duy nhất là với class adapter, chúng ta kế thừa cả **Target** và **Adaptee**, trong khi với object adapter, chúng ta sử dụng composition (kết hợp) để chuyển yêu cầu đến một Adaptee.

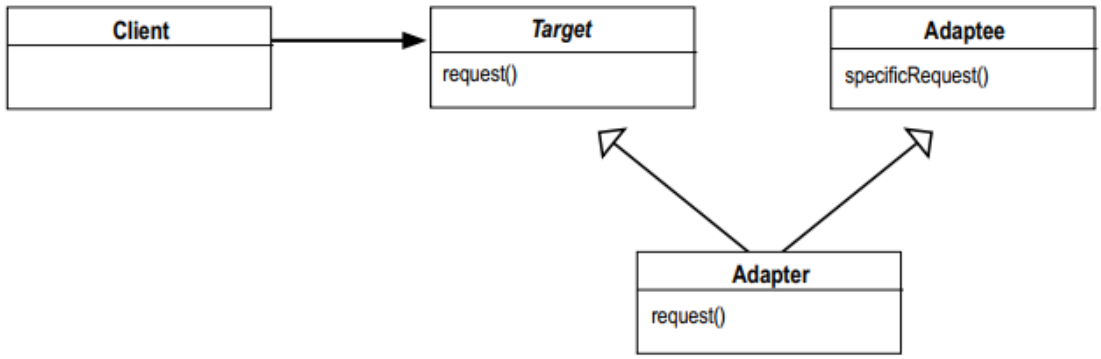
Sử dụng sức mạnh bộ não

Object adapter và class adapter sử dụng hai ý nghĩa khác nhau của adapting và adaptee (composition so với kế thừa). Làm thế nào để những khác biệt thực hiện ảnh hưởng đến tính khả dụng của adapter pattern?

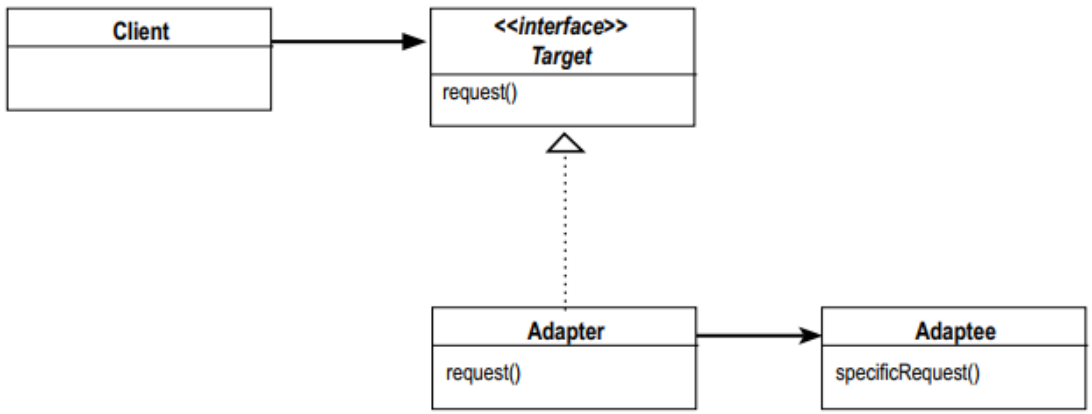
Duck Magnets

Công việc của bạn là lấy duck và turkey magnet và đưa chúng qua một phần của sơ đồ mô tả vai trò của 2 con “chim” đó, trong ví dụ trước đây của chúng tôi (Cố gắng không kéo lên phía trên). Sau đó thêm chú thích của riêng bạn để mô tả cách thức hoạt động của nó.

