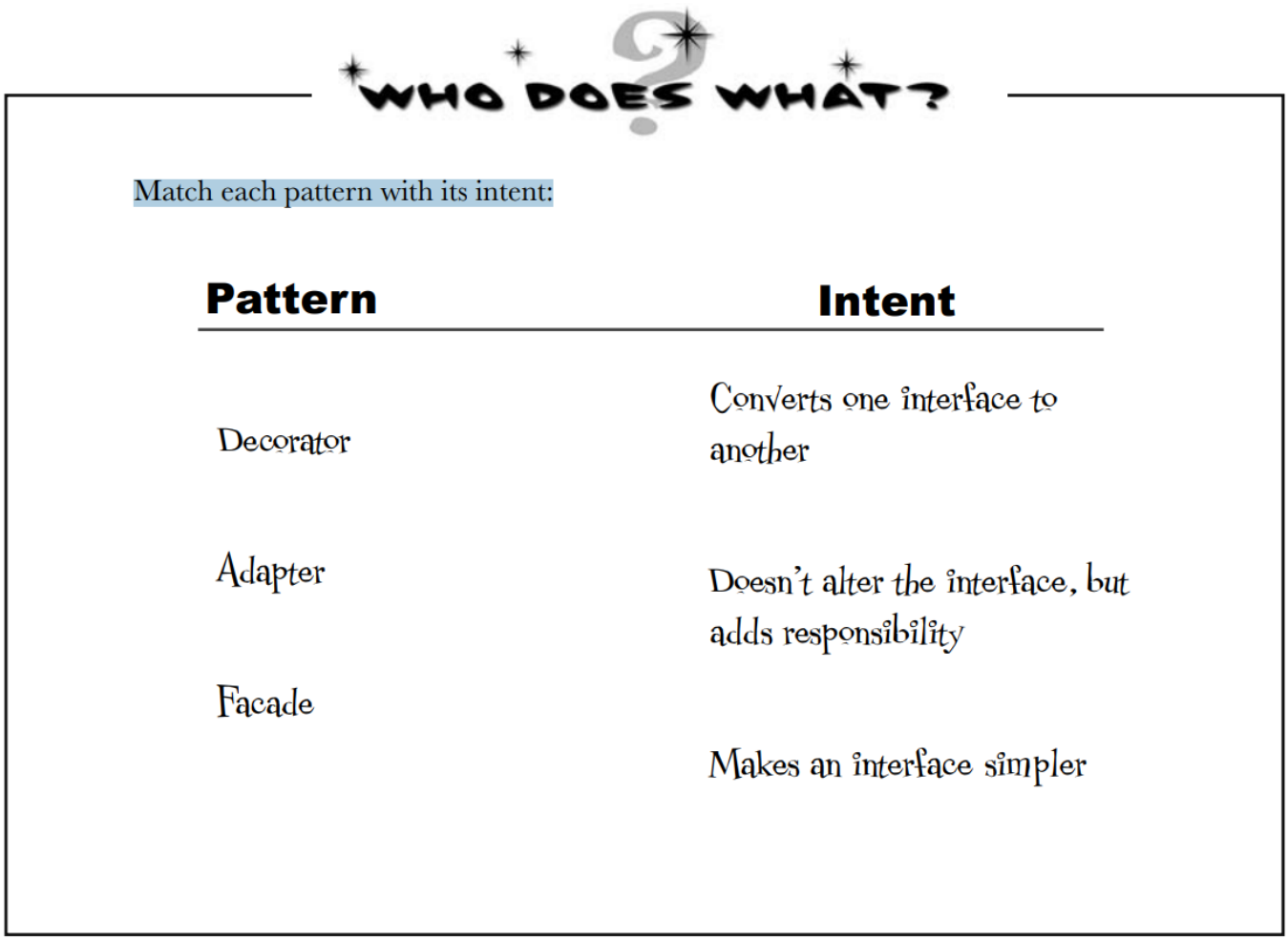
Chương 7: Adapter Pattern và Facade Pattern – Trở nên thích nghi (P2)

Có một Facade Pattern trong chương này

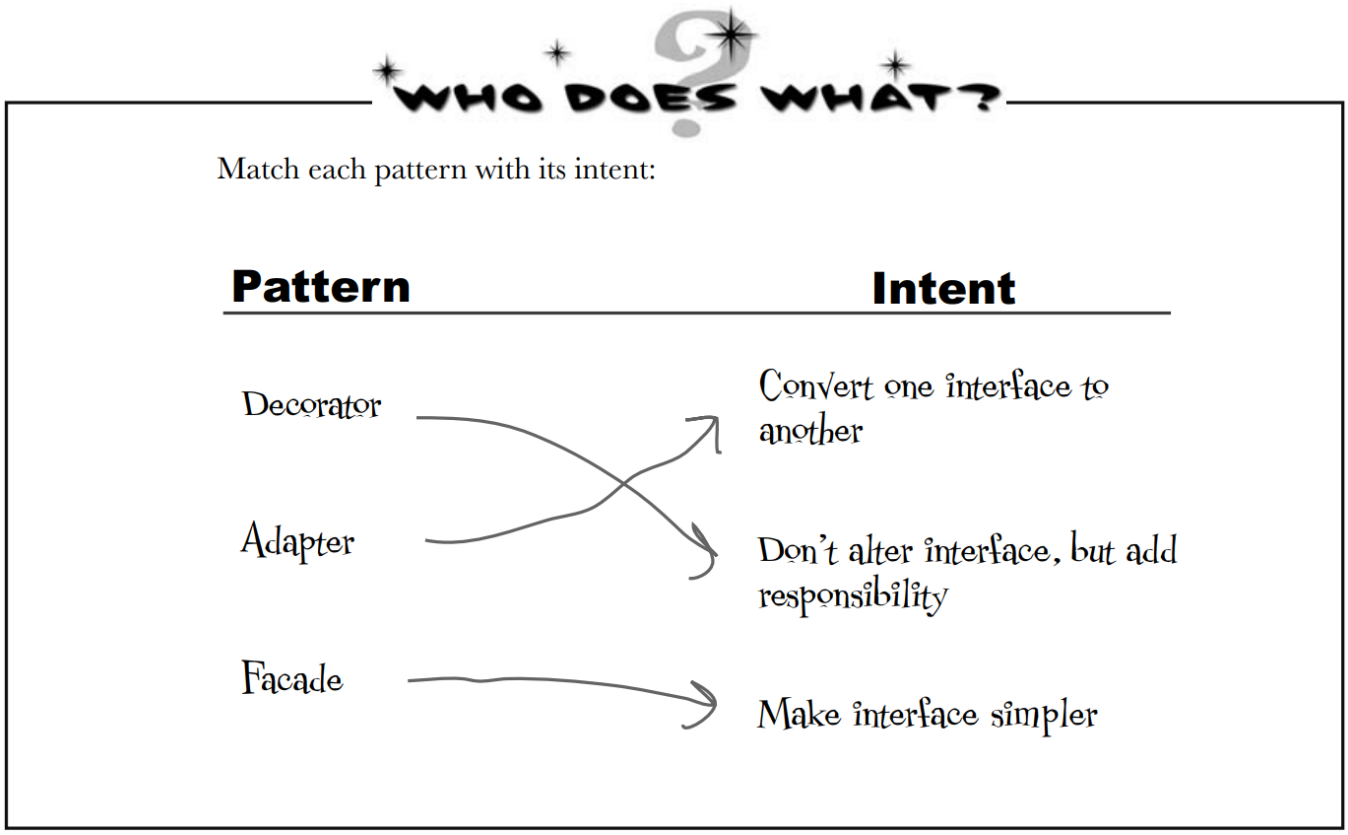
Bạn đã thấy cách Adapter Pattern (https://toihocdesignpattern.com/chuong-7-adapter-va-facade-pattern-tro-nen-thich-nghi-(phan-1).html) chuyển đổi giao diện của một lớp thành một giao diện mà client đang mong đợi. Bạn cũng biết chúng ta đạt được điều này trong Java bằng cách bọc đối tượng có interface không tương thích bằng một đối tượng được implement từ interface tương thích.

