

**TRƯỜNG ĐẠI HỌC GIAO THÔNG VẬN TẢI
KHOA CÔNG NGHỆ THÔNG TIN
BỘ MÔN KHOA HỌC MÁY TÍNH
TRẦN VĂN DŨNG**

BÀI GIẢNG

ĐẶC TẢ PHẨM MỀM

SỐ TÍN CHỈ: 3

Hà nội 2020

MỞ ĐẦU

Môn học “Đặc tả phần mềm” đã được đưa vào giảng dạy tại Khoa Công nghệ Thông tin của trường đại học Giao thông vận tải từ năm 2015 cho ngành Công nghệ thông tin và được giảng dạy cho các sinh viên năm cuối. Mục đích môn học nhằm trang bị các kỹ năng đọc hiểu và có khả năng viết các yêu cầu, đồng thời bước đầu biết cách thiết kế, tách các thành phần cho các phần mềm đơn giản. Với mục tiêu lựa chọn nội dung phù hợp với với khả năng của sinh viên và nhu cầu thực tế của các doanh nghiệp, nội dung môn học bao gồm hai phần Đặc tả và mẫu thiết kế dựa trên đặc tả mang tính thực hành cao. Phần đặc tả dựa trên mô hình dữ liệu và phụ thuộc module là cách tiếp cận trên xuống, phi hình thức nhưng có tính chính xác, hiệu quả và kiểm soát được các mục tiêu yêu cầu. Phần mẫu thiết kế dựa trên đặc tả yêu cầu chọn lọc 23 mẫu cơ bản của mẫu thiết kế trong Java, đặc thù cho xây dựng phần mềm hướng đối tượng. Một mục đích nữa mà môn học muốn hướng tới là tăng cường khả năng phân tích, đọc viết, tư duy phát triển phần mềm theo hướng đối tượng và dựa cấu phần cho các sinh viên.

Tài liệu môn học được dựa trên hai cuốn sách của các giáo sư có nhiều kinh nghiệm. Thứ nhất đó là cuốn “Program Development in Java: Abstraction, Specification and Object-Oriented Design” của các tác giả Barbara Liskov with John Guttag, Addison Wesley, Pearson Education, 2002. Nội dung thiết kế dựa trên đặc tả được tham khảo từ cuốn sách “Java Design Patterns – Reusable Solutions to Common Problems” của Rohit Joshi, Exelixis Media P.C., 2015. Cuốn sách thứ hai có kèm theo các chương trình mẫu để minh họa. Các sinh viên cố gắng đọc hiểu các chương trình mẫu và dựa trên đó xây dựng các chương trình ứng dụng nhỏ của mình để thấu hiểu các lợi ích đem lại của phát triển phần mềm hướng đối tượng.

Giáo trình gồm 6 chương. Chương đầu nêu tổng quan về Đặc tả và các tiêu chuẩn của một đặc tả tốt, cùng với các đặc tả trừu tượng dữ liệu, bộ duyệt và giao diện; Chương 2 nêu cách viết đặc tả yêu cầu phần mềm và các mô hình hỗ trợ; Chương 3 trình bày những khái niệm cơ bản về đặc tả thiết kế với ví dụ điển hình về máy tìm kiếm [1]; Chương 4 tiếp

tục xem xét đánh giá thiết kế, hoàn thiện thiết kế và phân tích so sánh hai cách tiếp cận trên xuống, dưới lên và kết hợp của chúng; Chương 5 giới thiệu về đặc tả hình thức với ví dụ hệ thống đa thang máy dựa trên khái niệm tập hợp và quan hệ [3]; Chương 6 đưa ra 23 mẫu thiết kế thường dùng trong Java và các chương trình minh họa đi kèm [2].

Do là lần đầu biên soạn và chưa có nhiều kinh nghiệm thực tế, nên không tránh khỏi những sai sót và lỗi in ấn nhất định. Tác giả xin vui lòng tiếp nhận mọi sự đóng góp giúp cho giáo trình “Đặc tả phần mềm” ngày càng tốt hơn. Mọi ý kiến xây dựng xin gửi về theo địa chỉ sau: Trần Văn Dũng, Bộ môn Khoa học máy tính, Khoa Công nghệ Thông tin, Đại học Giao thông Vận tải, Láng Thượng, Đống đà, Hà nội.

Trần Văn Dũng

MỤC LỤC

Mở đầu	2
Chương 1. Đặc tả	2
1.1 Đặc tả trừu tượng thủ tục	10
1.2 Đặc tả trừu tượng dữ liệu	18
1.3 Đặc tả trừu tượng bộ duyệt	23
1.4 Đặc tả trừu tượng dữ liệu đa hình	33
1.5 Đặc tả và các tập thỏa đặc tả	35
1.6 Một số tiêu chuẩn cho đặc tả	36
Chương 2. Đặc tả yêu cầu	51
2.1 Mô hình dữ liệu	51
2.2 Đặc tả yêu cầu	60
2.3 Kiểm thử	64
2.4 Đặc tả yêu cầu cho Search Engine	75
Tóm tắt chương	82
Chương 3. Đặc tả thiết kế	84
3.1 Tổng quan về quá trình thiết kế	84
3.2 Sổ tay thiết kế	87
3.3 Cấu trúc các chương trình tương tác	93
3.4 Bắt đầu thiết kế	96
3.5 Phương pháp thiết kế	102
3.6 Tiếp tục thiết kế	104
3.7 Lược đồ phụ thuộc module với mô hình dữ liệu	116
3.8 Đặc tả chương trình	118
3.9 Đánh giá phương pháp thiết kế	120

Chương 4. Đánh giá thiết kế	124
4.1 Phản biện thiết kế	124
4.2 Thứ tự tiến trình phát triển chương trình	136
4.3 Tóm tắt	141
Chương 5. Đặc tả hình thức	143
5.1 Quan hệ	143
5.2 Phương pháp hình thức thiết kế phần mềm	147
5.3 Đặc tả máy bán hàng	149
5.4 Đặc tả Hệ thống đa thang máy	159
Chương 6. Mẫu thiết kế	166
6.1 Mở đầu	166
6.2 Mẫu thiết kế Adapter	170
6.3 Mẫu thiết kế Façade	178
6.4 Mẫu thiết kế Composite	183
6.5 Mẫu thiết kế Bridge	190
6.6 Mẫu thiết kế Singleton	198
6.7 Mẫu thiết kế Observer	203
6.8 Mẫu thiết kế Mediator	214
6.9 Mẫu thiết kế Proxy	220
6.10 Mẫu thiết kế Chain of Responsibility	239
6.11 Mẫu thiết kế Flighweight	248
6.12 Mẫu thiết kế Builder	254
6.13 Mẫu thiết kế Factory Method	261
6.14 Mẫu thiết kế Abstract Factory	277
6.15 Mẫu thiết kế Prototype	284

6.16 Mẫu thiết kế Memento	293
6.17 Mẫu thiết kế Template	298
6.18 Mẫu thiết kế State	304
6.19 Mẫu thiết kế Strategy	315
6.20 Mẫu thiết kế Command	318
6.21 Mẫu thiết kế Interpreter	328
6.22 Mẫu thiết kế Decorator	335
6.23 Mẫu thiết kế Iterator	345
6.24 Mẫu thiết kế Visitor	350
Tài liệu tham khảo	358

CHƯƠNG 1: ĐẶC TẢ

Trong tài liệu này chúng ta sẽ nhấn mạnh tầm quan trọng của đặc tả trong tất cả các bước của quá trình phát triển phần mềm. Trong chương này chúng ta thảo luận ý nghĩa của đặc tả và một số tiêu chuẩn để viết đặc tả. Chúng ta cũng sẽ bàn hai cách sử dụng chính của đặc tả cho người sử dụng phần mềm và cho người phát triển phần mềm.

Các kiểu trừu tượng:

- Trừu tượng thủ tục cho phép chúng ta đưa ra các thao tác mới
- Trùu tượng dữ liệu cho phép chúng ta đưa ra kiểu mới của các đối tượng dữ liệu
- Trùu tượng bộ duyệt cho phép chúng ta duyệt trên các phần tử của một họ mà không biết chi tiết các phần tử đó nhận được như thế nào
- Phân cấp kiểu cho phép chúng ta khai quát từ các kiểu dữ liệu riêng biệt đến họ các kiểu liên quan nhau

Trước khi đi vào các định nghĩa hình thức về đặc tả, chúng ta có thể hiểu, đặc tả một chương trình là mô tả chính xác hành vi của chương trình đó.

1.1 Đặc tả trừu tượng thủ tục

Một trừu tượng là một ánh xạ nhiều một. Nó khai quát từ thực thể các chi tiết không liên quan, mà mô tả chỉ các chi tiết liên quan đến vấn đề cần giải quyết.

Có trừu tượng dựa trên tham số và trừu tượng dựa trên đặc tả.

Trong trừu tượng dựa trên tham số, chúng ta khai quát từ thực thể dữ liệu được sử dụng. Trùu tượng được xác định theo thuật ngữ các tham số hình thức, dữ liệu thực tế được gắn kết đến các hình thức này khi trừu tượng được sử dụng. Như vậy thực thể dữ liệu thực tế là không liên quan, nhưng sự có mặt, số lượng, và kiểu của các dữ liệu thực tế là liên quan. Ưu điểm của việc khai quát này là chúng giảm lượng code mà cần thiết phải viết, sửa và bảo trì.

Trong trừu tượng bởi đặc tả, chúng ta tập trung vào hành vi, mà người dùng có thể dựa trên đó và khai quát khỏi các chi tiết cài đặt hành vi đó. Như vậy hành vi nào là cái liên quan và phương pháp tạo ra hành vi đó là cái không liên quan. Như vậy trừu tượng thủ tục được đặc tả là mô tả giá trị đầu ra cần thỏa mãn điều kiện gì với đầu vào cho trước là hành vi cần quan tâm, còn việc tính toán thế nào như thế nào để được giá trị đầu ra đó là không liên quan.

Cơ chế trừu tượng:

- Trùu tượng bằng tham số khái quát từ thực thể dữ liệu bằng cách thay chúng với các tham số. Nó khái quát các module sao cho chúng có thể có thể được sử dụng trong nhiều hơn các tình huống.
- Trùu tượng bằng đặc tả trùu tượng khỏi chi tiết cài đặt qua hành vi mà người sử dụng có thể dựa vào đó. Nó tách module khỏi các cài đặt khác nhau.

Ví dụ trùu tượng tham số: bằng cách đưa vào tham số chúng ta có thể biểu diễn tập vô hạn các tính toán khác nhau bằng một chương trình mà khái quát lên từ chúng. Chẳng hạn

```
int squares (int x, y)
{
    return (x*x + y*y)
}
```

và ta trói buộc các giá trị đầu vào thực tế cho các tham số và tính toán theo thân thủ tục bằng lời gọi hàm

```
u = squares (w, z);
```

Trùu tượng bằng tham số là cơ chế rất mạnh, nó không chỉ cho phép chúng ta mô tả số phép toán vô hạn một cách đơn giản, mà còn rất hiệu quả hiện thực trong các ngôn ngữ lập trình. Tuy nhiên, nó là cơ chế không đủ mạnh để mô tả thuận tiện và đầy đủ một trùu tượng như việc sử dụng cẩn thận một thủ tục có thể cung cấp.

Trùu tượng thủ tục bằng đặc tả đem lại các lợi ích sau:

- Tính địa phương: cài đặt một trùu tượng thủ tục có thể được đọc hoặc được viết mà không cần kiểm tra cài đặt của bất cứ trùu tượng nào khác
- Tính thành phần: một trùu tượng có thể được tái cài đặt không yêu cầu thay đổi đến bất cứ trùu tượng nào khác sử dụng nó.

Để đạt được hai lợi ích trên, một trùu tượng thủ tục phải được định nghĩa chính xác. Chúng ta sẽ định nghĩa trùu tượng bằng đặc tả, mà được viết trong một ngôn ngữ đặc tả mà là hình thức hoặc không hình thức. Ưu điểm của đặc tả hình thức là chúng có ngữ nghĩa chính xác. Tuy nhiên trước hết chúng ta sẽ sử dụng đặc tả không hình thức, ở đó hành vi của các trùu tượng được mô tả bằng ngôn ngữ tự nhiên. Nhược điểm của đặc tả không hình thức là rất khó cho chúng một ngữ nghĩa chính xác, vì đặc tả không hình thức là không chính xác.

Đặc tả khác với lập trình, vì nó tập trung vào mô tả trùu tượng là gì, chứ không phải cách cài đặt chúng.

```
return_type pname (...)  
// REQUIRES: This clause states any constraints on use  
// MODIFIES: This clause identifies all modified inputs  
// EFFECTS: This clause defines the behavior
```

Hình 1.1. Đặc tả một thủ tục

Ví dụ

Trùu tượng tham số:

The `sqrt` procedure

```
float sqrt (float coef) {  
    // REQUIRES: coef > 0  
    // EFFECTS: Returns an approximation to the square root of coef  
    float ans = coef/2.0;  
    int i = 1;  
    while (i < 7) {  
        ans = ans - ((ans * ans - coef)/(2.0*ans));  
        i = i + 1;  
    }  
    return ans;  
}
```

Trùu tượng đặc tả:

Một thủ tục khai căn `sqrt` có tiêu đề:

float sqrt (float x)

được hiểu về mặt toán học là một hàm

sqrt: real -> real

có thể được đặc tả như sau:

```
static float sqrt (float x, float epsilon)  
// REQUIRES: x >= 0 && .00001 < epsilon < .001  
// EFFECTS: Returns sq such that x - epsilon <= sq*sq <= x + epsilon.
```

Hình 1.2. Đặc tả cho thủ tục khai căn bậc hai `sqrt`.

Ở đây, mệnh đề REQUIRES khẳng định điều kiện ràng buộc mà ở đó trừu tượng được định nghĩa. Điều kiện này là cần thiết, nếu thủ tục là bộ phận, tức là không xác định đối với một số đầu vào. Mệnh đề này được bỏ qua, nếu thủ tục là tổng thể, tức là xác định với mọi giá trị đầu vào trên tập đã cho.