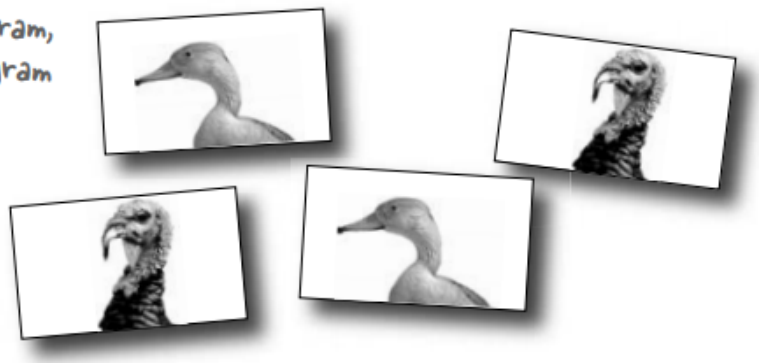
Class Adapter

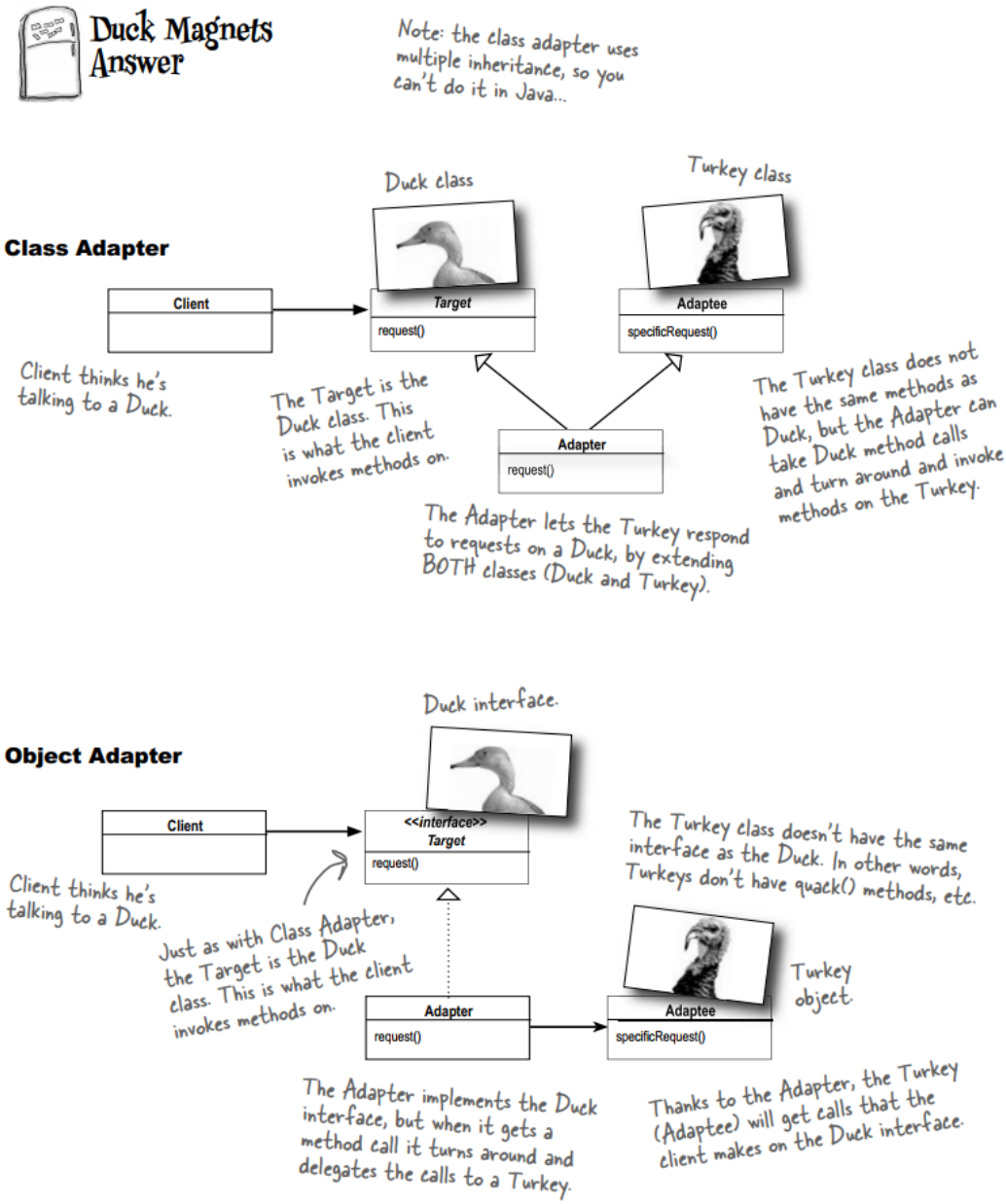


Object Adapter



Drag these onto the class diagram, to show which part of the diagram represents the Duck and which represents the Turkey.