Bây giờ chúng ta sẽ xem xét một mô hình có thể thay đổi giao diện, nhưng cho một lý do khác: để đơn giản hóa giao diện. Nó được đặt tên một cách khéo léo là **Facade Pattern** (mặt tiền) vì mẫu này che giấu tất cả sự phức tạp của một hoặc nhiều lớp đằng sau mặt tiền sạch sẽ, thiết kế rõ ràng.

Để theo dõi tiếp, hãy hoàn thành bài tập sau đây trước khi xem đáp án:



Đáp án:

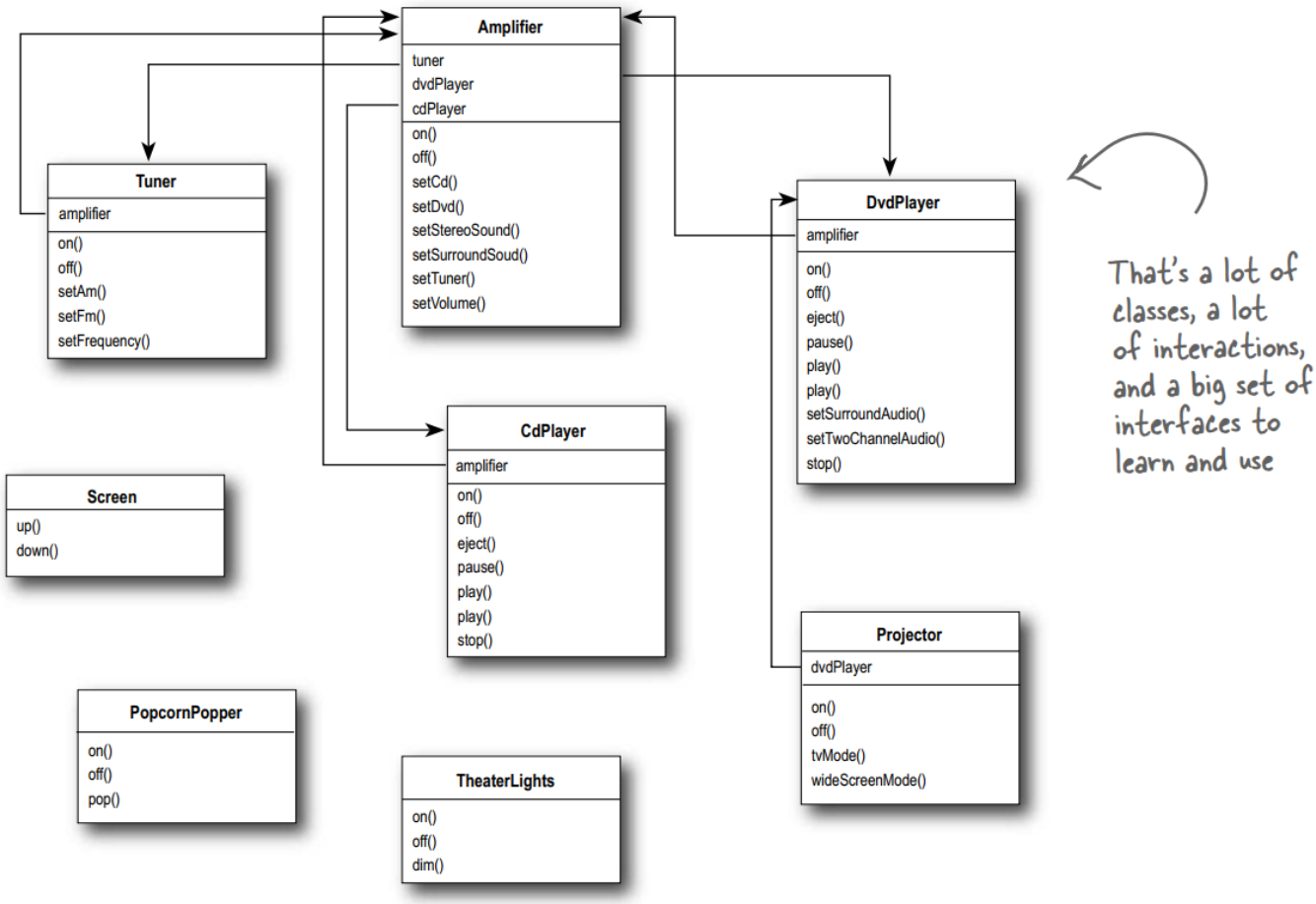


Nhà hát tại gia đình

Trước khi chúng ta đi sâu vào các chi tiết của **Facade Pattern**, hãy cùng xem qua ví dụ về việc: xây dựng nhà hát gia đình của riêng bạn.

Sau một lúc tự tìm hiểu và mài mò, và bạn đã lắp ráp một hệ thống hoàn chỉnh với đầu DVD, hệ thống video chiếu, màn hình tự động, âm thanh vòm và thậm chí là một bỏng ngô.

Kiểm tra tất cả các thành phần bạn đặt cùng nhau:



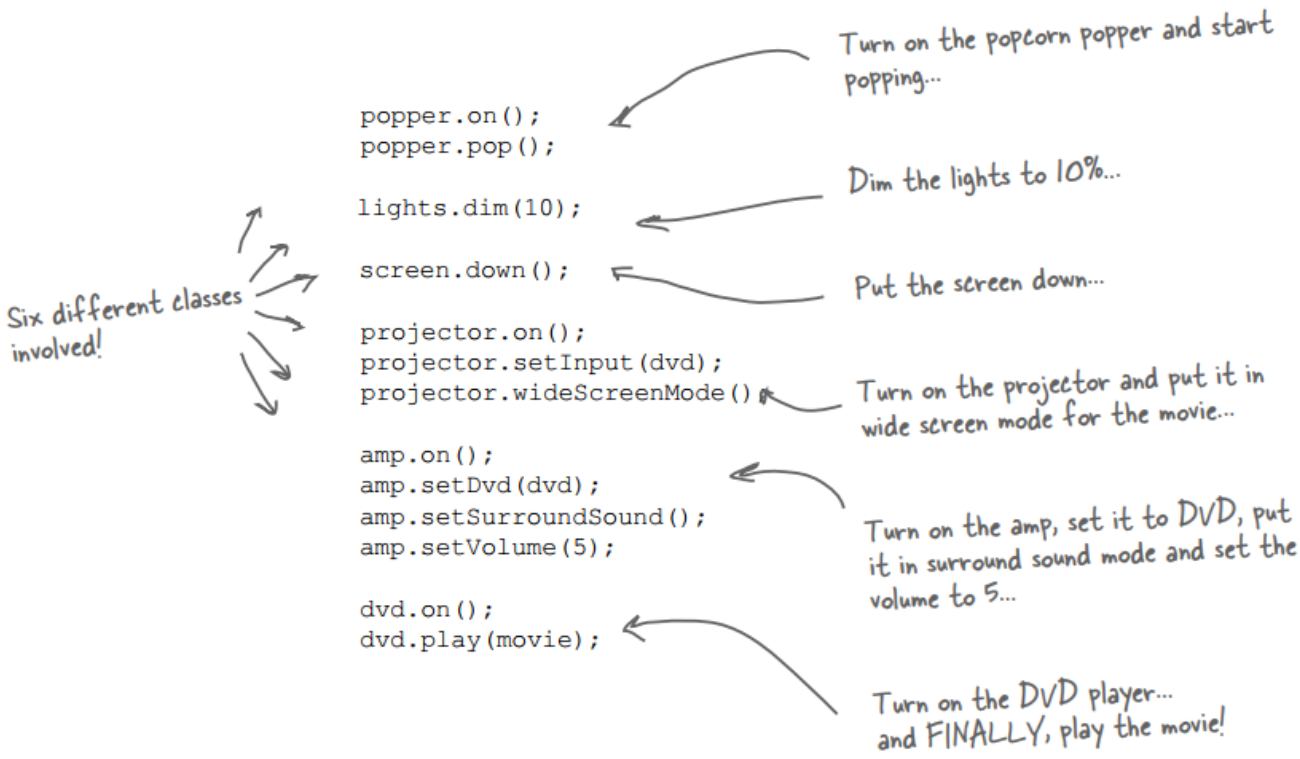
Bạn đã dành hàng tuần để đấu dây, lắp máy chiếu, thực hiện tất cả các kết nối và điều chỉnh. Bây giờ, thời gian để đưa tất cả vào chuyển động và thưởng thức một bộ phim…

Xem một bộ phim (theo một cách khó khăn)

Chọn ra một đĩa DVD, thư giãn và sẵn sàng cho phim phép thuật. Ồ, có một điều duy nhất – để xem phim, bạn cần thực hiện một vài nhiệm vụ:

1. Bật chế độ ăn bỏng ngô
2. Bắt đầu lấy bỏng ngô
3. Giảm ánh sáng
4. Đặt màn hình xuống
5. Bật máy chiếu
6. Đặt đầu vào máy chiếu là DVD
7. Đặt máy chiếu ở chế độ màn hình rộng
8. Bật bộ amplifier
9. Đặt đầu vào bộ amplifier thành DVD
10. Đặt đầu ra bộ amplifier thành âm thanh vòm
11. Đặt âm lượng bộ khuếch đại thành medium (5)
12. Bật đầu DVD
13. Bắt đầu phát DVD Player

Hãy cùng kiểm tra các nhiệm vụ tương tự về các lớp và các lệnh gọi phương thức cần thiết để thực hiện chúng:



Nhưng có thêm...

Khi bộ phim kết thúc, làm thế nào để bạn tắt mọi thứ?

- Bạn có muốn làm lại tất cả những điều này một lần nữa không?
- Sẽ rất phức tạp khi nghe CD hay radio?
- Nếu bạn quyết định nâng cấp hệ thống của mình, có lẽ bạn sẽ phải học một thêm một quy trình khác nữa.

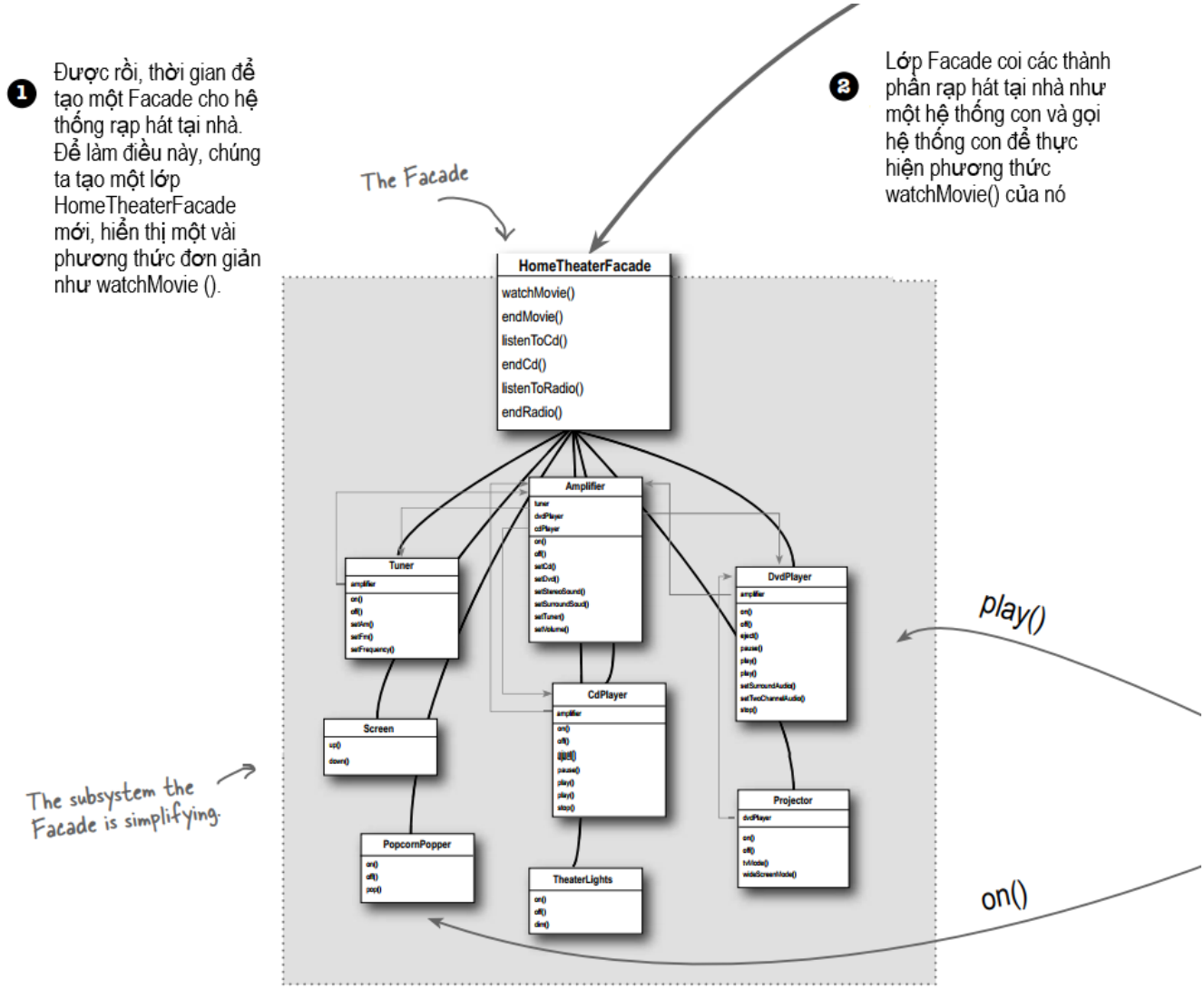
Vậy làm gì bây giờ? Sự phức tạp của việc sử dụng rạp hát tại nhà của bạn đang trở nên rõ ràng!