Mệnh đề MODIFIES liệt kê tên của các đầu vào mà bị thay đổi bởi thủ tục. Khi một số đầu vào bị thay đổi, chúng ta nói rằng thủ tục có tác động phụ. Mệnh đề này được bỏ qua nếu không có đầu vào nào bị thay đổi.

Cuối cùng mệnh đề EFFECTS mô tả hành vi của thủ tục cho mọi đầu vào thỏa mãn điều kiện của mệnh đề REQUIRES. Nó cần định nghĩa đầu ra nào được tạo và những thay đổi gì trên đầu vào được liệt kê ở mệnh đề MODIFIES. Nó không nói gì về hành vi của thủ tục khi mệnh đề REQUIRES không thỏa mãn.

Trong Ví dụ sau, chúng ta đặc tả một phần lớp Arrays, ở đây chúng ta cung cấp đặc tả cho một số phương thức tĩnh đứng riêng của lớp đó.

Trong đặc tả đó, chúng ta thấy *search* và *searchSort* không thay đổi đầu vào của chúng, nhưng *sort* thay đổi đầu vào, như được chỉ ra trong mệnh đề MODIFIES.

Lưu ý rằng *sort* và *search* là tổng thể, vì đặc tả của chúng không chứa mệnh đề REQUIRES. Tuy nhiên *searchSort* là bộ phận, vì nó đòi hỏi mảng đối số là đã được sắp xếp. Và cũng thấy rằng, mệnh đề EFFECTS không khẳng định gì, nếu đầu vào không thỏa mãn điều kiện ràng buộc. Trong trường hợp này, người cài đặt cần phải làm gì đó là điều đặc tả chưa đề cập đến, nghĩa là thủ tục chưa xác định đầy đủ.

Đặc tả thủ tục đứng riêng trong lớp *public* và thủ tục là phương thức *public static*:

```
public class Arrays {  
    // OVERVIEW: This class provides a number of standalone procedures that  
    //   are useful for manipulating arrays of ints.  
  
    public static int search (int[ ] a, int x)  
        // EFFECTS: If x is in a, returns an index where x is stored;  
        //   otherwise, returns -1.  
  
    public static int searchSorted (int[ ] a, int x)  
        // REQUIRES: a is sorted in ascending order  
        // EFFECTS: If x is in a, returns an index where x is stored;  
        //   otherwise, returns -1.  
  
    public static void sort (int[ ] a)  
        // MODIFIES: a  
        // EFFECTS: Rearranges the elements of a into ascending order  
        //   e.g., if a = [3, 1, 6, 1] before the call, on return a = [1, 1, 3, 6].  
}
```

Trong thủ tục *sort* có mệnh đề *MODIFIES* vì nó thay đổi *a* bằng cách sắp xếp lại *a*. Và *sort* là tổng thể.

Thủ tục tổng thể và thủ tục bộ phận:

- Một thủ tục là tổng thể, nếu hành vi của nó được đặc tả cho mọi đầu vào hợp lệ; ngược lại nó là bộ phận. Đặc tả của thủ tục bộ phận luôn chứa mệnh đề *REQUIRES*.
- Các thủ tục bộ phận là kém an toàn hơn tổng thể. Do đó chúng cần được sử dụng chỉ trong bối cảnh hạn chế như là hiệu năng tốt hơn.
- Khi có thể, cài đặt cần phải kiểm tra ràng buộc trong mệnh đề *REQUIRES* và đưa ra thông báo lỗi trong trường hợp chúng không thỏa mãn.

Dưới đây là đặc tả và cài đặt của một số thủ tục đứng riêng

```
public class Arrays {  
    // OVERVIEW: This class provides a number of standalone procedures that  
    // are useful for manipulating arrays of ints.  
  
    public static int searchSorted (int[ ] a, int x) {  
        // REQUIRES: a is sorted in ascending order.  
        // EFFECTS: If x is in a, returns an index where x is stored;  
        // otherwise, returns -1.  
        // uses linear search  
        if (a == null) return -1;  
        for (int i = 0; i < a.length; i++)  
            if (a[i] == x) return i; else if (a[i] > x) return -1;  
        return -1;  
    }  
  
    // other static methods go here  
}
```

Một cài đặt của thủ tục searchSort

Các tính chất của thủ tục và cài đặt của chúng:

- *Tính tối thiểu: một đặc tả là tối thiểu hơn một cái khác, nếu nó chứa ít ràng buộc hơn cho hành vi cho phép.*
- *Hành vi chưa xác định đầy đủ: một thủ tục là chưa xác định đầy đủ, nếu với một số đầu vào đặc tả của nó cho phép nhiều hơn một kết quả có thể.*
- *Một cài đặt tất định: một cài đặt của một thủ tục là tất định, nếu với cùng một đầu vào, nó luôn cho ra cùng một kết quả. Các cài đặt của một thủ tục chưa xác định đầy đủ luôn luôn như là tất định.*
- *Tính tổng quan: một cài đặt là tổng quan hơn cái khác nếu nó xử lý lớp đầu vào lớn hơn.*

Một cài đặt của *sort* được chỉ ra trong Hình vẽ sau. Lưu ý các phương thức quicksort và partition không khai báo là *public*, mà chỉ cần lớp *Array* và phương thức *sort*.

Một Ví dụ tiếp theo là cài đặt phương thức loại bỏ phần tử lặp trong Vector *removeDups*.

```
public class Arrays {  
    // OVERVIEW: ...  
  
    public static void sort (int[ ] a) {  
        // MODIFIES: a  
        // EFFECTS: Sorts a[0], ..., a[a.length - 1] into ascending order.  
        if (a == null) return;  
        quickSort(a, 0, a.length-1); }  
  
    private static void quickSort(int[ ] a, int low, int high) {  
        // REQUIRES: a is not null and 0 <= low & high < a.length  
        // MODIFIES: a  
        // EFFECTS: Sorts a[low], a[low+1], ..., a[high] into ascending order.  
        if (low >= high) return;  
        int mid = partition(a, low, high);  
        quickSort(a, low, mid);  
        quickSort(a, mid + 1, high); }  
  
    private static int partition(int[ ] a, int i, int j) {  
        // REQUIRES: a is not null and 0 <= i < j < a.length  
        // MODIFIES: a  
        // EFFECTS: Reorders the elements in a into two contiguous groups,  
        // a[i], ..., a[res] and a[res+1], ..., a[j], such that each  
        // element in the second group is at least as large as each  
        // element of the first group. Returns res.  
        int x = a[i];  
        while (true) {  
            while (a[j] > x) j--;  
            while (a[i] < x) i++;  
            if (i < j) { // need to swap  
                int temp = a[i]; a[i] = a[j]; a[j] = temp;  
                j--; i++; }  
            else return j; }  
    }  
}
```

Một cài đặt của thủ tục quickSort

```
public class Vectors {  
    // OVERVIEW: Provides useful standalone procedures for manipulating vectors.  
  
    public static void removeDupls (Vector v) {  
        // REQUIRES: All elements of v are not null.  
        // MODIFIES: v  
        // EFFECTS: Removes all duplicate elements from v; uses equals to  
        // determine duplicates. The order of remaining elements may change.  
        if (v == null) return;  
        for (int i = 0; i < v.size( ); i++) {  
            Object x = v.get(i);  
            int j = i + 1;  
            // remove all dupls of x from the rest of v  
            while (j < v.size( ))  
                if (!x.equals(v.get(j))) j++;  
                else { v.set(j, v.lastElement( ));  
                    v.remove(v.size( )-1); }  
        }  
    }  
}
```

Một cài đặt của thủ tục *removeDupls* trong vector

1.2 Đặc tả trừu tượng dữ liệu

Trùu tượng dữ liệu cho phép chúng ta khái quát chi tiết khỏi việc các đối tượng dữ liệu được cài đặt như thế nào, mà chỉ quan tâm đến hành vi của các đối tượng. Trùu tượng dữ liệu cho phép chúng ta sử dụng như kiểu dữ liệu. Nó được đặc trưng bởi các đối tượng với các thành phần dữ liệu của nó và các thao tác đi kèm.

Cũng như trường hợp đối với thủ tục, ý nghĩa của một kiểu cần không được cho bởi bất cứ cài đặt nào của nó. Thay vào đó một đặc tả cần được định nghĩa cho các hành vi của nó. Vì các đối tượng của một kiểu là được sử dụng chỉ bằng lời gọi các thao tác, hầu hết đặc tả bao gồm giải thích các thao tác đó làm gì.

Trong Java, một kiểu mới được định nghĩa bởi các lớp hoặc các giao diện. Bây giờ ta sẽ xem xét các lớp.

Mỗi một lớp định nghĩa một kiểu bằng định nghĩa một tên cho kiểu, một tập các hàm tạo *constructors*, và một tập các phương thức. Các hàm tạo được sử dụng để khởi tạo các đối tượng mới của kiểu này, chúng là các đối tượng được khởi tạo. Chỉ khi một đối tượng đã được tạo (và khởi đầu bằng hàm tạo), người sử dụng có thể được truy cập đến nó bằng việc gọi các phương thức của nó.

Một dạng của đặc tả trùu tượng dữ liệu được chỉ ra trên Hình 1.3. Phần tên của lớp *dname* chỉ ra rằng một kiểu dữ liệu mới được gọi là *dname* đang được định nghĩa. Phần đầu chứa một khai báo của tính nhìn thấy *visibility* của lớp này; hầu hết mọi lớp có tính nhìn thấy *public* sao cho chúng có thể được sử dụng bởi code ở bên ngoài gói chứa của chúng.

```
visibility class dname {  
    // TỔNG QUAN: Một mô tả ngắn gọn về hành vi của các đối tượng của kiểu này.  
    ...  
    // Các hàm tạo  
    // đặc tả cho các hàm tạo ở đây  
    // Các phương thức  
    // đặc tả cho các phương thức ở đây  
}
```

Hình 1.3. Dạng đặc tả một trùu tượng dữ liệu

Một đặc tả có ba phần. *Tổng quan* cho một mô tả ngắn gọn của một trùu tượng dữ liệu, bao gồm cách xem xét các đối tượng trùu tượng trong thuật ngữ các khái niệm được định nghĩa chính xác. Thông thường dùng một mô hình để thể hiện các đối tượng; tức là, nó mô tả các đối tượng trong thuật ngữ các đối tượng khác mà bạn đọc của đặc tả có thể đã có thể hiểu được. Chẳng hạn, ngắn xếp có thể được định nghĩa trong thuật ngữ các dãy toán học. Phần tổng quan cũng khẳng định các đối tượng của kiểu có tính thay đổi được không, sao cho trạng thái của chúng là có thể thay đổi theo thời gian hoặc không thể thay đổi.

Phần các hàm tạo của đặc tả định nghĩa các hàm tạo mà khởi tạo các đối tượng mới, trong khi phần các phương thức định nghĩa các phương thức mà cho phép truy cập đến thông qua đối tượng khi chúng đã được tạo. Mọi hàm tạo và các phương thức mà xuất hiện trong đặc tả sẽ là *public*.

Các hàm tạo và các phương thức là các thủ tục và chúng được đặc tả như sử dụng ký hiệu đặc tả dùng cho thủ tục, với khác biệt sau:

- Các phương thức và các hàm tạo cả hai đều thuộc về các đối tượng, hơn là về các lớp. Do đó, từ khóa *static* sẽ không xuất hiện ở đầu phương thức
- Đối tượng mà một phương thức hay hàm tạo thuộc về nó là sẵn sàng cho nó như một đối số ẩn, và đối tượng này có thể được tham chiếu đến trong đặc tả phương thức hoặc hàm tạo như là *this*.

Đặc tả trùu tượng dữ liệu có dạng các chú thích trong code. Khi một trùu tượng dữ liệu được nghĩ ra đầu tiên, mọi cái mà tồn tại là đặc tả; hầu hết mọi code trong lớp như thân các

phương thức đang là không có. Về sau, khi trừu tượng dữ liệu được cài đặt, code này mới được bổ sung.

Tính thay đổi và chia sẻ của đối tượng:

- Một đối tượng là thay đổi được, nếu trạng thái của nó có thể thay đổi. Chẳng hạn, các mảng là thay đổi được.
- Một đối tượng là không thay đổi được, nếu trạng thái của nó không bao giờ thay đổi. Chẳng hạn, các xâu là không thay đổi được.
- Một đối tượng được chia sẻ bởi hai biến, nếu nó có thể được truy cập thông qua một trong chúng.
- Nếu một đối tượng thay đổi được, được chia sẻ bởi hai biến, các thay đổi được thực hiện bởi một biến sẽ được nhìn thấy khi đối tượng được sử dụng thông qua biến kia.

Ví dụ đặc tả của IntSet

Hình 1.4 cho đặc tả của trừu tượng dữ liệu IntSet. IntSet là tập không hạn chế của các số nguyên với các thao tác tạo mới đối tượng, kiểm tra rỗng *empty*, và kiểm tra một số nguyên cho trước có thuộc một tập *IntSet* không, và bổ sung *add* hoặc loại bỏ *remove* phần tử. Tổng quan chỉ ra rằng IntSet là thay đổi được. Nó cũng chỉ ra rằng, chúng ta mô hình IntSet theo thuật ngữ tập hợp của toán học. Phần sau của đặc tả, chúng ta đặc tả mỗi thao tác sử dụng mô hình này.

```
public class IntSet {  
    // TỔNG QUAN: IntSets là các tập hợp số nguyên thay đổi được, không giới hạn  
    // Một tập điển hình IntSet là {x1, ..., xn}  
    // Các hàm tạo  
    public IntSet ()  
        // EFFECTS: Khởi tạo this là rỗng  
        // Các phương thức  
        public void insert (int x)  
            // MODIFIES: this  
            // EFFECTS: Bổ sung x vào this, tức là this_sau = this + {x}.  
            public void remove (int x)  
                // MODIFIES: this
```

```
// EFFECTS: Loại bỏ từ this, tức là this_sau = this - {x}.

public boolean isIn (int x)

// EFFECTS: Nếu x ở trong this, trả về true, ngược lại trả về false.

public int size ( )

// EFFECTS: Trả về số lượng các phần tử của this.

public int choose ( ) throws EmptyException

// EFFECTS: Nếu this là rỗng, throws EmptyException,
// ngược lại trả về một phần tử bất kỳ của this.

}
```

Hình 1.4. Đặc tả của trùu tượng dữ liệu IntSet

Hình 1.4 sử dụng ký hiệu tập hợp trong đặc tả các phương thức. Cụ thể, sử dụng + cho phép hợp tập hợp, và - cho phép trừ tập hợp.