Buổi nói chuyện tối nay: Object Adapter và Class Adapter gặp mặt nhau

Object Adapter: Bởi vì tôi sử dụng composition nên tôi hữu ích. Tôi không chỉ có thể điều chỉnh một lớp adaptee, mà là có thể điều chỉnh bất kỳ lớp con nào của nó.

Class Adapter: Điều đó đúng, tôi gặp rắc rối với điều đó bởi vì tôi được gắn với một lớp adaptee cụ thể, nhưng tôi có một lợi thế rất lớn vì tôi implement lại toàn bộ adaptee. Tôi cũng có thể override hành vi của adaptee nếu tôi cần bởi vì tôi chỉ là subclass.

Object Adapter: Ở thiết kế của tôi, chúng tôi thích sử dụng composition hơn thừa kế; bạn có thể đang lưu một vài dòng code, nhưng tất cả những gì tôi làm là viết một ít code để ủy quyền cho adaptee. Chúng tôi muốn giữ cho mọi thứ linh hoạt.

Class Adapter: Có thể linh hoạt, hiệu quả? Không. Sử dụng class adapter, chỉ có một trong tôi, không phải adapter và adaptee.

Object Adapter: Bạn thường được sử dụng trong một đối tượng nhỏ phải không? Bạn có thể nhanh chóng override một phương thức, nhưng bất kỳ hành vi nào tôi thêm vào adapter của mình đều sẽ làm việc với lớp adaptee và tất cả các lớp con của nó.

Class Adapter: Vâng, nhưng điều gì sẽ xảy ra nếu một lớp con của adaptee thêm một số hành vi mới. Rồi sao?

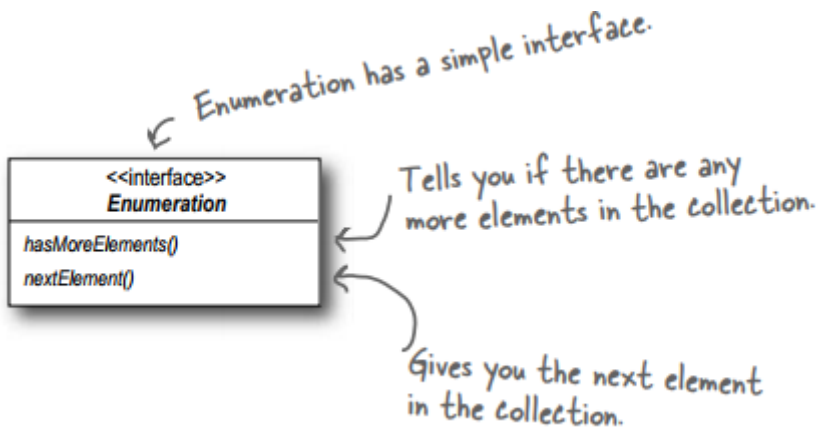
Object Adapter: Này, thôi nào, dừng lại 1 chút, tôi chỉ cần compose với lớp con để làm cho nó hoạt động.