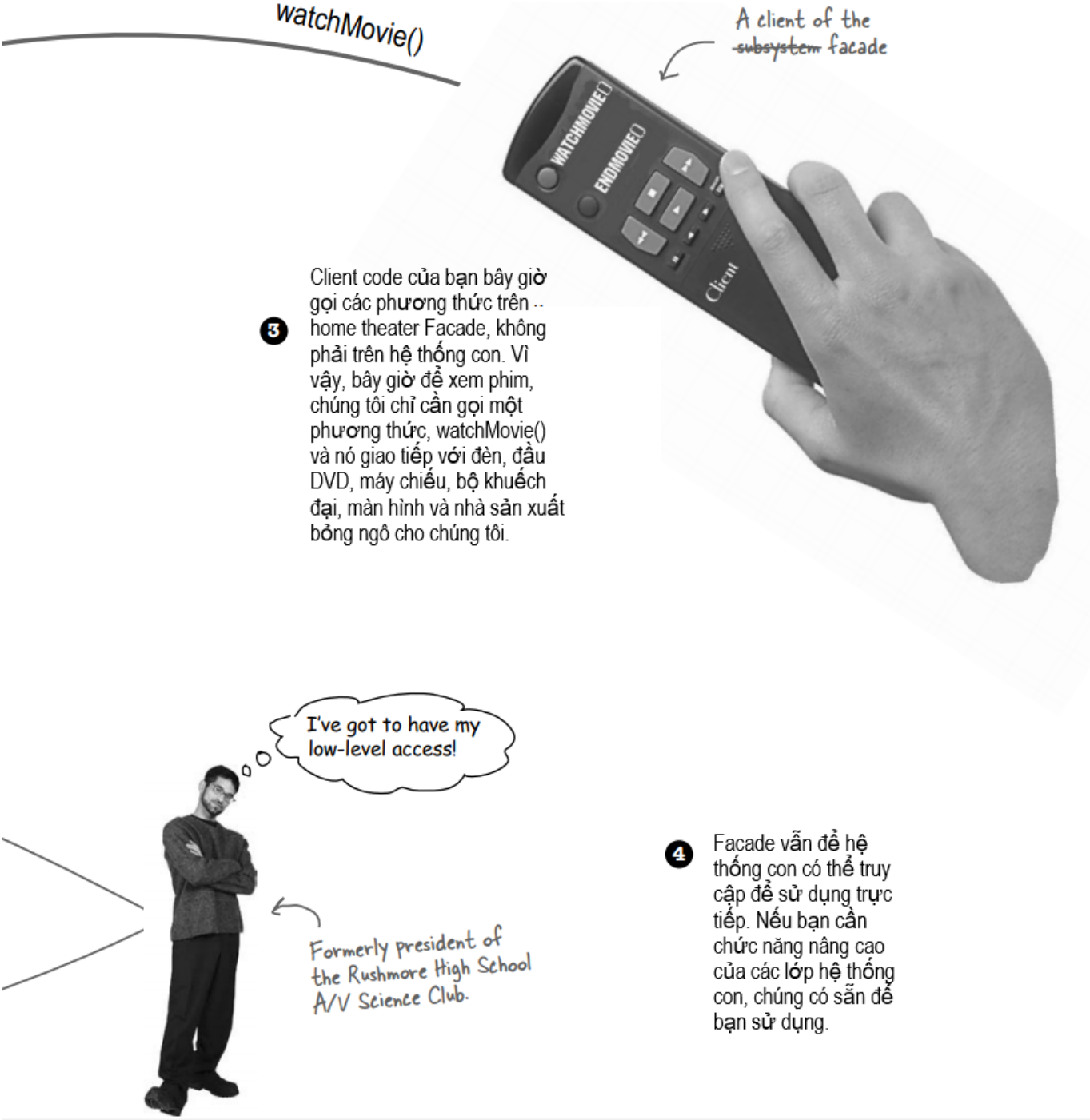
Hãy cùng xem, làm thế nào **Facade Pattern** có thể giúp chúng ta thoát khỏi mớ hỗn độn này để chúng ta có thể thưởng thức bộ phim.

Lights, Camera, Facade pattern!

Facade Pattern chính là thứ bạn cần: với **Facade Pattern**, bạn có thể có một hệ thống con phức tạp và làm cho nó dễ sử dụng hơn bằng cách triển khai lớp Facade cung cấp một giao diện hợp lý hơn. Đừng lo lắng vì mẫu sẽ không làm thay đổi hệ thống phức tạp của bạn; Nếu bạn cần sức mạnh của một hệ thống con phức tạp, nó vẫn ở đó để bạn sử dụng, nhưng nếu tất cả những gì bạn cần là một giao diện đơn giản, thì Facade pattern luôn sẵn sàng cho bạn.

Hãy cùng nhìn vào cách thức hoạt động của Facade:





Không có câu hỏi ngớ ngẩn

Hỏi: Nếu **Facade** đóng gói các lớp ở hệ thống con, làm thế nào một client vẫn có quyền truy cập vào hệ thống con đó?

Trả lời: Facade không đóng gói các lớp hệ thống con; **Facade** chỉ đơn thuần cung cấp một giao diện đơn giản hóa cho chức năng của chúng. Các lớp hệ thống con vẫn có sẵn để sử dụng trực tiếp bởi các client cần sử dụng các giao diện cụ thể hơn. Đây là một thuộc tính tuyệt vời của **Facade Pattern**: nó cung cấp giao diện đơn giản hóa trong khi vẫn có đầy đủ chức năng của hệ thống cho những người có thể cần nó.

Hỏi: **Facade** có thêm bất kỳ chức năng nào không hay nó chỉ đưa các yêu cầu đến hệ thống con?

Trả lời: Một **Facade** có thể tự do thêm vào đó là những chức năng của riêng mình ngoài việc sử dụng hệ thống con. Ví dụ, trong khi home threater facade của chúng ta không thực hiện bất kỳ hành vi mới nào, nó đủ thông minh để biết rằng bỏng ngô phải được "turn on" trước khi nó có thể ăn (pop) (cũng như các chi tiết về cách bật và chiếu một bộ phim đang chiếu).

Hỏi: Mỗi hệ thống con chỉ có một **Facade** ư?

Trả lời: Không nhất thiết. Mẫu chắc chắn cho phép bất kỳ số lượng **Facade** nào được tạo cho một hệ thống con nhất định.

Hỏi: Lợi ích khác của **Facade** ngoài việc tôi có giao diện đơn giản hơn là gì?

Trả lời: Facade Pattern cũng cho phép bạn tách rời triển khai ứng dụng client của mình khỏi bất kỳ một hệ thống con nào. Chẳng hạn như bạn được tăng lương và quyết định nâng cấp rạp hát tại nhà của mình lên tất cả các thành phần mới có giao diện khác nhau.

Chà, nếu bạn đã code ứng dụng client của mình sang **Facade** chứ không phải hệ thống con, client code của bạn không cần thay đổi, chỉ cần **Facade** thay đổi (và hy vọng nhà sản xuất đang cung cấp điều đó!).