Kiểu IntSet có một hàm tạo mà khởi tạo một tập hợp là rỗng; Lưu ý rằng đặc tả tham chiếu đến đối tượng tập hợp mới là this. Vì một hàm tạo luôn thay đổi this (khởi tạo nó), chúng ta không muốn chỉ ra thay đổi này trong mệnh đề *MODIFIES*. Trên thực tế, thay đổi này là không nhìn thấy được cho người sử dụng: họ không được truy cập vào đối tượng hàm tạo cho đến sau khi hàm tạo chạy, và do đó, họ không quan sát thấy sự thay đổi trạng thái.

Một khi đối tượng *IntSet* tồn tại, các phần tử có thể được bổ sung vào nó bằng cách gọi phương thức *insert* của nó, và các phần tử có thể được loại bỏ từ nó bằng cách gọi phương thức *remove* của nó; cũng như vậy đặc tả tham chiếu đến đối tượng là this. Hai phương thức này đều là tạo ra sự thay đổi *mutators*, vì chúng thay đổi trạng thái của đối tượng; các đặc tả của chúng thể hiện rõ ràng rằng, chúng là *mutators*, vì chúng chưa mệnh đề *MODIFIES* khẳng định là this được thay đổi. Lưu ý rằng, đặc tả của *insert* và *remove* sử dụng ký hiệu *this_sau* để chỉ ra giá trị của this sau khi thao tác trả về.

Các phương thức còn lại là các quan sát: chúng trả về thông tin về trạng thái của đối tượng của chúng, nhưng không thay đổi trạng thái. Các quan sát không có mệnh đề *MODIFIES*.

Phương thức chọn *choose()* trả về một đối tượng bất kỳ của *IntSet*; như vậy, nó là không xác định. Nó đưa ra một ngoại lệ tập là rỗng. Ngoại lệ này có thể là không cần kiểm tra, vì người sử dụng có thể gọi phương thức *size* trước khi gọi *choose* để dễ dàng tin tưởng rằng tập hợp là khác rỗng.

Trong đặc tả *IntSet*, chúng ta dựa trên việc bạn đọc đã biết tập hợp toán học là gì; ngược lại, đặc tả đó sẽ là không hiểu được. Nói chung, sự tin cậy dựa trên mô tả phi hình thức là một điểm yếu của đặc tả phi hình thức. Có thể hợp lý hơn là mong đợi bạn đọc hiểu một số khái niệm toán học, như tập hợp, dãy và các số nguyên.

Tuy nhiên không phải mọi kiểu có thể mô tả đẹp theo thuật ngữ của các khái niệm như vậy. Nếu các khái niệm là không phù hợp, chúng ta cần phải mô tả kiểu một cách tốt nhất có thể, ngay cả sử dụng các hình ảnh, nhưng tất nhiên, ở đây luôn có nguy hiểm là bạn đọc không hiểu mô tả hoặc hiểu một cách khác chúng ta mong muốn.

Lưu ý rằng đặc tả có dạng phiên bản sơ bộ của lớp. Code này có thể biên dịch, nếu các phương thức và các hàm tạo được cho thân của chúng là rỗng (ngoại trừ nếu phương thức mà trả về kết quả, sẽ cần kiểu đúng trong kết quả trả về). Điều này cho phép bạn dịch code mà sử dụng trừu tượng này, sao cho bạn có thể nhận được một loạt các lỗi mà chương trình dịch phát hiện được như lỗi kiểu. Bạn không thể chạy sử dụng code này, tuy nhiên sẽ chờ cho đến khi kiểu mới được cài đặt xong.

Ví dụ đặc tả trừu tượng Poly

Một ví dụ đặc tả trừu tượng khác sẽ được xét sau đây. Poly là các đa thức với các hệ số nguyên. Không giống như Inset, Poly là không thay đổi; một khi Poly được tạo, nó không thể bị thay đổi. Các thao tác trên đa thức một biến là hàm tạo, cộng, trừ, và nhân đa thức.

Poly có hai hàm tạo, một là tạo đa thức bằng không, hai là tạo một đơn thức bất kỳ. Poly không có phương thức thay đổi nào, nên không phương thức nào có mệnh đề MODIFIES.

```
public class Poly {  
    // OVERVIEW: Polys are immutable polynomials with integer coefficients.  
    // A typical Poly is  $c_0 + c_1x + \dots$   
  
    // constructors  
    public Poly ( )  
        // EFFECTS: Initializes this to be the zero polynomial.  
  
    public Poly (int c, int n) throws NegativeExponentException  
        // EFFECTS: If  $n < 0$  throws NegativeExponentException else  
        // initializes this to be the Poly  $cx^n$ .  
  
    // methods  
    public int degree ( )  
        // EFFECTS: Returns the degree of this, i.e., the largest exponent  
        // with a non-zero coefficient. Returns 0 if this is the zero Poly.  
  
    public int coeff (int d)  
        // EFFECTS: Returns the coefficient of the term of this whose exponent is d.  
  
    public Poly add (Poly q) throws NullPointerException  
        // EFFECTS: If q is null throws NullPointerException else  
        // returns the Poly this + q.  
  
    public Poly mul (Poly q) throws NullPointerException  
        // EFFECTS: If q is null throws NullPointerException else  
        // returns the Poly this * q.  
  
    public Poly sub (Poly q) throws NullPointerException  
        // EFFECTS: If q is null throws NullPointerException else  
        // returns the Poly this - q.  
  
    public Poly minus ( )  
        // EFFECTS: Returns the Poly - this.  
}
```

Sử dụng trùu tượng dữ liệu

Các chương trình sử dụng trùu tượng dữ liệu chỉ dựa trên đặc tả của chúng:

```
public static Poly diff (Poly p) throws NullPointerException {
    // EFFECTS: If p is null throws NullPointerException
    //   else returns the Poly obtained by differentiating p.
    Poly q = new Poly ( );
    for (int i = 1; i <= p.degree( ); i++)
        q = q.add(new Poly(p.coeff(i)*i, i - 1));
    return q;
}

public static IntSet getElements (int[ ] a)
    throws NullPointerException {
    // EFFECTS: If a is null throws NullPointerException else returns a set
    //   containing an entry for each distinct element of a.
    IntSet s = new IntSet( );
    for (int i = 0; i < a.length; i++) s.insert(a[i]);
    return s;
}
```

Cài đặt trừu tượng dữ liệu

Một lớp bao gồm cả định nghĩa kiểu mới và cung cấp một cài đặt cho nó. Đặc tả bao gồm định nghĩa kiểu dữ liệu. Phần còn lại của lớp là cung cấp cài đặt.

Để cài đặt một trừu tượng dữ liệu, chúng ta phải lựa chọn một *thể hiện* đối với các đối tượng của nó và sau đó cài đặt các hàm tạo để khởi tạo *thể hiện* một cách đúng đắn. Một thể hiện được chọn cần cho phép mọi thao tác được cài đặt một cách đơn giản và hiệu quả chấp nhận được. Ngoài ra, nếu một số thao tác cần chạy nhanh, thể hiện cần làm điều đó một cách tốt nhất có thể. Một thể hiện là nhanh đối với một số thao tác thường là sẽ chậm đối với các thao tác khác. Do đó, chúng ta có thể đòi hỏi nhiều cài đặt cho cùng một kiểu.

Chẳng hạn, một cách thể hiện hợp lý cho đối tượng *IntSet* là *vector*, ở đó mỗi số nguyên trong *IntSet* xuất hiện như một thành phần của *vector*. Chúng ta có thể lựa chọn cho mỗi phần tử trong tập hợp xuất hiện đúng một lần trong *vector* hoặc cho phép nó xuất hiện nhiều lần. Lựa chọn sau làm cho cài đặt thao tác chèn *insert* nhanh hơn, nhưng nó lại làm chậm thao tác *remove* và *isIn*. Vì *isIn* là dường như được gọi thường xuyên, nên chúng ta sẽ lựa chọn cái trước, và do đó, sẽ không có phần tử lặp trong *vector*.

Cài đặt trừu tượng dữ liệu trong Java

Một thể hiện thông thường có một số thành phần; trong Java, mỗi cái trong chúng là một biến thành viên của một lớp cài đặt một thành phần dữ liệu. Các cài đặt của các hàm tạo và các phương thức truy cập và vận dụng các biến thành viên.

Như vậy, khi xem xét từ góc độ một cài đặt, các đối tượng có cả hai là các phương thức và các biến thành viên. Để hỗ trợ trùu tượng, tuy nhiên, quan trọng là hạn chế truy cập đến các biến thành viên để cài đặt các phương thức và các hàm tạo; điều này cho phép chúng ta, chẳng hạn, cài đặt lại một kiểu trùu tượng mà không tác động đến code khi sử dụng kiểu này. Do đó, các biến thể hiện cần không được nhìn thấy cho người sử dụng; code mà sử dụng các đối tượng, có thể tham chiếu chỉ đến các phương thức của chúng.

Các biến thành viên là được ngăn không cho người sử dụng nhìn thấy bởi khai báo chúng là riêng tư *private*. Java cho phép các biến thành viên có các vùng nhìn khác với *private*. Nói chung đó là ý tưởng không tốt, nếu có biến thành viên là *public*.

Các khai báo của các biến thành viên không có phân loại *static*. Các biến này thuộc vào các đối tượng; đây là tập riêng của chúng cho mỗi đối tượng. Cũng có thể khai báo biến *static* bên trong một lớp. Các biến như vậy thuộc vào bản thân lớp, hơn là thuộc về các đối tượng cụ thể, cũng như phương thức *static* thuộc về lớp đó. Các biến *static* thường không sử dụng thường xuyên trong cài đặt trùu tượng dữ liệu.

Cài đặt IntSet

Phần này cho một cài đặt cho trùu tượng dữ liệu *IntSet*. Cài đặt cho trên Hình 1.5.

Vấn đề lưu ý ở đây là định nghĩa *thể hiện* của *IntSet*, hướng tới cài đặt các hàm tạo và các phương thức. Trong trường hợp này, thể hiện bao gồm một biến thành viên duy nhất. Vì biến này có vùng nhìn *private*, nó có thể được truy cập chỉ bởi code bên trong lớp.

```
public class IntSet {  
    // TỔNG QUAN: IntSets là các tập số nguyên không giới hạn.  
    private Vector els; // đây là thể hiện  
    // Các hàm tạo  
    public IntSet ( ) {  
        // EFFECTS: Khởi tạo this là rỗng.  
        els = new Vector ( ); }  
        // Các hàm tạo  
    public void insert (int x) {  
        // MODIFIES: this.  
        // EFFECTS: Bổ sung x vào this.  
        Integer y = new Integer (x);
```

```
if (getIndex (y) < 0) els.add(y); }
```

```
public void remove (int x ) {
```

```
// MODIFIES: this.
```

```
// EFFECTS: Loại bỏ x khỏi this.
```

```
int i = getIndex (new Integer (x ));
```

```
if (i < 0 return;
```

```
els.set(i, els.lastElement( ));
```

```
els.remove (els.size ( ) -1); }
```

```
public boolean isIn (int x ) {
```

```
// EFFECTS: Trả về true, nếu x ở trong this, ngược lại trả về false.
```

```
return getIndex (new Integer (x )) >= 0; }
```

```
private int getIndex (Interger x ) {
```

```
// EFFECTS: Nếu x có trong this, trả về chỉ số, nơi x xuất hiện, ngược lại trả
```

```
// về -1.
```

```
for (int i = 0; i < els.size ( ); i++)
```

```
if (x.equal(els.get(i))) return i;
```

```
return -1; }
```

```
public int size ( ) {
```

```
// EFFECTS: Trả về kích thước của this.
```

```
return els.size ( ); }
```

```
public int choose ( ) throws EmptyException {
```

```
// EFFECTS: Nếu this là null throws EmptyException, ngược lại
```

```
// trả về một phần tử bất kỳ của this.
```

```
if (els.size ( ) == 0) throws EmptyException (“IntSet.choose”);
```

```
return els.lastElement ( ); }
```

```
public String toString ( ) {
```

```
if (els.size ( ) == 0) return “IntSet:{}”;
```

```
String s = "IntSet: {" + elementAt(0).toString ();
for (int i = 0; i < els.size ( ); i++)
    s = s + ", " + elementAt(i).toString ();
return s = s + "}";
public boolean repOK ( ) {
    if (els == null) return false;
    for (int i = 0; i < els.size ( ); i++) {
        Object x = els.get(i);
        if (!(x instanceof Interger)) return false;
        for (int j = i + 1; j < els.size ( ); j++)
            if (x.equal(els.get(j))) return false;
    }
    return true; }
```

Hình 1.5. Cài đặt của IntSet

Các hàm tạo và các phương thức thuộc vào một đối tượng cụ thể của kiểu của chúng. Đối tượng được truyền như một đối số ẩn, bổ sung cho các hàm tạo và phương thức, và chúng có thể tham chiếu đến nó sử dụng từ khóa *this*. Chẳng hạn, một biến thành viên *els* có thể được truy cập sử dụng *this.els*. Tuy nhiên, tiền tố là không cần: code có thể tham chiếu đến các phương thức và các biến thành viên của đối tượng sở hữu của nó bằng cách chỉ sử dụng tên của chúng.