Class Adapter: Nghe có vẻ lộn xộn …

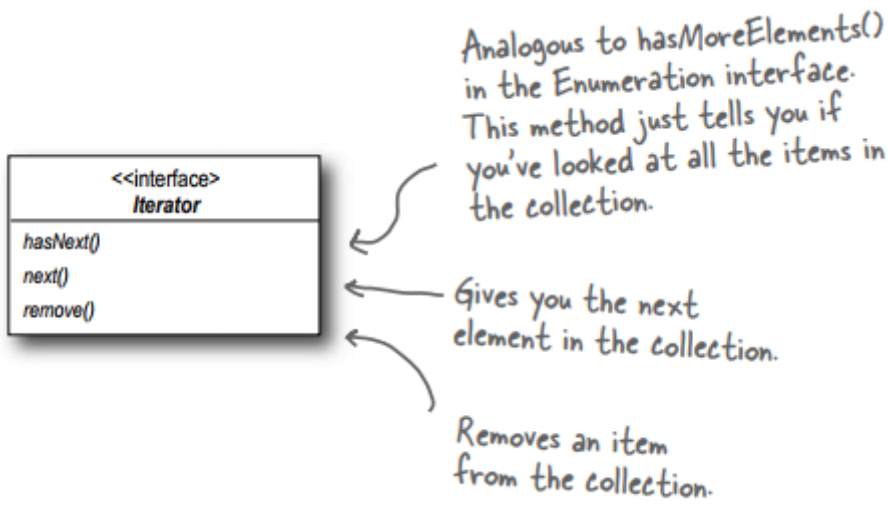
Object Adapter: Bạn muốn thấy lộn xộn? Nhìn vào gương!

Adapter Pattern thế giới thực

Chúng ta hãy xem việc sử dụng một Adapter đơn giản trong thế giới thực…



Enumerator thời cổ xưa



Nếu bạn đã là việc với Java một thời gian, bạn có thể nhớ rằng các kiểu collection ban đầu (**Vector**, **Stack**, **Hashtable** và một vài thứ khác) cài đặt một phương thức **elements()**, trả về một **Enumeration**. Giao diện **Enumeration** cho phép bạn duyệt qua các phần tử của của collection mà không cần biết thông tin cụ thể về cách chúng được quản lý trong collection.

Iterator thời hiện đại

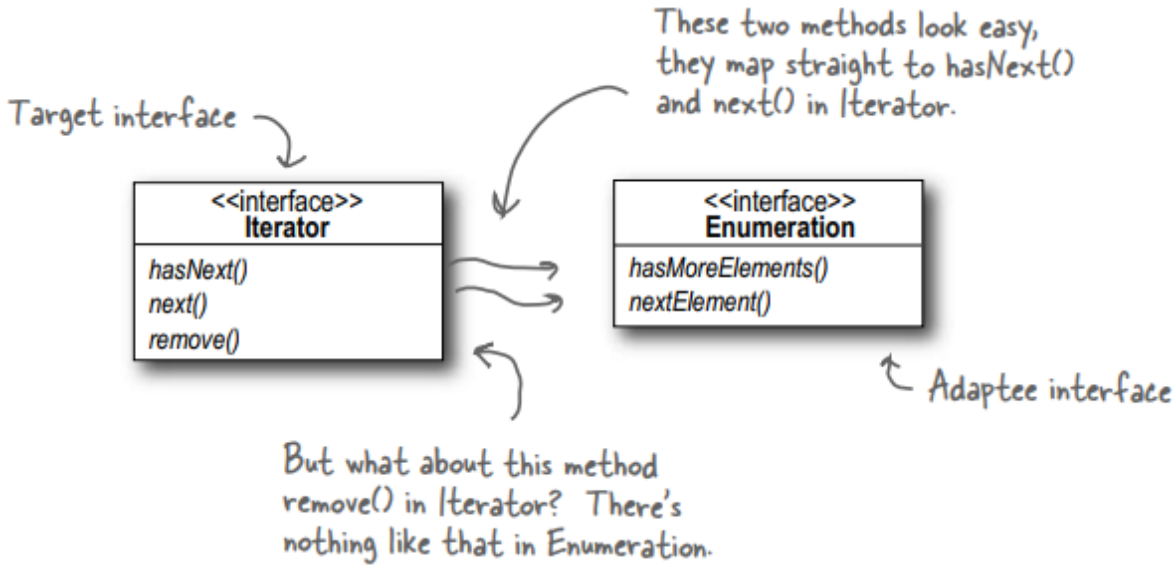
Khi Sun phát hành các lớp Collection gần đây hơn, họ bắt đầu sử dụng giao diện **Iterator**, như **Enumeration**, iterator cũng cho phép bạn duyệt qua một tập hợp các phần tử trong collection, nhưng được thêm khả năng xóa các phần tử.