Hỏi: Vậy cách để nói về sự khác biệt giữa Mẫu Adapter (https://toihocdesignpattern.com/chuong-7-adapter-va-facade-pattern-tro-nen-thich-nghi-phan-1.html) và Mẫu **Facade** là adapter (https://toihocdesignpattern.com/chuong-7-adapter-va-facade-pattern-tro-nen-thich-nghi-phan-1.html) bao bọc một lớp còn facade có thể đại diện cho nhiều lớp không?

Trả lời: Không! Hãy nhớ rằng, Adapter (https://toihocdesignpattern.com/chuong-7-adapter-va-facade-pattern-tro-nen-thich-nghi-phan-1.html) thay đổi giao diện của **một hoặc nhiều** (chứ không phải một) lớp thành một giao diện mà client đang mong đợi (Trong khi hầu hết các ví dụ trong sách này cho thấy adapter (https://toihocdesignpattern.com/chuong-7-adapter-va-facade-pattern-tro-nen-thich-nghi-phan-1.html) chỉ làm việc với một lớp), bạn có thể cần phải điều chỉnh nhiều lớp để cung cấp giao diện mà client yêu cầu. Tương tự như vậy, một Facade có thể cung cấp một giao diện đơn giản hóa từ một (hoặc nhiều) lớp có giao diện rất phức tạp.

Sự khác biệt giữa hai loại này không nằm ở việc chúng có bao nhiêu lớp mà chúng bao bọc, khác biệt là ý định của chúng. Mục đích của Mẫu Adapter (https://toihocdesignpattern.com/chuong-7-adapter-va-facade-pattern-tro-nen-thich-nghi-phan-1.html) là thay đổi giao diện sao cho phù hợp với giao diện mà client đang mong đợi. Mục đích của Mẫu Facade là cung cấp giao diện đơn giản cho hệ thống con.

Một Facade không chỉ đơn giản hóa một giao diện, nó tách rời một client khỏi một hệ thống con của các thành phần. Facade và Adapter (https://toihocdesignpattern.com/chuong-7-adapter-va-facade-pattern-tro-nen-thich-nghi-phan-1.html) có thể bao bọc nhiều lớp, nhưng mục đích của Facade là để đơn giản hóa, trong khi một Adapter (https://toihocdesignpattern.com/chuong-7-adapter-va-facade-pattern-tro-nen-thich-nghi-phan-1.html) là để chuyển đổi giao diện sang một dạng khác.

Xây dựng nhà hát gia đình bằng Facade Pattern

Hãy cùng nhau bước qua việc xây dựng **HomeTheaterFacade**: Bước đầu tiên là sử dụng composition để facade có quyền truy cập vào tất cả các thành phần của hệ thống con:

```
public class HomeTheaterFacade {
    Amplifier amp;
    Tuner tuner;
    DvdPlayer dvd;
    CdPlayer cd;
    Projector projector;
    TheaterLights lights;
    Screen screen;
    PopcornPopper popper;

    public HomeTheaterFacade(Amplifier amp,
                Tuner tuner,
                DvdPlayer dvd,
                CdPlayer cd,
                Projector projector,
                Screen screen,
                TheaterLights lights,
                PopcornPopper popper) {

        this.amp = amp;
        this.tuner = tuner;
        this.dvd = dvd;
        this.cd = cd;
        this.projector = projector;
        this.screen = screen;
        this.lights = lights;
        this.popper = popper;
    }

        // other methods here

}
```

Here's the composition; these are all the components of the subsystem we are going to use.

The facade is passed a reference to each component of the subsystem in its constructor. The facade then assigns each to the corresponding instance variable.

We're just about to fill these in...

Cài đặt giao diện đơn giản hóa

Bây giờ, thời gian để đưa các thành phần của hệ thống con lại với nhau thành một giao diện hợp nhất.

Hãy thực hiện các phương thức **watchMovie()** và **endMovie()**:

```
public void watchMovie(String movie) {
    System.out.println("Get ready to watch a movie...");
    popper.on();
    popper.pop();
    lights.dim(10);
    screen.down();
    projector.on();
    projector.wideScreenMode();
    amp.on();
    amp.setDvd(dvd);
    amp.setSurroundSound();
    amp.setVolume(5);
    dvd.on();
    dvd.play(movie);
}

public void endMovie() {
    System.out.println("Shutting movie theater down...");
    popper.off();
    lights.on();
    screen.up();
    projector.off();
    amp.off();
    dvd.stop();
    dvd.eject();
    dvd.off();
}
```

watchMovie() follows the same sequence we had to do by hand before, but wraps it up in a handy method that does all the work. Notice that for each task we are delegating the responsibility to the corresponding component in the subsystem.

And endMovie() takes care of shutting everything down for us. Again, each task is delegated to the appropriate component in the subsystem.

Sử dụng sức mạnh bộ não bạn

Hãy suy nghĩ về các **Facade** bạn đã gặp trong API Java. Bạn muốn có một vài cái mới ở đâu không?

Thời gian để xem một bộ phim (một cách dễ dàng) với Facade Pattern

```
public class HomeTheaterTestDrive {
    public static void main(String[] args) {
        // instantiate components here

        HomeTheaterFacade homeTheater =
                new HomeTheaterFacade(amp, tuner, dvd, cd,
                        projector, screen, lights, popper);

        homeTheater.watchMovie("Raiders of the Lost Ark");
        homeTheater.endMovie();
    }
}
```

Here we're creating the components right in the test drive. Normally the client is given a facade, it doesn't have to construct one itself.

First you instantiate the Facade with all the components in the subsystem.

Use the simplified interface to first start the movie up, and then shut it down.

Here's the output.

Calling the Facade's watchMovie() does all this work for us...

```
File Edit Window Help SnakesWhy'dItHaveToBeSnakes?
%java HomeTheaterTestDrive

Get ready to watch a movie...
Popcorn Popper on
Popcorn Popper popping popcorn!
Theater Ceiling Lights dimming to 10%
Theater Screen going down
Top-O-Line Projector on
Top-O-Line Projector in widescreen mode (16x9 aspect ratio)
Top-O-Line Amplifier on
Top-O-Line Amplifier setting DVD player to Top-O-Line DVD Player
Top-O-Line Amplifier surround sound on (5 speakers, 1 subwoofer)
Top-O-Line Amplifier setting volume to 5
Top-O-Line DVD Player on
Top-O-Line DVD Player playing "Raiders of the Lost Ark"
Shutting movie theater down...
Popcorn Popper off
Theater Ceiling Lights on
Theater Screen going up
Top-O-Line Projector off
Top-O-Line Amplifier off
Top-O-Line DVD Player stopped "Raiders of the Lost Ark"
Top-O-Line DVD Player eject
Top-O-Line DVD Player off
%
```

...and here, we're done watching the movie, so calling endMovie() turns everything off.

Định nghĩa Facade Pattern

Để sử dụng Facade, chúng tôi tạo một lớp đơn giản hóa và thống nhất một tập hợp các lớp phức tạp hơn thuộc về một số hệ thống con. Không giống như rất nhiều mẫu khác, Facade khá đơn giản; không cần suy nghĩ về sự trừu tượng để có được nó trong đầu của bạn. Nhưng điều đó không làm cho nó trở nên kém mạnh mẽ: Mẫu Facade cho phép chúng ta tránh sự kết hợp chặt chẽ (tight coupling) (nhớ lại loose coupling – kết nối lỏng lẻo) giữa các client và các hệ thống con, và, như bạn sẽ thấy sau đây, cũng giúp chúng ta tuân thủ một nguyên tắc hướng đối tượng mới.