Cài đặt Poly

```
public class Poly {  
    // OVERVIEW: ...  
    private int[ ] trms;  
    private int deg;  
  
    // constructors  
    public Poly ( ) {  
        // EFFECTS: Initializes this to be the zero polynomial.  
        trms = new int[1]; deg = 0; }  
  
    public Poly (int c, int n) throws NegativeExponentException {  
        // EFFECTS: If n < 0 throws NegativeExponentException else  
        //   initializes this to be the Poly cxn.  
        if (n < 0)  
            throw new NegativeExponentException("Poly(int,int) constructor");  
        if (c == 0) { trms = new int[1]; deg = 0; return; }  
        trms = new int[n+1];  
        for (int i = 0; i < n; i++) trms[i] = 0;  
        trms[n] = c;  
        deg = n; }  
  
    private Poly (int n) { trms = new int[n+1]; deg = n; }  
  
    // methods  
    public int degree ( ) {  
        // EFFECTS: Returns the degree of this, i.e., the largest exponent  
        //   with a non-zero coefficient. Returns 0 if this is the zero Poly.  
        return deg; }  
  
    public int coeff (int d) {  
        // EFFECTS: Returns the coefficient of the term of this whose exponent is d.  
        if (d < 0 || d > deg) return 0; else return trms[d]; }  
  
    public Poly sub (Poly q) throws NullPointerException {  
        // EFFECTS: If q is null throws NullPointerException else  
        //   returns the Poly this - q.  
        return add (q.minus( )); }  
  
    public Poly minus ( ) {  
        // EFFECTS: Returns the Poly -this.  
        Poly r = new Poly(deg);  
        for (int i = 0; i < deg; i++) r.trms[i] = - trms[i];  
        return r; }
```

```

public Poly add (Poly q) throws NullPointerException {
    // EFFECTS: If q is null throws NullPointerException else
    // returns the Poly this + q.
    Poly la, sm;
    if (deg > q.deg) {la = this; sm = q;} else {la = q; sm = this;}
    int newdeg = la.deg; // new degree is the larger degree
    if (deg == q.deg) // unless there are trailing zeros
        for (int k = deg; k > 0; k--)
            if (trms[k] + q.trms[k] != 0) break; else newdeg--;
    Poly r = new Poly(newdeg); // get a new Poly
    int i;
    for (i = 0; i <= sm.deg && i <= newdeg; i++)
        r.trms[i] = sm.trms[i] + la.trms[i];
    for (int j = i; j <= newdeg; j++) r.trms[j] = la.trms[j];
    return r; }

public Poly mul (Poly q) throws NullPointerException {
    // EFFECTS: If q is null throws NullPointerException else
    // returns the Poly this * q.
    if ((q.deg == 0 && q.trms[0] == 0) ||
        (deg == 0 && trms[0] == 0)) return new Poly();
    Poly r = new Poly(deg+q.deg);
    r.trms[deg+q.deg] = 0; // prepare to compute coeffs
    for (int i = 0; i <= deg; i++)
        for (int j = 0; j <= q.deg; j++)
            r.trms[i+j] = r.trms[i+j] + trms[i]*q.trms[j];
    return r; }
}

```

- Một hàm trừu tượng giải thích ý nghĩa của thể hiện. Nó ánh xạ trạng thái của mỗi đối tượng thể hiện hợp lệ vào một đối tượng trừu tượng mà nó dự định thể hiện. Điều này được cài đặt bằng phương thức `toString`.
- Bất biến thể hiện định nghĩa mọi giả thiết chung mà ẩn dưới các cài đặt của các thao tác của một kiểu. Nó định nghĩa các thể hiện nào là hợp lệ bằng việc ánh xạ mỗi thể hiện vào `true` (nếu thể hiện hợp lệ) hoặc `false` (nếu thể hiện không hợp lệ). Nó được cài đặt bởi phương thức `repOk` (như trong Hình 1.5 trên).

Bảng 1.6. Hàm trừu tượng và bất biến thể hiện

Các tính chất của Trùu tượng dữ liệu:

- Một trừu tượng dữ liệu là thay đổi được, nếu nó có bất cứ phương thức thay đổi nào; ngược lại, dữ liệu trừu tượng là không thay đổi được.

- Có bốn kiểu thao tác được cung cấp bởi trùu tượng dữ liệu: các hàm tạo creators tạo ra các đối tượng mới; các hàm sinh producers tạo ra các đối tượng mới từ các đối tượng đã có như các đối số; các hàm thay đổi mutators cập nhật trạng thái của đối tượng và các hàm quan sát observers cung cấp thông tin về trạng thái của đối tượng.
- Một kiểu dữ liệu là thỏa đáng, nếu nó cung cấp đủ thao tác sao cho bất cứ người dùng nào cần thực hiện với các đối tượng của nó có thể làm được một cách thuận tiện với mức hiệu quả chấp nhận được.

1.3 Đặc tả một trùu tượng bộ duyệt

Bộ duyệt trong Java

Java cung cấp bộ duyệt bằng một kiểu đặc biệt của thủ tục, mà chúng ta tham chiếu là một bộ duyệt iterator. Một số iterators là các phương thức của các trùu tượng dữ liệu, và một trùu tượng dữ liệu có thể cung cấp một số phương thức iterator. Ngoài ra cũng có thể có iterators đứng một mình.

Một iterator trả về một kiểu đặc biệt của đối tượng dữ liệu, gọi là một bộ sinh generator. Một generator giữ trạng thái của một bộ duyệt trong thể hiện của nó. Nó có phương thức hasNext mà có thể được sử dụng để xác định có còn phần tử để lấy không, và một phương thức next để lấy phần tử tiếp theo và tịnh tiến trạng thái của bộ sinh để ghi lại sự trả về của phần tử đó.

Mọi bộ sinh generators đều thuộc về các kiểu mà là các kiểu con của giao diện iterator. Giao diện này được mô tả trên Hình 1.7; nó được định nghĩa trong gói java.util. Đặc tả cung cấp mô tả tổng quan của mọi kiểu bộ sinh; mọi kiểu như vậy có các đối tượng có hai phương thức và hành vi được chỉ ra. Ngoại lệ NoSuchElementException là một ngoại lệ không được kiểm tra, vì hy vọng mọi sử dụng một bộ sinh tránh được nguyên nhân gây ra ngoại lệ này.

```
public interface Iterator {  
    public boolean hasNext ( );  
        // EFFECTS: Trả về true, nếu còn các phần tử để lấy ra,  
        // ngược lại trả về false  
    public Object next ( ) throws NoSuchElementException;  
        // MODIFIES: this  
        // EFFECTS: Nếu còn kết quả trả về, trả về kết quả tiếp theo đó,  
        // ngược lại throws NoSuchElementException
```

}

Hình 1.7. Đặc tả giao diện Iterator

Đặc tả Iterators

Iterator và Generator

- Một iterator là một thủ tục mà trả về một generator. Một trùu tượng dữ liệu có thể có một hoặc nhiều phương thức iterator, và cũng có thể có các iterator riêng.
- Một generator là một đối tượng mà tạo ra các phần tử được sử dụng trong iterator đó. Nó có các phương thức lấy phần tử tiếp theo và xác định có còn phần tử nào nữa không. Kiểu generator là kiểu con của Iterator.
- Đặc tả một iterator xác định hành vi của một generator, một generator không có đặc tả của riêng nó. Đặc tả iterator thông thường có mệnh đề `REQUIRES` ở cuối ràng buộc code mà sử dụng generator đó.

Đặc tả của một iterator giải thích toàn bộ bộ duyệt: làm sao bộ duyệt sử dụng các đối số của nó để sinh ra bộ sinh generator, và hành vi của generator. Một đặc tả cho trên *Hình 1.8* giải thích cái gì các phương thức của generator làm là tổng quát: nó không giải thích chính xác bất cứ generator cụ thể nào làm cái gì. Chúng ta đưa các thông tin bị sót này vào đặc tả của iterator.

```
public class IntSet {  
    // như trước cộng thêm với  
    public Iterator elements ( )  
        // EFFECTS: Trả về một generator mà sẽ sinh ra mọi phần tử của this  
        // (như các Integer), mỗi cái một lần, theo thứ tự bất kỳ.  
        // REQUIRES: this cần không được thay đổi khi generator đang được  
        // sử dụng.  
}
```

Hình 1.8. Đặc tả iterator của IntSet

```
public class Poly {  
    // as before plus:  
  
    public Iterator terms ()  
        // EFFECTS: Returns a generator that will produce exponents  
        // of nonzero terms of this (as Integers) up to the degree,  
        // in order of increasing exponent.  
}
```

Đặc tả iterator của Poly

Trong Hình 1.8 chúng ta đặc tả *iterator elements* cho tập hợp; Phương thức này có thể thay thế phương thức *choose* được mô tả trước kia. Có hai điểm thú vị ở đây. Thứ nhất, lưu ý rằng đặc tả *elements* bao gồm yêu cầu trên code sử dụng *generator*. Nó không rõ ràng là *generator* cần phải làm gì nếu tập hợp bị thay đổi trong khi *generator* đang được sử dụng. Do đó, chúng ta ngăn sự thay đổi như vậy trong đặc tả *elements*. Hầu hết một *generator* trên một đối tượng thay đổi sẽ có yêu cầu này. Chúng ta khẳng định yêu cầu này trong mệnh đề *REQUIRES* như thường lệ, nhưng vì đây là yêu cầu cho sử dụng *generator*, hơn là lời gọi đến cho *iterator*, nên chúng ta đặt mệnh đề *REQUIRES* ở cuối đặc tả. Thông thường mệnh đề *REQUIRES* là ở phần đầu tiên của đặc tả. Trên thực tế, một *iterator* có thể có hai mệnh đề *REQUIRES*: một áp dụng lên một số đối số và cái khác áp đặt ràng buộc lên sử dụng *generator* được trả về.

Điểm thứ hai là, không giống như phương thức *choose*, *iterator elements* không tung ra bất cứ ngoại lệ nào. Thông thường là việc sử dụng *iterator* đã hạn chế các vấn đề liên quan đến một số đối số (như tập hợp rỗng) mà sẽ nảy sinh cho các thủ tục liên quan như *choose*.

Sử dụng generator

- Sử dụng code tương tác với một *generator* thông qua giao diện *iterator*.
- Sử dụng code cần tuân theo ràng buộc áp đặt lên nó bởi mệnh đề *REQUIRES* của *iterator*.
- Generator có thể được truyền như đối số và trả về như kết quả.
- Đôi khi là có ích chuẩn bị trước một *generator*: bỏ qua một số phần tử đầu, trước khi duyệt qua các phần tử còn lại.

```
public class Comp {  
    public static Poly diff (Poly p) throws NullPointerException {  
        // EFFECTS: If p is null throws NullPointerException else  
        //   returns the poly obtained by differentiating p.  
        Poly q = new Poly( );  
        Iterator g = p.terms( );  
        while (g.hasNext( )) {  
            int exp = ((Integer) g.next( )).intValue( );  
            if (exp == 0) continue; // ignore the zero term  
            q = q.add(new Poly(exp*p.coeff(exp), exp-1)); }  
        return q;  
    }  
  
    public static void printPrimes (int m) {  
        // MODIFIES: System.out  
        // EFFECTS: Prints all the primes less than or equal to m on System.out  
        Iterator g = Num.allPrimes( );  
        while (true) {  
            Object p = g.next( );  
            if (p > m) return;  
            System.out.println("The next prime is: " + p.toString( )); }  
    }  
  
    public static int max (Iterator g) throws EmptyException,  
        NullPointerException {  
        // REQUIRES: g contains only Integers  
        // MODIFIES: g  
        // EFFECTS: If g is null throws NullPointerException; if g is empty,  
        //   throws EmptyException; else consumes all elements of g  
        //   and returns the largest int in g.  
        try {  
            int m = ((Integer) g.next( )).intValue( );  
            while (g.hasNext( )) {  
                int x = g.next( );  
                if (m < x) m = x; }  
            return m; }  
        catch (NoSuchElementException e)  
        { throw new EmptyException("Comp.max"); }  
    }  
}
```

Hình 1.9. Sử dụng iterator và generator

Cài đặt iterator

- Một cài đặt Iterator đòi hỏi cài đặt một lớp cho generator gắn kết.
- Một lớp generator là một lớp bên trong tĩnh (static inner class). Nó được lồng trong lớp chứa iterator và có thể truy cập thông tin riêng của lớp chứa nó.
- Lớp generator định nghĩa một kiểu con của Iterator.
- Một cài đặt của generator giả thiết code sử dụng tuân theo các ràng buộc áp đặt lên nó bởi mệnh đề *REQUIRES* của iterator.

Hình 1.10. Cài đặt Iterators

```
public class Poly {  
    private int[] trms;  
    private int deg;  
  
    public Iterator terms () { return new PolyGen(this); }  
  
    // inner class  
    private static class PolyGen implements Iterator {  
        private Poly p; // the Poly being iterated  
        private int n; // the next term to consider  
  
        PolyGen (Poly it) {  
            // REQUIRES: it != null  
            p = it;  
            if (p.trms[0] == 0) n = 1; else n = 0; }  
  
        public boolean hasNext () { return n <= p.deg; }  
  
        public Object next () throws NoSuchElementException {  
            for (int e = n; e <= p.deg; e++)  
                if (p.trms[e] != 0) { n = e + 1; return new Integer(e); }  
            throw new NoSuchElementException("Poly.terms"); }  
    } // end PolyGen  
}
```

Cài đặt iterator terms

```
public class Num {  
  
    public static Iterator allPrimes( ) { return new PrimesGen( ); }  
  
    // inner class  
    private static class PrimesGen implements Iterator {  
        private Vector ps; // primes yielded  
        private int p; // next candidate to try  
  
        PrimesGen ( ) { p = 2; ps = new Vector( ); }  
  
        public boolean hasNext ( ) { return true; }  
  
        public Object next ( ) {  
            if (p == 2) { p = 3; return 2; }  
            for (int n = p; true; n = n + 2)  
                for (int i = 0; i < ps.size( ); i++) {  
                    int el = ((Integer) ps.get(i)).intValue( );  
                    if (n%el == 0) break; // not a prime  
                    if (el*el > n) { // have a prime  
                        ps.add(new Integer(n)); p = n + 2; return n; }  
                }  
        }  
    } // end PrimesGen  
}
```

Hình 1.11. Cài đặt iterator *allPrimes*

Orderd List

Bây giờ chúng ta cung cấp một ví dụ khác về một bộ duyệt. *Iterator* này là một phần của *OrderedIntList* – một trùu tượng thay đổi được mà giữ các phần tử của nó theo thứ tự sắp xếp. *Iterator smallToBig* sẽ sinh ra các phần tử của danh sách theo thứ tự này. Lưu ý rằng *OrderedIntList* sẽ không phù hợp nếu không có *iterator*, vì sẽ không có cách thuận tiện để lấy ra cái mà ở trong danh sách mà không xóa các phần tử khỏi danh sách.