Và bây giờ...

Chúng tôi thường phải đối mặt với code được để lại, chúng thể hiện Enumerator interface, nhưng chúng tôi đã sử dụng code mới để chỉ sử dụng các Iterator. Có vẻ như chúng ta cần xây dựng một adapter.

Chuyển đổi một Iterator thành Enumeration

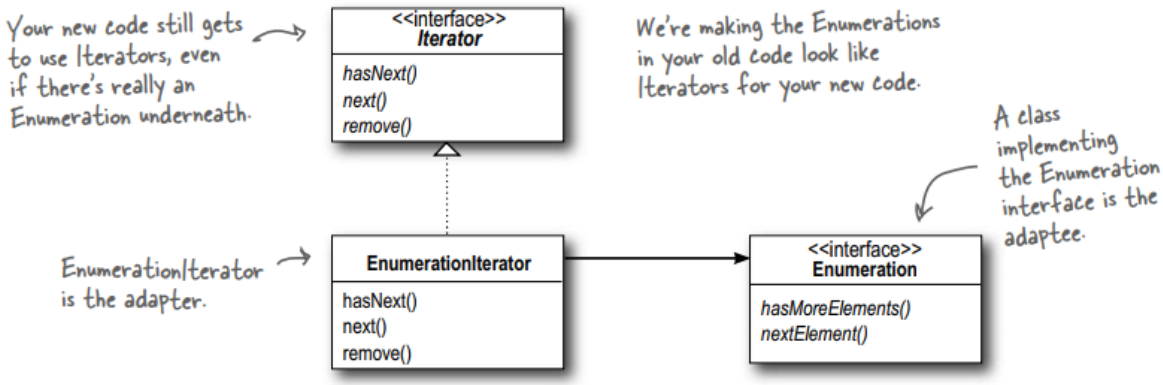
Đầu tiên, chúng tôi sẽ xem xét hai giao diện để tìm hiểu cách các phương thức ánh xạ từ cái này sang cái kia. Nói cách khác, chúng tôi sẽ tìm ra những gì cần gọi trên adaptee khi client gọi một phương thức trên adapter.



Thiết kế Adapter

Ở đây, các lớp sẽ trông như: chúng ta cần một adapter thực hiện giao diện Target interface (Iterator) và được kết hợp (conpose) với một adaptee. Các phương thức **hasNext()** và **next()** rất đơn giản để map từ target đến adaptee: chúng ta chỉ cần thực hiện chúng ngay bây giờ. Nhưng bạn sẽ làm gì với phương thức **remove()**? Hãy suy nghĩ về nó một lát (và chúng tôi sẽ đối phó với nó phía bên dưới).

Bây giờ, ở đây, sơ đồ lớp:



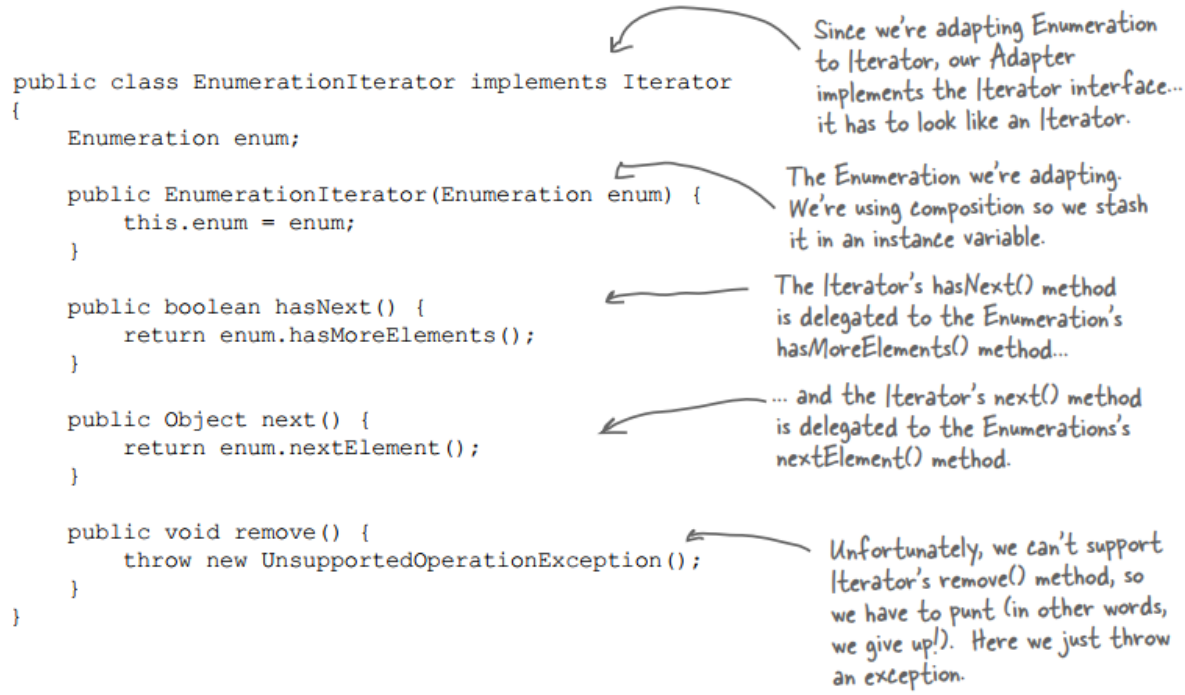
Xử lý phương thức remove()

Chà, chúng tôi biết Enumeration không hỗ trợ **remove**. Nó là một "read only" interface. Không có cách nào để kế thừa phương thức **remove()** đầy đủ chức năng trên adapter. Điều tốt nhất chúng ta có thể làm là ném runtime exception. May mắn thay, các nhà thiết kế giao diện Iterator đã thấy trước nhu cầu này và định nghĩa phương thức **remove()** để nó hỗ trợ **UnsupportedOperationException**.

Đây là một trường hợp mà adapter không hoàn hảo; client sẽ phải để ý các trường hợp ngoại lệ tiềm ẩn, nhưng miễn là client cẩn thận và adapter có document đầy đủ thì đây là một giải pháp hoàn toàn hợp lý.

Viết adapter EnumerationIterator

Ở đây, code đơn giản nhưng hiệu quả cho tất cả các lớp kế thừa vẫn đang tạo ra **Enumeration**:



BÀI TẬP:

Mặc dù Java đã đi theo hướng của **Iterator**, tuy nhiên vẫn có rất nhiều client code phụ thuộc vào giao diện **Enumeration**, do đó, một Adapter chuyển đổi **Iterator** thành **Enumeration** cũng khá hữu ích.

Viết một Adapter điều chỉnh **Iterator** thành một **Enumeration**. Bạn có thể test code của mình bằng cách chuyển đổi một **ArrayList**. Lớp **ArrayList** hỗ trợ giao diện **Iterator** nhưng không hỗ trợ **Enumeration**.

ĐÁP ÁN:

```
public class IteratorEnumeration implements Enumeration {
    Iterator iterator;

    public IteratorEnumeration(Iterator iterator) {
        this.iterator = iterator;
    }

    public boolean hasMoreElements() {
        return iterator.hasNext();
    }

    public Object nextElement() {
        return iterator.next();
    }
}
```

Sử dụng sức mạnh bộ não

Một số AC adapter không chỉ thay đổi giao diện – chúng còn thêm các tính năng khác như bảo vệ khẩn cấp (như tự động ngắt nguồn), đèn tín hiệu và các loại chuông và còi khác.

Nếu bạn định triển khai các loại tính năng này, bạn sẽ sử dụng mẫu nào?