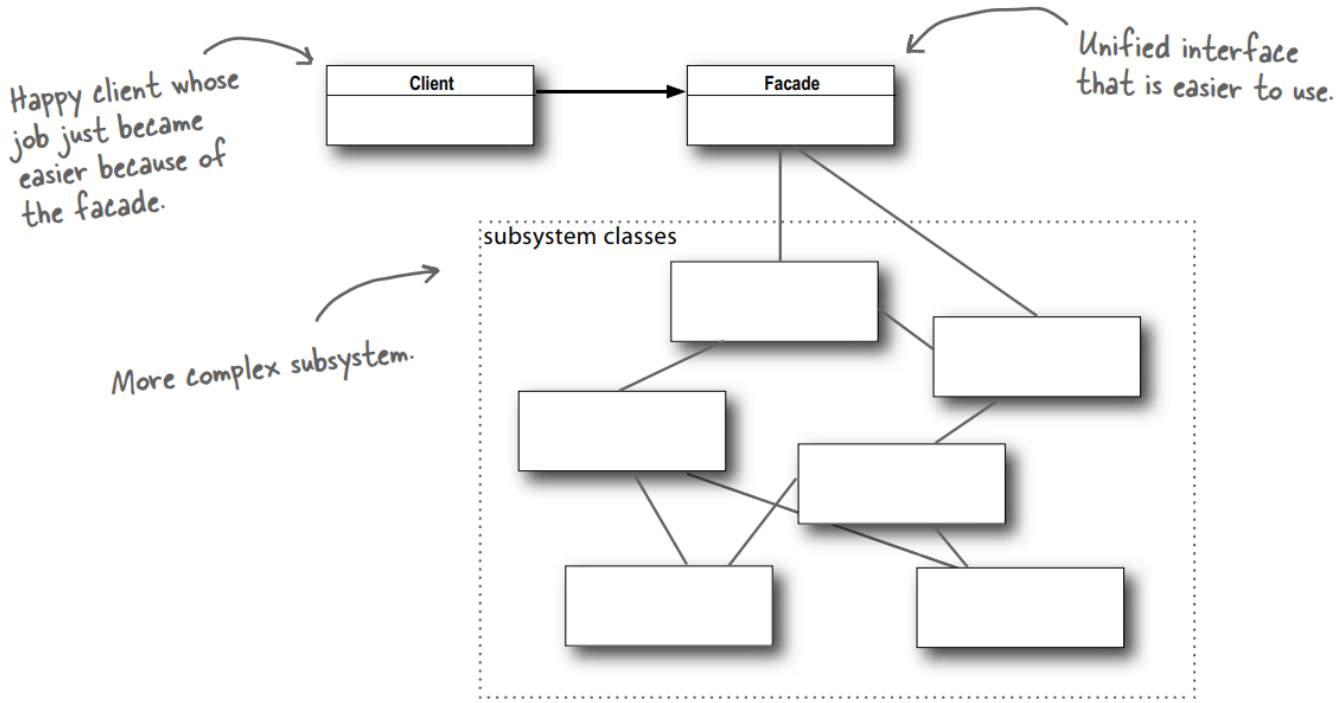
Trước khi chúng tôi giới thiệu nguyên tắc mới đó, chúng ta hãy xem định nghĩa chính thức của mẫu:

> **The Facade Pattern** provides a unified interface to a set of interfaces in a subsytem. Facade defines a higher-level interface that makes the subsystem easier to use.

(Mẫu Facade cung cấp một giao diện hợp nhất cho một tập hợp các giao diện trong một hệ thống con. Facade định nghĩa một giao diện cấp cao hơn giúp hệ thống con dễ sử dụng hơn)

Ở đây không có nhiều thứ mà bạn chưa biết, nhưng một trong những điều quan trọng nhất cần nhớ về một mẫu là ý định của nó. Định nghĩa này cho chúng ta biết nhiều và rõ ràng rằng mục đích của Facade để làm cho một hệ thống con dễ sử dụng hơn thông qua một giao diện đơn giản hóa.

Bạn có thể thấy điều này trong sơ đồ lớp của mẫu:



Chỉ nhiêu đó thôi; bạn đã biết được có một mẫu khác! Bây giờ, đã đến lúc cho nguyên tắc OO mới đó. Xem ra, điều này có thể thách thức một số giả định!

Nguyên tắc biết càng ít càng tốt (The Principle of Least Knowledge)

Nguyên tắc biết càng ít càng tốt hướng dẫn chúng ta giảm bớt sự tương tác giữa các đối tượng với một vài "người bạn thân" của chúng.

Nguyên tắc thường được nêu là:

(Biết càng ít càng tốt – Chỉ giao tiếp với "người bạn thân" của bạn)

Nhưng điều này có nghĩa gì trong thực tế? Điều đó có nghĩa là khi bạn đang thiết kế một hệ thống, cho bất kỳ đối tượng nào, hãy cẩn thận với số lượng các lớp mà nó tương tác và cũng như cách nó tương tác với các lớp đó.

Nguyên tắc này ngăn chúng ta tạo ra các thiết kế có số lượng lớn các lớp được ghép với nhau, khi đó thay đổi trong một phần của tầng hệ thống sẽ ảnh hưởng sang các phần khác. Khi bạn xây dựng nhiều sự phụ thuộc giữa nhiều lớp, bạn đang xây dựng một hệ thống mỏng manh, sẽ tốn kém để duy trì và phức tạp để người khác hiểu.

Sử dụng sức mạnh bộ não

Đoạn code này kết hợp bao nhiêu lớp?

```
1  public float getTemp() {
2          return station.getThermometer().getTemperature();
3  }
```

Làm thế nào để không phụ thuộc "người bạn thân"?

Được rồi, nhưng làm thế nào để bạn không phụ thuộc? Nguyên tắc cung cấp một số hướng dẫn: lấy bất kỳ object; bây giờ từ bất kỳ method nào trong object đó, nguyên tắc cho chúng ta biết rằng chúng ta chỉ nên gọi các phương thức thuộc về:

- Bản thân object (được tạo trong chính object).
- Các đối tượng được truyền vào dưới dạng tham số cho phương thức.
- Bất kỳ đối tượng nào phương thức tạo hoặc khởi tạo.
- Bất kỳ "thành phần" nào của đối tượng (Hãy nghĩ về một "thành phần" như bất kỳ đối tượng nào được tham chiếu bởi một biến instance. Nói cách khác, hãy nghĩ về điều này như một mối quan hệ HAS-A)

(Điều này hướng dẫn cho chúng ta không gọi các phương thức trên các đối tượng được return từ việc gọi các phương thức khác! (xem lại đoạn code bên trên phần "***Sử dụng sức mạnh bộ não***"))

Điều này nghe có vẻ nghiêm ngặt phải không? Tác hại của việc gọi phương thức của một object được return từ một method khác là gì? Chà, nếu chúng ta làm điều đó, thì chúng ta sẽ đưa ra yêu cầu của một đối tượng phụ khác, và tăng số lượng đối tượng chúng ta biết trực tiếp (object được gọi từ một object khác thì không được biết trực tiếp).