Đặc tả *OrderedIntList* được cho trên *Hình 1.12*. Các phương thức *addEl* và *remEl* tung ra một ngoại lệ khi phần tử đã có hoặc chưa có ở trong danh sách sắp xếp. Lựa chọn này phản ánh niềm tin rằng người sử dụng sẽ muốn biết về tình huống này mà không kiểm tra nó một cách tường minh (bằng gọi *isIn*).

```
public class OrderedIntList {
```

// *TỔNG QUAN*: Một danh sách sắp xếp là một danh sách các số nguyên được // sắp xếp và có thể thay đổi được. Một danh sách thông thường là dãy // $[x_1, \dots, x_n]$, ở đó $x_i < x_j$, nếu $i < j$.

// Các hàm tạo

public OrderedIntList ()

// *EFFECTS*: Khởi tạo this là một danh sách sắp xếp rỗng.

// Các phương thức

public void addEl (int el) throws DuplicateException

// *MODIFIES*: this

// *EFFECTS*: Nếu el ở trong this, throws DuplicateException;
// ngược lại, bổ sung el vào this.

public void remEl (int el) throws NotFoundException

// *MODIFIES*: this

// *EFFECTS*: Nếu el không ở trong this, throws NotFoundException,
// ngược lại, loại bỏ el khỏi this.

public boolean isIn (int el)

// *EFFECTS*: Nếu el ở trong this, trả về true, ngược lại, trả về false.

public boolean isEmpty ()

// *EFFECTS*: Trả về true, nếu this là rỗng, ngược lại trả về false.

public int least () throws EmptyException

// *EFFECTS*: Nếu this là rỗng trả về EmptyException;
// ngược lại, trả về phần tử nhỏ nhất của this

public Iterator smallToBig ()

// *EFFECTS*: Trả về một generator mà sẽ sinh ra các phần tử của this (như // là các số nguyên), mỗi một chính xác một lần, theo thứ tự từ nhỏ nhất đến // lớn nhất.

// *REQUIRES*: this cần không được thay đổi trong khi generator đang được // sử dụng

```
public boolean repOK( )
public String toString( )
}
```

Hình 1.12. Đặc tả của danh sách sắp xếp OrderedIntList

```
public class OrderedIntList {
    private boolean empty;
    private OrderedIntList left, right;
    private int val;

    public OrderedIntList ( ) { empty = true; }

    public void addEl (int el) throws DuplicateException {
        if (empty) {
            left = new OrderedIntList( ); right = new OrderedIntList( );
            val = el; empty = false; return; }
        if (el == val)
            throw new DuplicateException("OrderedIntList.addEl");
        if (el < val) left.addEl(el); else right.addEl(el); }

    public void remEl (int el) throws NotFoundException {
        if (empty) throw new NotFoundException("OrderedIntList.remEl");
        if (el == val)
            try { val = right.least( ); right.remEl(val); }
            catch (EmptyException e) { empty = left.empty; val = left.val;
                right = left.right; left = left.left; return; }
        else if (el < val) left.remEl(el); else right.remEl(el); }

    public boolean isIn (int el) {
        if (empty) return false;
        if (el == val) return true;
        if (el < val) return left.isIn(el); else return right.isIn(el);
    }

    public boolean isEmpty ( ) { return empty; }

    public int least ( ) throws EmptyException {
        if (empty) throw new EmptyException("OrderedIntList.least");
        try { return left.least( ); }
        catch (EmptyException e) { return val; }
    }
}
```

Hình 1.13. Cài đặt OrderedIntList

```
public Iterator smallToBig ( ) { return new OLGens(this, count( )); }

private int count ( ) {
    if (empty) return 0;
    return 1 + left.count( ) + right.count( ); }

// inner class
private static class OLGens implements Iterator {
    private int cnt; // count of number of elements left to generate
    private OLGens child; // the current sub-generator
    private OrderedIntList me; // my node

    OLGens (OrderedIntList o, int n) {
        // REQUIRES: o != null
        cnt = n;
        if (cnt > 0) { me = o;
            child = new OLGens(o.left, o.left.count( )); }
    }

    public boolean hasNext ( ) { return cnt > 0; }

    public Object next ( ) throws NoSuchElementException {
        if (cnt == 0)
            throw new NoSuchElementException("OrderedIntList.smallToBig");
        cnt--;
        try { return new Integer(child.next( )); }
        catch (NoSuchElementException e) { }
        // if get here, must have just finished on the left;
        child = new OLGens(me.right, cnt);
        return new Integer(me.val); }
    } // end of OLGens
```

Hình 1.14. Cài đặt Iterator của OrderedIntList

Cài đặt của *OrderedIntList* sử dụng cây sắp xếp. Ý tưởng này là mỗi đỉnh trên cây chứa một giá trị và hai đỉnh con, một ở bên trái và một ở bên phải. Hai đỉnh con này bản thân chúng là các danh sách sắp xếp, và do đó thể hiện là đệ qui. Cây này được sắp xếp sao cho mọi giá trị ở cây gốc con bên trái là nhỏ hơn giá trị ở gốc cha, và mọi giá trị ở cây gốc con bên phải là lớn hơn giá trị ở nút cha, tức là nó là cây tìm kiếm nhị phân.

Hình 1.13 cho phần cài đặt của danh sách sắp xếp. Lưu ý rằng, cài đặt *addEl* thông báo không tường minh ngoại lệ *DuplicateException* được phát sinh bởi các lời gọi đệ qui của nó; cài đặt của *remEl* là tương tự.

Bộ duyệt *Iterator smallToBig* được cài đặt trong Hình 1.14. Bộ sinh bắt đầu từ sinh các phần tử ở cây con bên trái. Sau khi mọi phần tử này được sinh ra, nó trả về giá trị của đỉnh trên cùng và sau đó tạo ra các phần tử của cây con bên phải. Vì điều này quan trọng cho hai phương thức của *generator*, và đặc biệt phương thức *hasNext*, để thực hiện hiệu quả, cài đặt cần giữ thông tin có bao nhiêu phần tử ở bên trái được tạo ra. Nó làm điều đó bằng việc tính có bao nhiêu phần tử ở trong danh sách tại thời điểm việc lặp bắt đầu.

1.4 Đặc tả một trùu tượng dữ liệu đa hình

Đa hình:

- *Đa hình khái quát các trùu tượng sao cho chúng có thể làm việc cho nhiều kiểu. Nó cho phép chúng ta tránh phải định nghĩa lại trùu tượng khi chúng ta muốn sử dụng chúng cho nhiều hơn một kiểu, thay vào đó, một trùu tượng duy nhất trỏ nên có ích rộng lớn hơn.*
- *Một thủ tục hay một iterator có thể đa hình đối với các kiểu của một hay nhiều đối số. Một trùu tượng dữ liệu có thể là đa hình đối với các kiểu phần tử mà đối tượng của nó chia.*

```
public class Set {  
    // OVERVIEW: Sets are unbounded, mutable sets of objects.  
    // null is never an element of a Set. The methods use equals  
    // to determine equality of elements.  
  
    // constructors  
    public Set ( )  
        // EFFECTS: Initializes this to be empty.  
  
    // methods  
    public void insert (Object x) throws NullPointerException  
        // MODIFIES: this  
        // EFFECTS: If x is null throws NullPointerException else  
        // adds x to the elements of this.  
  
    public void remove (Object x)  
        // MODIFIES: this  
        // EFFECTS: If x is in this, removes x from this, else does nothing.  
  
    public boolean isIn (Object x)  
        // EFFECTS: Returns true if x is in this else returns false.  
  
    public boolean subset (Set s)  
        // EFFECTS: If all elements of this are elements of s returns true  
        // else returns false specifications of size and elements.  
        // Specifications of size and elements  
}
```

Đặc tả bộ phận của Set đa hình

Có một khác biệt quan trọng giữa code trên và code của IntSet là IntSet chỉ chứa các số nguyên và điều này được đảm bảo bởi chương trình dịch. Không có đảm bảo như vậy đối với Set, ngay cả khi sử dụng Set đồng nhất, tức là mọi phần tử ở đó đều thuộc cùng một kiểu. chương trình dịch cũng không ép buộc ràng buộc này.

Trong các phiên bản mới của java đã có thể cài đặt các lớp đa hình tham số để chương trình dịch đảm bảo ràng buộc đồng nhất kiểu như đối với một kiểu cụ thể.

```
public class Set {  
  
    private Vector els;  
  
    public Set ( ) { ole = new Vector( ); }  
    private Set (Vector x) { els = x; }  
  
    public void insert (Object x) throws NullPointerException {  
        if (getIndex(x) < 0) els.add(x); }  
  
    private int getIndex (Object x) {  
        for (int i = 0; i < els.size( ); i++)  
            if (x.equals(els.get(i))) return i;  
        return -1; }  
  
    public boolean subset (Set s) {  
        if (s == null) return false;  
        for (int i = 0; i < els.size( ); i++)  
            if (!s.isIn(els.get(i))) return false;  
        return true; }  
  
    public Object clone ( ) { return new Set((Vector) els.clone( )); }  
}
```

Cài đặt bộ phận của Set đa hình

Sử dụng trùu tượng dữ liệu đa hình:

```
Set s = new Set( );  
s.insert(new Integer(3));  
...  
Iterator g = s.elements( );  
while (g.hasNext( )) {  
    int i = ((Integer) g.next( )).intValue( );  
    ... }
```

1.5 Phân cấp kiểu

Một phân cấp kiểu:

- Một phân cấp kiểu được sử dụng để định nghĩa họ kiểu bao gồm một kiểu cha và các kiểu con của nó. Phân cấp có thể được mở rộng qua nhiều mức.
- Một số họ kiểu được sử dụng để cung cấp đa cài đặt của một kiểu nào đó: các kiểu con cung cấp các cài đặt khác nhau của kiểu cha.
- Hơn nữa, các kiểu con có thể mở rộng hành vi của kiểu cha, chẳng hạn cung cấp thêm các phương thức.
- Nguyên lý thế cung cấp trừu tượng bởi đặc tả cho họ kiểu bằng cách yêu cầu là các kiểu con có hành vi phù hợp với đặc tả kiểu cha của chúng.

Ngữ nghĩa của kiểu con

Các kiểu con cần thỏa mãn *nguyên lý thế* sao cho người sử dụng có thể viết và suy luận về code chỉ sử dụng kiểu đặc tả của kiểu cha.

Như vậy *nguyên lý thế* yêu cầu đặc tả của kiểu con hỗ trợ suy luận dựa trên đặc tả của kiểu cha. Ba tính chất sau cần được thỏa mãn:

- Qui tắc Signature. Các kiểu con cần phải có mọi phương thức của kiểu cha và signatures của các phương thức kiểu con cần tương thích với các signatures của các phương thức tương ứng của kiểu cha.
- Qui tắc Method. Các lời gọi các phương thức kiểu con cần có “hành vi giống” như của các phương thức lớp cha tương ứng.
- Qui tắc Property. Kiểu con cần bảo toàn mọi tính chất mà đã được chứng minh cho các đối tượng kiểu cha.

Suy luận về nguyên lý thế:

- Qui tắc Signature đảm bảo rằng, nếu một chương trình là đúng đắn kiểu dựa trên đặc tả lớp cha, thì nó cũng đúng đắn kiểu dựa trên đặc tả lớp con.
- Qui tắc Method đảm bảo suy luận về các lời gọi của lớp cha là đúng ngay cả trên thực tế qua lời gọi đến code cài đặt cho lớp con.
- Qui tắc Property đảm bảo rằng suy luận về các tính chất của các đối tượng dựa trên đặc tả lớp cha là vẫn đúng khi các đối tượng thuộc về lớp con. Các tính chất này cần được khẳng định khi xét đặc tả lớp cha.

Định nghĩa phân cấp kiểu:

- Kiểu cha được định nghĩa bởi một lớp hoặc giao diện, mà cung cấp đặc tả của nó, trong trường hợp lớp, cung cấp cài đặt một phần hoặc hoàn chỉnh.
- Lớp trừu tượng chỉ cung cấp cài đặt một phần; nó không có đối tượng và không có hàm tạo mà người sử dụng có thể gọi đến.
- Lớp con được kế thừa cài đặt của các phương thức lớp cha, nhưng nó cũng có thể ghi đè các cài đặt này (đối với các phương thức nonfinal).