Buổi nói chuyện tối nay: Decorator Pattern (https://toihocdesignpattern.com/chuong-3-head-first-design-patterns-tieng-viet-decorator-pattern-doi-tuong-trang-tri.html) và Adapter Pattern thảo luận về sự khác biệt của họ

Decorator: Tôi quan trọng hơn. Công việc của tôi là tất cả về trách nhiệm (responsibility) – bạn biết rằng khi cài đặt thêm một Decorator có liên quan thì sẽ có một số trách nhiệm hoặc hành vi mới được thêm vào thiết kế của bạn.

Adapter: Bạn muốn tất cả vinh quang trong khi các adapter của chúng tôi đang ở trong hậu trường và làm những công việc ít người biết đến: chuyển đổi các interface. Công việc của chúng tôi có thể không hào nhoáng, nhưng client của chúng tôi chắc chắn đánh giá cao vì chúng tôi làm cho cuộc sống của họ đơn giản hơn.

Decorator: Điều đó có thể đúng, nhưng đừng nghĩ rằng chúng tôi không làm việc chăm chỉ. Khi chúng ta phải decorate một giao diện lớn, whoa, có thể giảm rất nhiều code.

Adapter: Hãy thử trở thành một adapter khi bạn phải kết hợp nhiều lớp để cung cấp giao diện mà client của bạn mong đợi. Bây giờ thì khó khăn. Nhưng chúng tôi có một câu nói: “an uncoupled client is a happy client” (Một client tách biệt xử lí là một client hạnh phúc).

Decorator: Dễ thương đấy. Đừng nghĩ rằng chúng tôi có được tất cả vinh quang; đôi khi tôi chỉ là một decorator đang được bao bọc bởi những người biết có bao nhiêu decorator khác. Khi một cuộc gọi phương thức được ủy quyền cho bạn, bạn sẽ không biết có bao nhiêu decorator khác đã xử lý nó và bạn không biết rằng bạn sẽ được chú ý vì những nỗ lực của bạn phục vụ yêu cầu.

Adapter: Này, nếu các adapter đang làm công việc của họ, client của chúng tôi thậm chí không bao giờ biết chúng tôi đang ở đó. Nó có thể là một công việc vô ơn.

Nhưng, điều tuyệt vời về adapter của chúng tôi là chúng tôi cho phép client sử dụng các thư viện và tập hợp con mới mà không thay đổi bất kỳ đoạn code nào, họ chỉ dựa vào chúng tôi để thực hiện chuyển đổi cho họ. Này, nó là một phần nhỏ, nhưng chúng tôi rất giỏi về nó.

Decorator: Các decorator của chúng tôi cũng làm điều đó, chỉ chúng tôi cho phép hành vi mới được thêm vào các lớp mà không thay đổi code hiện có. Tôi vẫn nói rằng các adapter chỉ là những decorator "lạ mắt" – ý tôi là, giống như chúng tôi, bạn cũng bao bọc một object.

Adapter: Không, không, không, không hề. Chúng tôi luôn chuyển đổi interface của những gì chúng tôi bọc bên trong, bạn không bao giờ làm điều này. Tôi nói rằng một decorator giống như một adapter; chỉ là bạn không thay đổi giao diện!

Decorator: À, không. Công việc của chúng tôi trong cuộc sống là mở rộng các hành vi hoặc trách nhiệm của các đối tượng chúng tôi bọc, chúng tôi không phải là một "mẫu đơn giản để hiểu" (simple pass through)..

Adapter: Này, bạn đang gọi ai là "mẫu đơn giản để hiểu"? Xem tiếp và chúng ta sẽ thấy bạn chuyển đổi một vài giao diện trong bao lâu!

Decorator: Có lẽ chúng ta nên đồng ý hay không đồng ý. Chúng ta dường như trông hơi giống nhau trên giấy, nhưng rõ ràng chúng ta cách xa nhau trong ý định của chúng ta.

Adapter: Ồ vâng, tôi đồng ý với bạn.

(còn tiếp.., (https://toihocdesignpattern.com/chuong-4-factory-pattern-va-abstract-factory-pattern-phan-2.html)phần 2 (https://toihocdesignpattern.com/chuong-7-adapter-va-facade-pattern-tro-nen-thich-nghi-phan-2.html)).