Trong những trường hợp như vậy, nguyên tắc buộc chúng ta yêu cầu đối tượng đó tạo ra yêu cầu dùm chúng ta (tạo method cho object chúng ta biết trực tiếp và thực hiện request tới object phụ, thay vì đoạn code ở phần "***Sử dụng sức mạnh bộ não***") ; theo cách đó, chúng ta không thể biết về các đối tượng thành phần của nó (và chúng tôi giữ cho "nhóm bạn bè" của chúng tôi nhỏ hơn). Ví dụ:

Without the Principle

```
public float getTemp() {
    Thermometer thermometer = station.getThermometer();
    return thermometer.getTemperature();
}
```

Here we get the thermometer object from the station and then call the getTemperature() method ourselves.

With the Principle

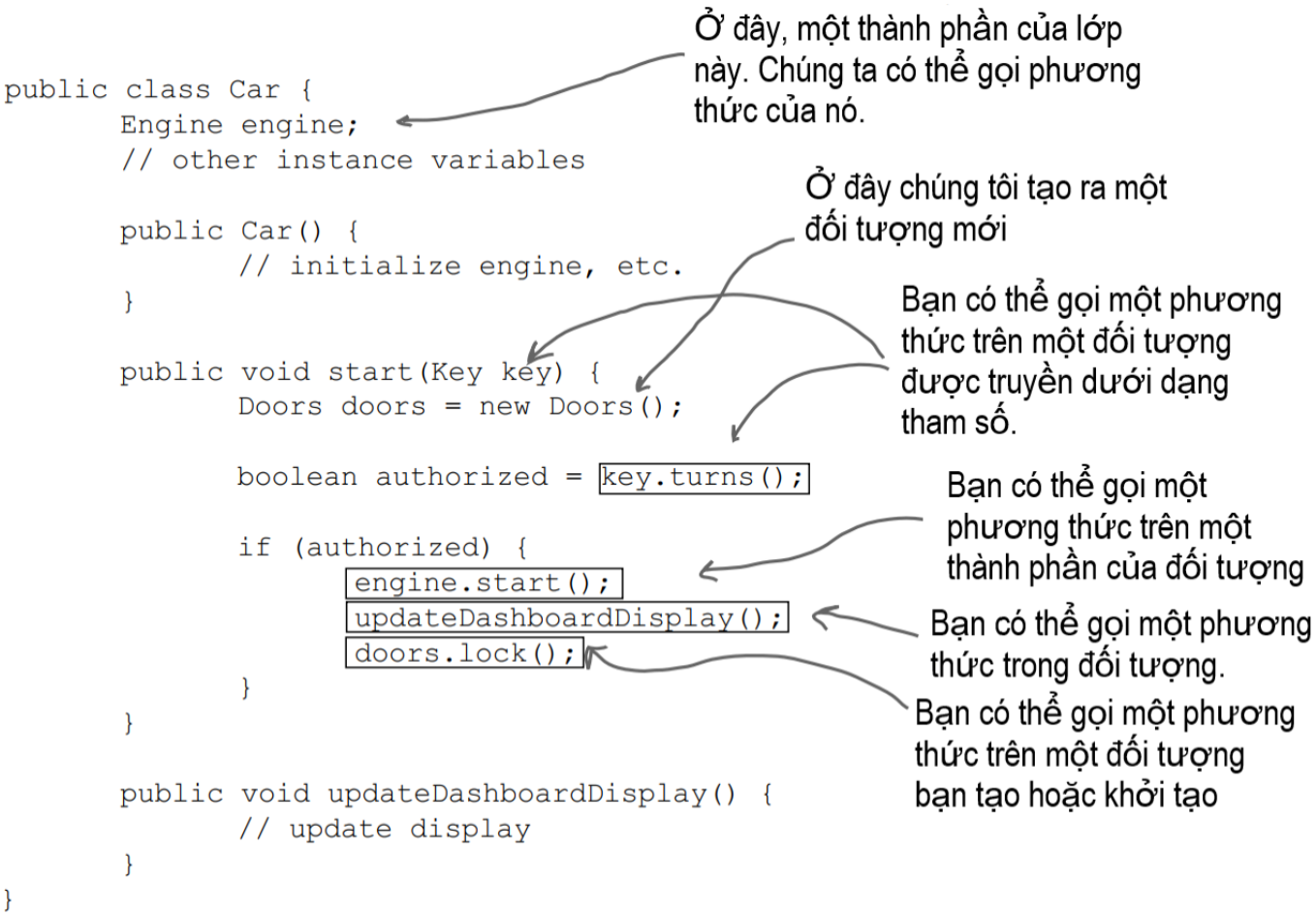
```
public float getTemp() {
    return station.getTemperature();
}
```

When we apply the principle, we add a method to the Station class that makes the request to the thermometer for us. This reduces the number of classes we're dependent on.

(Sự khác biệt giữa hai phương thức là Phương thức 1: muốn có temperature thì phải thông qua instance **Thermometer**, trong khi Phương thức 2: tạo 1 phương thức **getTemperature** trực tiếp (xử lí tất cả code gọi bên trong này))

Giữ các cuộc gọi phương thức của bạn trong giới hạn...

Ở đây, một lớp Car thể hiện tất cả các cách bạn có thể gọi các phương thức và vẫn tuân thủ Nguyên tắc biết càng ít càng tốt:



Không có câu hỏi ngớ ngẩn

Hỏi: Có một nguyên tắc khác gọi là Law of Demeter; chúng liên quan như thế nào?

Trả lời: Hai nguyên tắc đó là một và giống nhau và bạn sẽ gặp các thuật ngữ này được trộn lẫn với nhau. Chúng tôi thích sử dụng Principle of Least Knowledge vì một vài lý do: (1) tên này trực quan hơn và (2) việc sử dụng từ "Law (Luật)" ngụ ý chúng tôi luôn phải áp dụng nguyên tắc này. Trong thực tế, không có nguyên tắc nào là "Luật", tất cả các nguyên tắc nên được sử dụng khi nào và ở đâu chúng hữu ích. Tất cả các thiết kế liên quan đến sự đánh đổi (trừu tượng so với tốc độ, không gian so với thời gian, v.v.) và trong khi các nguyên tắc cung cấp hướng dẫn, tất cả các yếu tố cần được tính đến trước khi áp dụng chúng.

Hỏi: Có bất kỳ nhược điểm nào khi áp dụng Nguyên tắc biết càng ít càng tốt không?

Trả lời: Có; Mặc dù nguyên tắc này làm giảm sự phụ thuộc giữa các đối tượng và các nghiên cứu đã cho thấy điều này làm giảm độ phức tạp khi bảo trì phần mềm, nhưng cũng có trường hợp áp dụng nguyên tắc này dẫn đến: lớp A gọi lớp B, lớp B lại có phương thức gọi lớp C. Điều này có thể dẫn đến tăng độ phức tạp và thời gian development cũng như giảm hiệu năng trong runtime.