- Biểu diễn lớp con bao gồm các biến khởi tạo của nó và các biến từ lớp cha của nó, nhưng nó chỉ có thể truy cập đến các biến khởi tạo của lớp cha, nếu các biến này được khai báo là *protected*.

```
public class IntSet {  
    // OVERVIEW: IntSets are mutable, unbounded sets of integers.  
    // A typical IntSet is {x1, ..., xn}.  
  
    // constructors  
    public IntSet ()  
        // EFFECTS: Initializes this to be empty.  
  
    // methods  
    public void insert (int x)  
        // MODIFIES: this  
        // EFFECTS: Adds x to the elements of this.  
  
    public void remove (int x)  
        // MODIFIES: this  
        // EFFECTS: Removes x from this.  
  
    public boolean isIn (int x)  
        // EFFECTS: If x is in this returns true else returns false.  
  
    public int size ()  
        // EFFECTS: Returns the cardinality of this.  
  
    public Iterator elements ()  
        // EFFECTS: Returns a generator that produces all elements of this  
        // (as Integers), each exactly once, in arbitrary order.  
        // REQUIRES: this not be modified while the generator is in use.  
  
    public boolean subset (IntSet s)  
        // EFFECTS: Returns true if this is a subset of s else returns false.  
  
    public boolean repOk ()  
}
```

Đặc tả IntSet

```
public class IntSet {  
    private Vector els; // the elements  
  
    public IntSet () { els = new Vector (); }  
  
    private int getIndex (Integer x) { ... }  
  
    public boolean isIn (int x) {  
        return getIndex(new Integer(x)) >= 0; }  
  
    public boolean subset (IntSet s) {  
        if (s == null) return false;  
        for (int i = 0; i < els.size(); i++)  
            if (!s.isIn(((Integer) els.get(i)).intValue()))  
                return false;  
        return true; }  
  
    // implementations of other methods go here  
}
```

Cài đặt bộ phận IntSet

```
public class MaxIntSet extends IntSet {  
    // OVERVIEW: MaxIntSet is a subtype of IntSet with an additional  
    // method, max, to determine the maximum element of the set.  
  
    // constructors  
    public MaxIntSet ()  
        // EFFECTS: Makes this be the empty MaxIntSet.  
  
    // methods  
    public int max () throws EmptyException  
        // EFFECTS: If this is empty throws EmptyException else returns the  
        // largest element of this.  
}
```

Đặc tả MaxIntSet

```
public class MaxIntSet extends IntSet {
    private int biggest; // the biggest element if set is not empty

    public MaxIntSet ( ) { super( ); }

    public void insert (int x) {
        if (size ( ) == 0 || x > biggest) biggest = x;
        super.insert(x); }

    public void remove (int x) {
        super.remove(x);
        if (size( ) == 0 || x < biggest) return;
        Iterator g = elements( );
        biggest = ((Integer) g.next( )).intValue( );
        while (g.hasNext( )) {
            int z = ((Integer) g.next( )).intValue( );
            if (z > biggest) biggest = z; }
    }

    public int max ( ) throws EmptyException {
        if (size( ) == 0) throw new EmptyException ("MaxIntSet.max");
        return biggest; }

    public boolean repOk ( ) {
        if (!super.repOk( )) return false;
        if (size( ) == 0) return true;
        boolean found = false;
        Iterator g = elements ( );
        while (g.hasNext( )) {
            int z = ((Integer) g.next( )).intValue( );
            if (z > biggest) return false;
            if (z == biggest) found = true; }
        return found;
    }
}
```

Cài đặt MaxIntSet

1.6 Đặc tả và các tập thỏa đặc tả

Mục đích của một đặc tả là xác định hành vi của một trừu tượng. Người sử dụng sẽ dựa trên hành vi này, trong khi người cài đặt sẽ cần phải cung cấp nó. Một cài đặt mà cung cấp hành vi được mô tả được gọi là thỏa mãn đặc tả.

Chúng ta định nghĩa ý nghĩa của một đặc tả là một tập tất cả các modules chương trình mà thỏa mãn nó. Chúng ta gọi đây là tập thỏa đặc tả. Chẳng hạn xét đặc tả sau:

```
static int p (int y)  
// REQUIRES: y > 0  
// EFFECTS: Return x such that x > y.
```

Đặc tả này được thỏa mãn bởi bất cứ thủ tục tên *p* nào, mà khi được gọi với đối số lớn hơn không, trả về một giá trị lớn hơn đối số của nó.

Các thành viên của tập thỏa đặc tả chứa

```
static int p (int y) { return y + 1}  
static int p (int y) { return y * 2}  
static int p (int y) { return y * 3}
```

Mọi đặc tả được thỏa mãn bởi một tập vô hạn các chương trình. Có thể tóm tắt các định nghĩa này như sau.

Đặc tả:

- Một đặc tả mô tả hành vi của một trừu tượng nào đó.
- Một cài đặt thỏa mãn một đặc tả nào đó nếu nó cung cấp hành vi được mô tả.
- Ý nghĩa của một đặc tả là một tập tất cả chương trình mà thỏa mãn nó. Tập này được gọi là tập thỏa đặc tả.

Điều quan trọng là nhớ rằng, một đặc tả, một tập thỏa đặc tả của nó và một thành viên cụ thể của tập thỏa đặc tả là những thứ rất khác nhau, như sự khác nhau của một chương trình, tập các lệnh có thể của chương trình đó và một thực thi của chương trình đó trên một tập dữ liệu duy nhất.

1.7 Một số tiêu chuẩn cho đặc tả.

Các thuộc tính của một đặc tả tốt:

- Một đặc tả là vừa đủ hẹp, nếu nó loại bỏ mọi cài đặt mà là không chấp nhận được cho người sử dụng trừu tượng.
- Một đặc tả là đủ tổng quát, nếu nó không loại bỏ các cài đặt chấp nhận được.
- Một đặc tả cần phải rõ ràng sao cho là dễ hiểu đối với người sử dụng.

Đặc tả tốt có nhiều dạng, nhưng mọi dạng đó đều có một số đặc tính chung. Ba đặc tính quan trọng là tính thu hẹp, tính tổng quan và tính rõ ràng, sẽ được bàn luận sau đây.

1.7.1 Tính thu hẹp

Có sự khác biệt rất lớn giữa việc biết rằng một số thành viên của tập thỏa đặc tả là phù hợp và việc biết rằng mọi thành viên của tập thỏa đặc tả là phù hợp. Điều này cũng tương tự như sự khác biệt giữa việc biết rằng một chương trình làm việc với một số đầu vào và việc biết rằng nó làm việc với mọi đầu vào. Sự khác biệt này chúng ta sẽ nhấn mạnh khi mà kiểm thử chương trình. Một đặc tả cần thu hẹp vừa đủ để suy ra bất cứ cài đặt nào mà không chấp nhận được cho người sử dụng trừu tượng đó. Yêu cầu này là cơ sở của hầu hết mọi việc sử dụng đặc tả.

Nói chung, việc bàn luận một đặc tả có là vừa đủ hạn chế hay không bao gồm cả việc sử dụng thành viên nào của tập thỏa đặc tả. Có một số lỗi chung, mà hầu như luôn dẫn đến đặc tả hạn chế không thích đáng. Một thiếu sót như vậy là có lỗi trong việc xác định các yêu cầu cần thiết trong mệnh đề REQUIRES. Chẳng hạn, Hình 1.15 sau cho ba đặc tả cho một bộ duyệt *elems iterator* cho một túi bag các số nguyên. (Một IntBag giống như một tập các số nguyên IntSet, nhưng các phần tử có thể xuất hiện nhiều hơn một lần, chẳng hạn bag có thể có 3 hai lần, bag đôi khi còn được gọi là đa tập hợp multiset). Đặc tả đầu tiên lỗi khi giải quyết vấn đề, điều gì sẽ xảy ra nếu bag thay đổi, trong khi bộ sinh generator trả về các phần tử elems đang sử dụng. Khi đó nó cho phép các cài đặt thể hiện các hành vi rất khác nhau. Chẳng hạn, việc thay đổi bag có tác động đến các giá trị trả về bởi bộ sinh generator không?

Một cách để giải quyết vấn đề này là yêu cầu túi bag không được thay đổi khi bộ sinh generator đang được sử dụng, như đã làm trong đặc tả thứ hai. Đặc tả này có thể hoặc không thể là hạn chế đủ hẹp, vì nó không chứa thứ tự mà theo đó các phần tử được trả về. Nó có thể là tốt hơn nếu nó định nghĩa một thứ tự hoặc chứa câu “theo thứ tự tùy ý”. Ngoài ra đặc tả này lỗi khi làm rõ cần phải làm gì khi một phần tử được chứa trong bag nhiều hơn một lần. Vì điều đó, nó không cần ngay cả nói tường minh là bộ sinh trả về chỉ các phần tử mà ở trong túi bag. Đặc tả thứ ba điều chỉnh sự khác biệt này và là đặc tả tốt.

public iterator elems()

// EFFECTS: trả về một bộ sinh generator mà sản sinh ra mỗi phần tử của this

public iterator elems()

// EFFECTS: trả về một bộ sinh generator mà sản sinh ra mỗi phần tử của this

// REQUIRES: this không được sửa trong khi bộ sinh generator đang sử dụng

public iterator elems()

// EFFECTS: trả về một bộ sinh generator mà sản sinh ra mỗi phần tử của this theo thứ tự tùy ý. Mỗi phần tử được tạo ra chính xác một số lần mà nó xuất hiện trong this.

// REQUIRES: this không được sửa trong khi bộ sinh generator đang sử dụng

Hình 1.15. Ba đặc tả của elems

Các lỗi khác thường là thất bại khi xác định khi nào ngoại lệ *exceptions* cần được cảnh báo và thất bại khi đặc tả hành vi tại các trường hợp biên. Chẳng hạn xem xét thủ tục *IndexString* mà nhận các xâu s1 và s2 và nếu s1 là một từ con của s2, trả về chỉ số tại đó ký tự đầu tiên của s1 xuất hiện trong s2. Chẳng hạn, *Indexing (“ab”, “babc”)* trả về 1. Một đặc tả mà chỉ chứa thông tin này có thể không là hạn chế đủ, vì nó không giải thích cái gì sẽ xảy ra khi s1 không là từ con của s2, hoặc nếu nó xuất hiện nhiều lần trong s2, hoặc nếu s1 hoặc s2 là rỗng. Đặc tả trên Hình 1.16 sau là hạn chế đủ.

public static int indexingString (String s1, String s2)

throws NullPointerException, EmptyException

// EFFECTS: If s1 hoặc s2 là rỗng, throws NullPointerException, else

// nếu s1 là rỗng, throws EmptyException, else

// nếu s1 xuất hiện trong s2, trả về chỉ số nhỏ nhất mà tại đó s1 xuất hiện, else trả về -1. Chẳng hạn,

// Indexing (“ab”, “babc”) = 1

// Indexing (“b”, “a”) = -1

Hình 1.16. Đặc tả của thủ tục IndexingString

1.7.2 Tính tổng quan

Một đặc tả tốt phải là đủ tổng quan để đảm bảo rằng chỉ một số ít chương trình chấp nhận được hoặc không có cái nào bị loại bỏ. Tính quan trọng của tiêu chuẩn tổng quan có thể ít hiển nhiên hơn so với tính hạn chế. Không phải cốt yếu là đảm bảo rằng cài đặt không chấp nhận được loại bỏ, nhưng cài đặt mong muốn hơn (mà là hiệu quả và rõ ràng) cũng cần không phải bị loại bỏ. Chẳng hạn, đặc tả sau

public static float sqrt (float sq, float e)

// REQUIRES: $sq \geq 0 \ \&\& e > 0.001$

// EFFECTS: Trả về rt sao cho $0 \leq (rt * rt - sq) \leq e$

chứa cài đặt thuật toán tìm căn bậc hai gần đúng mà lớn hơn hoặc bằng căn bậc hai thực. Ràng buộc này có thể làm mất mát không cần thiết tính hiệu quả.

Mong muốn của chúng ta là làm cho đặc tả tổng quan nhất có thể để dẫn đến kiểu đặc tả định nghĩa. Một đặc tả định nghĩa liệt kê tường minh các tính chất mà các thành viên của tập thỏa đặc tả phải thể hiện được. Một cách khác với đặc tả định nghĩa là đặc tả thao tác. Một đặc tả thao tác, thay vì mô tả các tính chất của tập thỏa đặc tả thì lại nêu các bước để xây dựng nó. Chẳng hạn,

public static int search (int[] a, int x)

throws NotFoundException, NullPointerException

// EFFECTS: if a là null throws NullPointerException else kiểm tra lần lượt

// $a[0], a[1], \dots$ và trả về chỉ số của phần tử đầu tiên mà bằng x . Nếu không có phần tử nào bằng x .

là một đặc tả thao tác của tìm kiếm, trong khi đó

public static int search (int[] a, int x)

throws NotFoundException, NullPointerException

// EFFECTS: if a là null throws NullPointerException else trả về chỉ số i sao cho

// $a[i] = x$; Thông báo NotFoundException, nếu không có i như vậy.

là một đặc tả định nghĩa của tìm kiếm. Đặc tả đầu tiên chỉ ra rằng cài đặt như thế nào, trong khi đặc tả thứ hai chỉ mô tả tính chất mà đầu vào và đầu ra của nó phải thỏa mãn. Không chỉ vì đặc tả định nghĩa ngắn hơn, mà nó cũng cho người cài đặt tự do hơn, có thể lựa chọn kiểm tra các phần tử của mảng theo một thứ tự nào đó khác với đi từ đầu đến cuối.

Các đặc tả thao tác có một số ưu điểm. Quan trọng nhất là chúng trông có vẻ tương đối dễ dàng được tạo ra bởi người lập trình được đào tạo – đặc biệt vì xây dựng của họ rất gần với lập trình. Chúng thông thường dài hơn đặc tả định nghĩa, tuy nhiên chúng thường dẫn đến đặc tả chi tiết quá. Đặc tả thao tác cho tìm kiếm, chẳng hạn, chỉ rõ chỉ số nào cần được trả về nếu x xuất hiện nhiều hơn một lần trong mảng a . Điều đó có thể là hạn chế nhiều hơn là mong muốn. Như một ví dụ khác, thử viết đặc tả thao tác cho thủ tục khai căn bậc hai.

Một kiểm tra tốt cho tính tổng quát là kiểm tra mỗi tính chất được yêu cầu bởi đặc tả trong cả hai mệnh đề REQUIRES và EFFECTS, và hỏi xem nó có thực sự cần thiết không. Nếu

nó không, thì cần loại bỏ hoặc làm yếu đi. Cũng như vậy, một phần của một đặc tả, mà là thao tác hơn là định nghĩa cần được xem xét lại với sự suy xét kỹ.

1.7.3 Tính rõ ràng

Khi ta nói làm cho chương trình tốt, chúng ta xem xét không chỉ các tính toán nó mô tả, mà còn các tính chất của bản thân văn bản chương trình, chẳng hạn như, nó có tách được các phần rõ ràng không và ghi chú đầy đủ không. Tương tự, khi ta đánh giá một đặc tả, chúng ta cần xem xét không chỉ các tính chất của tập thỏa đặc tả, mà còn xét các tính chất của bản thân đặc tả, như nó có dễ đọc không.

Một đặc tả tốt cần làm cho trao đổi dễ dàng hơn giữa mọi người. Một đặc tả là hạn chế đủ và tổng quát đủ, tức là nó có ý nghĩa chính xác, nhưng điều đó chưa đủ. Nếu ý nghĩa này là rất khó để người đọc khám phá và các công cụ đặc tả là rất hạn chế.

Mọi người có thể hiểu sai đặc tả theo hai cách. Họ có thể nghiên cứu nó và đi đến nhận thức là họ không hiểu nó. Chẳng hạn, bạn đọc của đặc tả thứ hai *elems* trên Hình 1.1 trên có thể bị nhầm lẫn về việc cần làm gì nếu một phần tử xuất hiện trong *bag* nhiều hơn một lần. Điều này là rắc rối, nhưng nó không nguy hiểm như khi người ta nghĩ rằng họ hiểu đặc tả, nhưng trên thực tế là không. Trong trường hợp như vậy, người sử dụng và người cài đặt trừu tượng có thể hiểu đặc tả đó theo các cách khác nhau, dẫn đến các modules không làm việc được với nhau. Chẳng hạn, người cài đặt *elems* có thể quyết định tạo ra mỗi phần tử một số lần như nó xuất hiện trong túi, trong khi đó người sử dụng dự kiến mỗi phần tử chỉ được tạo một lần.

Tính rõ ràng là quan trọng, nhưng là tiêu chuẩn không định hình. Dễ dàng nói là một đặc tả tốt cần phải là dễ hiểu, nhưng rất khó để nói làm sao đạt được điều đó. Nhiều yếu tố, trong số đó tính súc tích, dư thừa và cấu trúc có thể là quan trọng nhất, hỗ trợ cho tính rõ ràng.

Thể hiện chính xác nhất có thể không luôn là đặc tả tốt nhất, nhưng nó thường xuyên là điểm bắt đầu tốt nhất. Có nhiều nguyên nhân đúng để tăng kích thước của đặc tả bằng cách bổ sung thông tin dư thừa hoặc các mức độ cấu trúc, nhưng cũng tránh nói dài dòng. Khi đặc tả dài thêm, nó có thể chứa thêm các lỗi và khó đọc cẩn thận hơn và làm cho dễ hiểuลำ hơn.

Quan trọng là không nhầm lẫn giữa độ dài và tính đầy đủ. Giống như chương trình mà tăng bằng cách bổ sung thêm thường dài hơn chúng cần phải thế. Thay vì bổ sung vào đặc tả, quan trọng là quay lại từ chỗ thay đổi cục bộ và xem có cách nào cũng có thông tin. Viết nhiều lần mô tả ngắn đầy đủ hơn là viết đầy đủ dài.

Một đặc tả chứa văn bản dư thừa là kém chính xác hơn nó có thể như vậy. Dư thừa cần không được đưa vào nếu không có lý do xác đáng, nhưng nó có thể được suy xét theo hai cách: giảm khả năng mà đặc tả có thể bị hiểu lầm và bị phát hiện ra lỗi.

Theo nhiều khía cạnh, vai trò đặc tả cũng giống như cuốn sách. Nó cần phải được thiết kế không chỉ chứa thông tin mà còn trao đổi thông tin một cách hiệu quả. Dư thừa có thể được sử dụng để giảm khả năng mà điểm quan trọng sẽ bị bỏ qua. Mẫu chốt là thể hiện cùng một thông tin theo nhiều cách, là dư thừa nhưng không bị lặp. Xét ví dụ sau:

static boolean subset (IntSet s1, IntSet s2)

throws NullPointerException

// EFFECTS: if s1 hoặc s2 là null throws NullPointerException, else

// return true if s1 là tập con của s2, else return false.

static boolean subset (IntSet s1, IntSet s2)

throws NullPointerException

// EFFECTS: if s1 hoặc s2 là null throws NullPointerException, else

// return true if mỗi phần tử của s1 là một phần tử của s2, else return false.

static boolean subset (IntSet s1, IntSet s2)

throws NullPointerException

// EFFECTS: if s1 hoặc s2 là null throws NullPointerException, else

// return true if s1 là tập con của s2, else return false, tức là

// return true if mỗi phần tử của s1 là một phần tử của s2, else return false.

Đặc tả thứ nhất là chính xác và đối với đa số bạn đọc, nó là khá rõ ràng. Tuy nhiên, một số bạn đọc có thể hoài nghi: từ “tập con” được chọn hay là tác giả muốn nói đến “tập con thực sự”. Đặc tả thứ hai, tuy khó đọc hơn đặc tả thứ nhất, nhưng không còn hoài nghi về từ tập con. Đặc tả thứ ba có dư thừa, nhưng rõ ràng là dễ đọc và hiểu chính xác.

Bằng việc bắt đầu một việc theo nhiều cách, một đặc tả giúp cho bạn đọc hiểu chính xác, ngăn hiểu lầm và giảm thời gian nghiên cứu đặc tả. Một kiểu dư thừa có ích là đưa ra một hoặc nhiều ví dụ như đã làm với *IndexingString*.

Một đặc tả khẳng định cùng một sự việc theo nhiều cách cũng cho phép các bạn đọc khác nhau sẽ tìm được cách thể hiện khác nhau của cùng một thông tin cho dễ hiểu. Thông thường, phân phê phán của một đặc tả là khái niệm tên, mà có ý nghĩa đối với một số bạn đọc, nhưng không có ý nghĩa với các bạn đọc khác. Chẳng hạn, xét

```

static float pv (float inc, float r, int n)

// REQUIRES: inc > 0 && r > 0 && n > 0

// EFFECTS: trả về giá trị hiện tại thu nhập hàng năm của inc cho n năm với tỷ lệ
// đầu tư không rủi ro r

// tức là,  $pv(inc, r, n) = inc + (inc/(1+r)) + \dots + (inc/(1+r)^{n-1})$ .
// Chẳng hạn,  $pv(100, 0.10, 3) = 100 + 100/1.1 + 100/1.21$ 

```

Đối với bạn đọc hiểu về tài chính, một đặc tả không sử dụng câu “giá trị hiện tại” sẽ không là dễ hiểu như đã làm bên trên. Đối với bạn đọc không có kiến thức tài chính, phần đặc tả tiếp theo sau “tức là” gần như là không có giá trị. Phần đặc tả tiếp theo sau “Chẳng hạn” có thể được sử dụng trong nhóm bạn để khẳng định việc hiểu của họ.

Nếu các bạn đọc thu được lợi ích từ dư thừa, điều quan trọng là mọi thông tin dư thừa cần được đánh dấu rõ ràng. Ngược lại, bạn đọc có thể mất thêm thời gian để hiểu có thông tin mới gì từ đây. Một cách tốt khác để chỉ ra thông tin này là dư thừa là dùng các lời tựa như “tức là”, “chẳng hạn”.

Dư thừa không làm giảm số lỗi trong một đặc tả. Thay vào đó, nó làm chúng tin cậy hơn và cung cấp cho bạn đọc cơ hội nhận biết chúng. Chẳng hạn,

```

static boolean tooCold (int temp)

// EFFECTS: Trả về true nếu temp là  $\leq 0$  độ F;
// ngược lại là false

```

```

static boolean tooCold (int temp)

// EFFECTS: Trả về true nếu temp là  $\leq 0$  độ F;
// ngược lại là false, tức là trả về chính xác true khi temp không lớn hơn
// điểm đóng băng của nước ở nhiệt độ và áp suất chuẩn

```

Đặc tả thứ nhất đề xuất cho bạn đọc không có lý do nào để nghi ngờ, nhưng đặc tả thứ hai sẽ giống lên hồi chuông cho nhiều bạn đọc. Điều cảnh báo này tất nhiên sẽ dẫn đến việc phải điều chỉnh lại đặc tả.

Một trong những vấn đề chính với đặc tả không hình thức là mỗi bạn đọc có một kiến thức khác nhau khi đối đầu với nhiệm vụ đọc đặc tả. Dưa thêm dư thừa vào sẽ làm cho vấn đề được hiểu đúng hơn. Chẳng hạn, xét

```
static int billion ( )
```

// EFFECTS: Trả về số nguyên một tỷ

static int billion ()

// EFFECTS: Trả về số nguyên một tỷ, tức là 10^9 .

Cả người Anh và Mỹ đều thấy đặc tả thứ nhất không rõ ràng. Không may, họ hiểu điều đó theo các cách khác nhau. Ở Mỹ, billion là 10^9 ; còn ở Anh billion là 10^{12} . Nên đặc tả thứ hai, tuy có dư thừa, nhưng là tốt hơn.

Tại sao phải đặc tả

Đặc tả là cần thiết để đạt được tính cấu thành của chương trình. Trừu tượng được sử dụng để phân tách chương trình thành các modules. Một trừu tượng dường như là không thể nắm bắt được. Thiếu mô tả, chúng ta không có cách gì biết được chương trình cần thiết là gì, nó khác biệt như thế nào với các cài đặt khác. Đặc tả cung cấp mô tả này.

Một đặc tả cung cấp một thỏa thuận giữa nhà cung cấp và người sử dụng của một dịch vụ. Nhà cung cấp đồng ý viết một module mà thuộc về tập thỏa đặc tả. Người sử dụng đồng ý không dựa trên hiểu biết về thành viên nào của tập thỏa đặc tả này sẽ được cung cấp – đó là module không gì khác chính là cái được khẳng định bởi đặc tả. Thỏa thuận này làm cho có thể tách cài đặt khỏi việc sử dụng đơn vị chương trình. Các đặc tả cung cấp các bức tường lửa logic mà cho phép chia để trị thành công.

Các đặc tả rõ ràng là có ích để làm tài liệu chương trình. Mỗi hành động viết đặc tả đều đem lại lợi ích, vì nó làm rõ thêm trừu tượng được đặc tả. Viết đặc tả hầu như luôn dạy chúng ta cái gì đó quan trọng về tập thỏa đặc tả được mô tả. Trong một số trường hợp, sự hiểu biết tốt hơn như vậy là kết quả quan trọng nhất của nỗ lực đặc tả.

Mục đích viết đặc tả là cả hai, vừa hạn chế đủ và tổng quan đủ. Như vậy, chúng ta nhấn mạnh đặc biệt đến các ngoại lệ yêu cầu và các điều kiện biên. Làm điều đó trong khi luôn đặt ra câu hỏi về hành vi của trừu tượng, các câu hỏi về *IndexString* như, sẽ làm gì nếu một xâu nào đó là rỗng. Luôn hỏi và trả lời các câu hỏi như vậy buộc chúng ta phải suy nghĩ cẩn thận về trừu tượng và việc sử dụng nó.

Việc xây dựng đặc tả tập trung sự chú ý vào cái gì được yêu cầu đối với một chương trình. Nó phục vụ như một cơ chế để sinh ra các câu hỏi, mà cần phải được trả lời khi trao đổi với người sử dụng một chương trình hay một module, hơn là bởi những người cài đặt. Bằng việc khuyến khích hỏi các câu hỏi này ở giai đoạn sớm phát triển hệ thống, đặc tả sẽ giúp chúng ta điều chỉnh hiểu biết của chúng ta về yêu cầu và thiết kế hệ thống, trước khi chúng ta bắt đầu cài đặt.

Đặc tả cần được viết sớm, như việc lưu các quyết định đã được thực hiện. Vì đặc tả trở thành không thích hợp chỉ khi trừu tượng của họ là không dùng được, họ cần tiếp tục suy xét trong quá trình phát triển chương trình. Là lỗi quan trọng nếu coi quá trình viết đặc tả là pha tách biệt của dự án phần mềm.

Khi viết xong, các đặc tả có thể phục vụ nhiều mục đích khác nhau. Chúng có ích cho người thiết kế, cài đặt, cũng như người bảo trì. Trong pha cài đặt chu trình sống phần mềm, việc có đặc tả tốt giúp cả cài đặt các module chuyên biệt và cài đặt các modules sử dụng nó. Như đã nói, đặc tả tốt là kết quả cuộc chiến cân bằng giữa tính hạn chế và tính tổng quan. Nó nói cho người cài đặt, dịch vụ gì cần cung cấp, nhưng không áp đặt bất cứ ràng buộc không cần thiết nào về việc dịch vụ được cung cấp như thế nào. Bằng cách như vậy, nó cho phép người cài đặt có nhiều tính mềm dẻo mà phù hợp với nhu cầu của người sử dụng. Tất nhiên đặc tả là cốt yếu cho người sử dụng, người mà ngược lại sẽ không có cách nào khác để biết được họ có thể dựa trên cái gì trong khi cài đặt các modules. Không có đặc tả, tất cả những gì có, chỉ là mã chương trình, không rõ ràng code này thay đổi nhiều như thế nào với thời gian. Khi kiểm thử, đặc tả cung cấp thông tin mà cần được sử dụng để sinh dữ liệu kiểm thử và xây dựng các kịch bản mô phỏng module chuyên biệt. Khi lỗi xuất hiện, đặc tả được sử dụng để chỉ ra lỗi nằm ở đâu. Hơn nữa nó xác định các ràng buộc mà cần phải quan sát thấy trong khi hiệu chỉnh lỗi, mà giúp chúng ta tránh đưa các lỗi mới vào khi hiệu chỉnh lỗi cũ.

Cuối cùng, một đặc tả có thể là một công cụ bảo trì hữu ích. Việc có một tài liệu chính xác rõ ràng là điều kiện tiên quyết để bảo trì hiệu quả. Chúng ta cần biết module làm gì và nếu nó là phức tạp, thì nó làm điều đó như thế nào. Hai khía cạnh này của tài liệu là hòa quyện vào nhau. Sử dụng đặc tả như tài liệu giúp cho giữ nó riêng biệt và làm cho nó dễ dàng thấy sự phân nhánh của các sửa đổi được đề xuất. Chẳng hạn, một sửa đổi đề xuất, mà yêu cầu chúng ta chỉ cài đặt lại một trừu tượng, không thay đổi đặc tả có tác động nhỏ hơn nhiều cái mà thay đổi đặc tả.

Tóm lại, đặc tả có hai mục đích chính:

- Thứ nhất, viết đặc tả sẽ làm sáng tỏ trừu tượng mà được đặc tả bằng cách tập trung vào các tính chất của trừu tượng. Việc sử dụng chúng cần được chú ý đến các điều kiện ràng buộc.
- Thứ hai là đặc tả được sử dụng như tài liệu. Đặc tả có giá trị trong mọi khâu của quá trình phát triển phần mềm, từ thiết kế đến bảo trì. Đặc tả mô tả một module làm gì, chứ nó không nói đến module được cài đặt như thế nào. Một cài đặt tốt có thể dễ dàng được thay thế, mà không ảnh hưởng đến các module sử dụng nó.