Một trong hai lớp này vi phạm Nguyên tắc biết càng ít càng tốt đúng không? Tại sao hay tại sao không?

```
public House {
    WeatherStation station;

    // other methods and constructor

    public float getTemp() {
         return station.getThermometer().getTemperature();
    }
}
```

```
public House {
     WeatherStation station;

    // other methods and constructor

    public float getTemp() {
         Thermometer thermometer = station.getThermometer();
         return getTempHelper(thermometer);
    }

    public float getTempHelper(Thermometer thermometer) {
        return thermometer.getTemperature();
    }
}
```

Đáp án:

```
public House {
    WeatherStation station;

    // other methods and constructor

    public float getTemp() {
         return station.getThermometer().getTemperature();
    }
}
```

Violates the Principle of Least Knowledge! You are calling the method of an object returned from another call.

```
public House {
     WeatherStation station;

    // other methods and constructor

    public float getTemp() {
         Thermometer thermometer = station.getThermometer();
         return getTempHelper(thermometer);
    }

    public float getTempHelper(Thermometer thermometer) {
        return thermometer.getTemperature();
    }
}
```

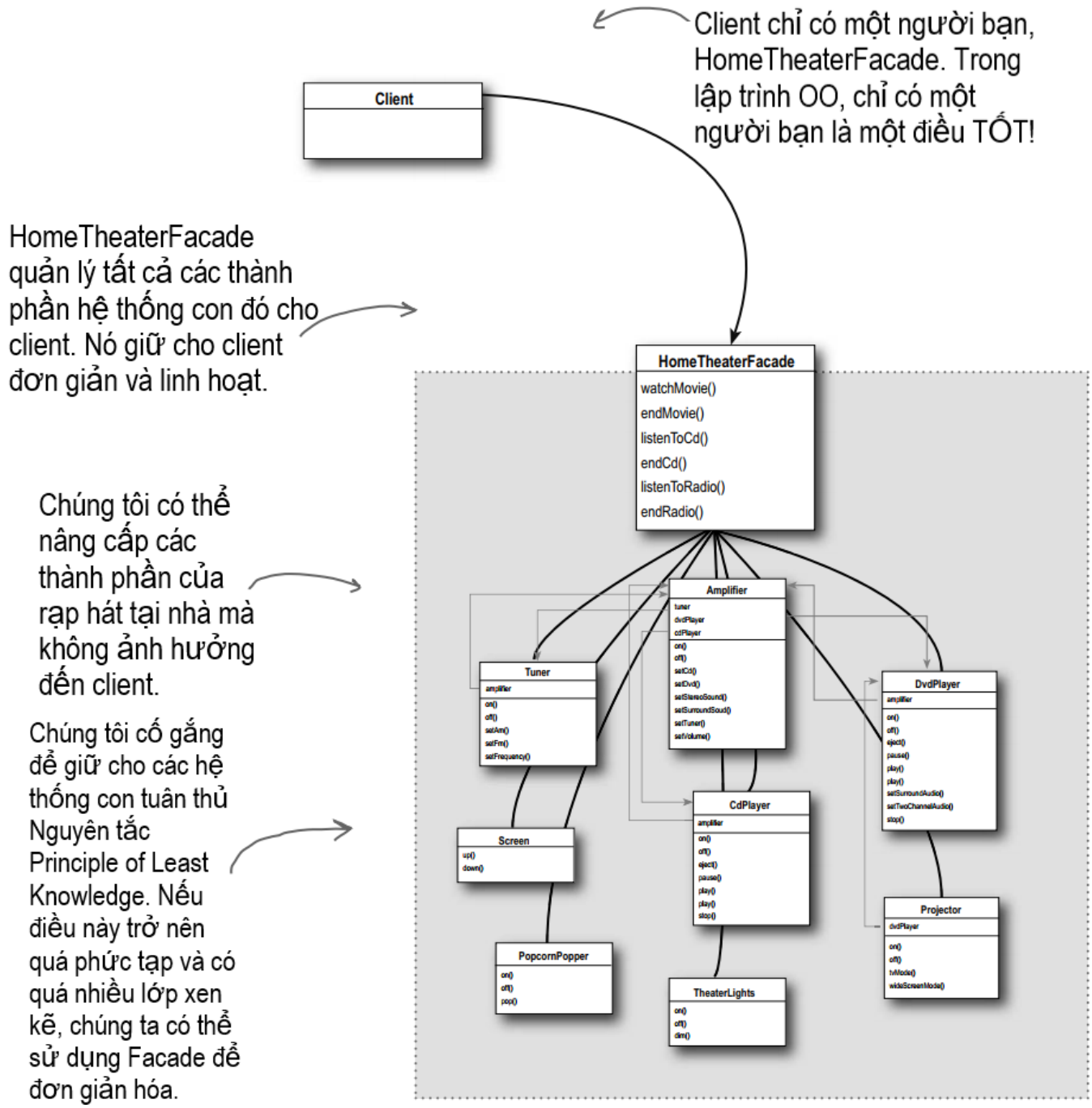
Doesn't violate Principle of Least Knowledge! This seems like hacking our way around the principle. Has anything really changed since we just moved out the call to another method?

Sử dụng sức mạnh bộ não

Bạn có thể nghĩ về việc sử dụng phổ biến Java vi phạm Nguyên tắc biết càng ít càng tốt không? Bạn có nên quan tâm?

Trả lời: Suy nghĩ về **System.out.println()**?

Facade Pattern và The Principle of Least Knowledge



Tóm tắt

- Khi bạn cần sử dụng một lớp hiện có và interface của nó không phải là thứ bạn cần, hãy sử dụng một Adapter (https://toihocdesignpattern.com/chuong-7-adapter-va-facade-pattern-tro-nen-thich-nghi-phan-1.html).
- Khi bạn cần đơn giản hóa và thống nhất một giao diện lớn hoặc bộ giao diện phức tạp, hãy sử dụng Facade.

- Một Adapter (https://toihocdesignpattern.com/chuong-7-adapter-va-facade-pattern-tro-nen-thich-nghi-phan-1.html) thay đổi một interface thành một interface khác mà client mong đợi.
- Một Facade tách rời một client khỏi một hệ thống con phức tạp.
- Việc thực hiện một Adapter (https://toihocdesignpattern.com/chuong-7-adapter-va-facade-pattern-tro-nen-thich-nghi-phan-1.html) có thể đòi hỏi ít công việc hoặc rất nhiều công việc tùy thuộc vào kích thước và độ phức tạp của target interface.
- Việc thực hiện một Facade đòi hỏi chúng ta phải kết hợp facade với hệ thống con của nó và sử dụng ủy quyền để thực hiện công việc của facade.
- Có hai dạng của Mẫu Adapter (https://toihocdesignpattern.com/chuong-7-adapter-va-facade-pattern-tro-nen-thich-nghi-phan-1.html): object adapter và class adapter. Class Adapter yêu cầu đa kế thừa.
- Bạn có thể thực hiện nhiều Facade cho một hệ thống con.
- Một Adapter (https://toihocdesignpattern.com/chuong-7-adapter-va-facade-pattern-tro-nen-thich-nghi-phan-1.html) bao bọc một đối tượng để thay đổi giao diện của nó, một Decorator (https://toihocdesignpattern.com/chuong-3-head-first-design-patterns-tieng-viet-decorator-pattern-doi-tuong-trang-tri.html) bao bọc một đối tượng để thêm các hành vi và trách nhiệm mới, và một Facade bao bọc một bộ các đối tượng để đơn giản hóa chúng.