CHƯƠNG 2: ĐẶC TẢ YÊU CẦU

Tiếp nối phần phân tích yêu cầu, chúng ta sẽ thể hiện kết quả của quá trình đó qua tài liệu yêu cầu mà thể hiện sự hiểu biết về sản phẩm sẽ được phát triển nhằm đáp ứng yêu cầu của khách hàng. Tài liệu quan trọng nhất của tài liệu yêu cầu là đặc tả yêu cầu.

Như chúng ta đã thấy, một chương trình là đối tượng dữ liệu, và đặc tả của nó sẽ giống với những gì chúng ta đã biết về các kiểu trùu tượng. Đặc tả cho các kiểu trùu tượng dựa trên cái nhìn tổng quan để định nghĩa mô hình cho các trạng thái của các đối tượng của chúng. Trong các đặc tả này, chúng ta đã có thể sử dụng các mô hình đơn giản mà dựa trên một tập nhỏ các khái niệm toán học. Đối với đặc tả yêu cầu, tuy nhiên, các mô hình đơn giản như vậy là chưa đủ. Chúng ta cần mô hình trạng thái của toàn bộ chương trình. Trạng thái này thường có cấu trúc phức tạp, ngay cả khi chúng ta hạn chế những gì mà liên quan đến phần của cấu trúc mà nhìn thấy được đối với người sử dụng chương trình. Chẳng hạn, người sử dụng của hệ thống file cần hiểu về các file và các thư mục và chúng kết nối với nhau như thế nào.

Như vậy chúng ta cần một cách nào đó để mô tả trạng thái của chương trình. Điều này có thể được thực hiện một cách phi hình thức như trong ngôn ngữ tiếng Việt, nhưng những định nghĩa như vậy sẽ không chính xác và dài dòng. Do đó chúng ta sẽ sử dụng một kỹ thuật khác: chúng ta định nghĩa chương trình bằng phương tiện của *mô hình dữ liệu*. Mô hình này được sử dụng trong đặc tả yêu cầu.

Kết quả là một định nghĩa chính xác chấp nhận được về những gì yêu cầu. Định nghĩa như vậy là cơ sở tốt để thiết kế chương trình, từ bây giờ người thiết kế sẽ hiểu được cái gì cần xây dựng. Tiếp theo, phần thực hành định nghĩa mô hình và sau đó là sử dụng nó để viết đặc tả là phần có giá trị của quá trình phân tích yêu cầu, vì nó đưa ra các vấn đề cần quan tâm khi phân tích. Điều này dẫn đến một đặc tả mà phản ánh suy nghĩ cẩn thận hơn về các vấn đề đó và nó dường như đáp ứng được các yêu cầu của khách hàng như là các kết quả mong muốn.

2.1 Mô hình dữ liệu

Một mô hình dữ liệu bao gồm một đồ thị và một mô tả văn bản.

Các thành phần của một mô hình dữ liệu. Một mô hình dữ liệu bao gồm:

- Các định nghĩa các miền, các tập con và các quan hệ.
- Một đồ thị mà chỉ ra các tập hợp và các quan hệ tương quan với nhau như thế nào và định nghĩa các ràng buộc trên chúng.
- Các định nghĩa của các quan hệ suy diễn và các ràng buộc bổ sung.

Đồ thị định nghĩa các kiểu dữ liệu mà được thao tác và chúng có quan hệ với nhau như thế nào. Đồ thị và mô tả văn bản cùng nhau định nghĩa các ràng buộc mà chương trình phải tuân theo chúng. Định nghĩa các ràng buộc này làm cho chúng ta tập trung vào các chi tiết mà được xem xét kỹ lưỡng suốt quá trình phân tích yêu cầu.

Đồ thị có các đỉnh và các cạnh. Các đỉnh thể hiện các kiểu dữ liệu mà được thao tác bởi chương trình. Mỗi đỉnh là một tập các phần tử được đặt tên. Chẳng hạn, mô hình dữ liệu cho hệ thống file sẽ chứa đỉnh File, thể hiện các file, và đỉnh Dir, thể hiện các thư mục. Mỗi tập chứa mọi phần tử mà tồn tại ở một thời điểm cụ thể nào đó (chẳng hạn mọi file tồn tại ở thời điểm đó).

Các vật trong các tập hợp là không có cấu trúc; không có chi tiết nào cho chúng, ngoại trừ có mối quan hệ đến các tập khác. Các cạnh thể hiện các quan hệ này. Mục đích của đồ thị là cung cấp một cơ chế nhìn thấy được thuận tiện để chỉ ra các quan hệ. Ký hiệu này cũng cho phép các quan hệ được ràng buộc theo cách mà sẽ được giải thích phần sau. Như vậy, một đồ thị biểu diễn các kiểu bất biến nào đó.

Mô hình này mô tả trạng thái của hệ thống. Trạng thái này có thể thay đổi qua thời gian (chẳng hạn, như kết quả của chương trình đáp ứng yêu cầu của người sử dụng). Mô hình thể hiện sự thay đổi này theo hai cách. Trước hết, các tập tự nó có thể thay đổi: như trạng thái chương trình thay đổi, các vật có thể được bổ sung hoặc loại bỏ khỏi các tập hợp. Thứ hai, các quan hệ giữa các tập hợp có thể thay đổi.

2.1.1 Các tập hợp con

Một số tập hợp được thể hiện bằng các đỉnh trong đồ thị là các tập con của các tập hợp khác. Chúng ta sẽ gọi các tập hợp mà không có các tập hợp cha trên đó là *các miền* (*domains*). Mỗi miền là rời nhau đối với mọi miền khác.

Cạnh tập con dùng để chỉ ra rằng một tập là tập con của tập khác. Chúng ta biểu diễn thông tin này bằng mũi tên với đầu tam giác đóng. Mũi tên đi từ tập con đến tập cha. *Đầu mũi tên chỉ ra các tập con đó có chứa mọi phần tử của tập cha của nó (đầu mũi tên tô đặc) hoặc chỉ một số phần tử của tập cha (đầu mũi tên không tô đặc)*.

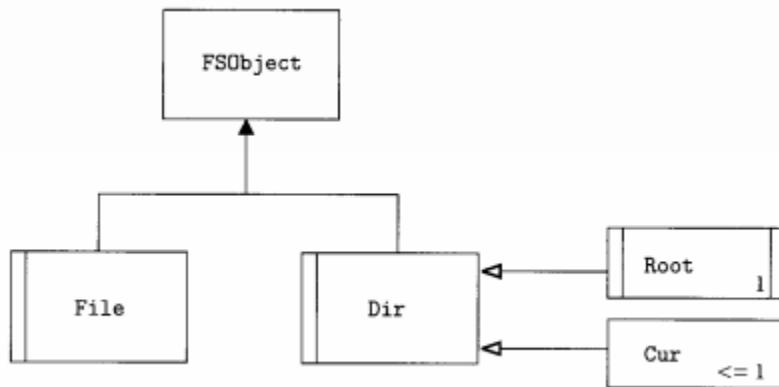
Các tập con có thể chia sẻ một mũi tên tam giác đóng; trong trường hợp này, chúng là rời nhau, và mũi tên tô đặc hay không chỉ ra hợp của chúng có phủ kín tập cha hay không. Các tập con không chia sẻ mũi tên có thể không cần thiết là rời nhau.

Ba ràng buộc là có ích để định nghĩa các tập con. Trước hết, các tập con có thể đôi khi là cố định. Điều này có nghĩa là thành viên của tập con này là cố định cho mọi thời điểm; tập con đó không bao giờ có thêm hoặc bớt đi các phần tử. *Một tập con cố định được chỉ rõ bằng hai gạch kép ở hai phía của đỉnh đó*.

Thứ hai, một tập con có thể là *tĩnh*. Điều này có nghĩa là thành viên của nó được xác định tĩnh; *một phần tử không bao giờ thay đổi chuyển từ thuộc tập con thành không thuộc tập con*. Chúng ta sẽ ký hiệu tập con tĩnh bằng gạch kép ở phía bên trái của đinh. Lưu ý rằng một tập con là cố định, thì sẽ là tĩnh.

Thứ ba, đôi khi là có ích khẳng định tường minh, một tập con chứa bao nhiêu phần tử. Điều này có thể được thực hiện bằng cách viết kích thước của tập con bên trong hộp đối với đinh đó; chẳng hạn, “1” có nghĩa là một tập con chứa chính xác một phần tử, trong khi “ $<= 1$ ” có nghĩa là tập con hoặc là rỗng hoặc là chứa một phần tử.

Hình 2.1 chỉ ra một số quan hệ và ràng buộc tập con cho hệ thống File. Một đối tượng của hệ thống File *FSObject* có thể hoặc là một file hoặc là một thư mục. Hai tập con này là rời nhau và bao phủ tập cha của chúng. Tiếp theo, chúng là tĩnh: một đối tượng của hệ thống file không thể chuyển đổi từ thư mục sang file hoặc ngược lại.



Hình 2.1 Đồ thị bộ phận cho hệ thống file

Ngoài ra, có hai tập con thứ vị của *Dir*. *Root* thể hiện thư mục gốc; thư mục này là cố định với mọi thời điểm (và do đó đinh của nó được đánh dấu là cố định), và có chính xác một thư mục gốc. *Cur* thể hiện thư mục hiện thời. Chỉ có nhiều nhất một thư mục hiện thời, và thư mục khác bất kỳ có thể được chọn là thư mục hiện thời khi hệ thống file đang sử dụng. Lưu ý rằng vì *Root* và *Cur* không chia sẻ một mũi tên, nên chúng không cần thiết phải là rời nhau; và thư mục gốc có thể trở thành thư mục hiện thời.

2.1.2 Các quan hệ

Các quan hệ được sử dụng để chỉ ra các phần tử của một tập hợp là liên quan đến các phần tử của tập hợp khác như thế nào. Chúng được biểu diễn bằng mũi tên có đầu mở. Mỗi mũi tên quan hệ được đánh dấu với một cái tên. Mỗi quan hệ có một gốc (là đinh mà nó đi ra từ đó) và một đích (là đinh mà được trỏ đến). Chẳng hạn, có thể có mũi tên *contents* từ đinh *DirEntry* (thể hiện các bản nhập trong các thư mục) đến đinh *FSObject*, chỉ ra rằng

một bản nhập thư mục sẽ nêu tên một đối tượng của hệ thống file. Ở đây *DirEntry* là gốc, và *FSObject* là đích.

Tên quan hệ không cần là duy nhất trong sơ đồ, ngoại trừ là nếu hai quan hệ có cùng một nguồn, thì chúng cần có tên khác nhau.

Đôi khi là hữu ích định nghĩa phép ngược của một quan hệ. Chẳng hạn, chúng ta có một quan hệ *parent* mà ánh xạ một thư mục vào thư mục cha của nó (mỗi thư mục ngoại trừ thư mục gốc *Root* có thư mục cha). Nhưng cũng là có ích định nghĩa quan hệ *children* mà là ngược của *parent*: nếu thư mục d1 là *parent* của d2, thì d2 là *child* của d1. Một mũi tên duy nhất có thể biểu diễn cả hai: quan hệ và ngược của nó. Nhấn trên mũi tên đặt tên cho quan hệ chính *primary* và quan hệ ngược. Chúng ta sử dụng r1 (~r2) có nghĩa là r1 là quan hệ chính và r2 là quan hệ ngược của nó. Do đó, đối với hệ thống file chúng ta có *parent* (~*children*).

Đòi thị cũng định nghĩa số lượng và tính thay đổi của các quan hệ. Số lượng định nghĩa số các phần tử mà quan hệ ánh xạ đến hoặc từ đó. Một quan hệ thực tế ánh xạ vào một tập hợp: nó ánh xạ một phần tử từ nguồn vào một tập các phần tử ở đích. Số lượng của quan hệ định nghĩa có bao nhiêu phần tử ở trong tập hợp đó. Chẳng hạn, quan hệ *parent* ánh xạ vào nhiều nhất một thư mục (vì gốc root không có cha, và mọi thư mục khác có đúng một cha). Số lượng được chỉ ra bởi đánh dấu mũi tên quan hệ với một trong những ký hiệu sau:

!: nghĩa là chính xác một

?: nghĩa là không hoặc một

***: nghĩa là không hoặc nhiều hơn

+: nghĩa là một hoặc nhiều hơn

Các ký hiệu số lượng cần xuất hiện trên cả hai đầu của mũi tên. Ký hiệu ở đầu đích giải thích có bao nhiêu phần tử ở trong tập đó mà một phần tử cụ thể ở nguồn ánh xạ vào. Chẳng hạn, mỗi đối tượng bản nhập thư mục được ánh xạ bởi *contents* vào một đối tượng duy nhất của hệ thống file, và do đó đầu đích của mũi tên *contents* được đánh dấu với *!*. Nay giờ chúng ta xem xét đầu nguồn của mũi tên này. Đơn giản chúng ta đảo quan hệ này, xét nó như một ánh xạ từ đích đích đến đích nguồn, và chúng ta sẽ đưa ra câu hỏi: có bao nhiêu phần tử nguồn trong tập đó được ánh xạ vào một phần tử đích duy nhất? Chẳng hạn, một đối tượng hệ thống file có thể được tham chiếu đến bởi một số bản nhập thư mục, nhưng thư mục gốc root không được tham chiếu bởi bất cứ bản nhập thư mục nào. Do đó chúng ta có thể đánh dấu đầu nguồn của mũi tên *contents* là ***.

Các quan hệ có thể là thay đổi được hoặc không thay đổi được. Một quan hệ được gọi là thay đổi được, nếu ánh xạ đó có thể thay đổi – chẳng hạn, phần tử nguồn có thể được gắn kết với một đích khác ở một thời điểm trong tương lai. Tính thay đổi cũng được áp dụng

tại cả hai đầu của mũi tên: nếu đích gắn kết với phần tử nguồn có thể thay đổi, đầu đích là thay đổi được, trong khi phần tử nguồn gắn kết với đích có thể thay đổi, đầu nguồn là thay đổi được. Người ta dùng ký hiệu | đặt trước mũi tên nói lên nguồn không thể thay đổi và | đặt sau mũi tên nói lên đích không thể thay đổi.

Sau đây là một số ví dụ thể hiện các ràng buộc này, sử dụng biểu diễn văn bản để chỉ ra cả hai số lượng và tính thay đổi:

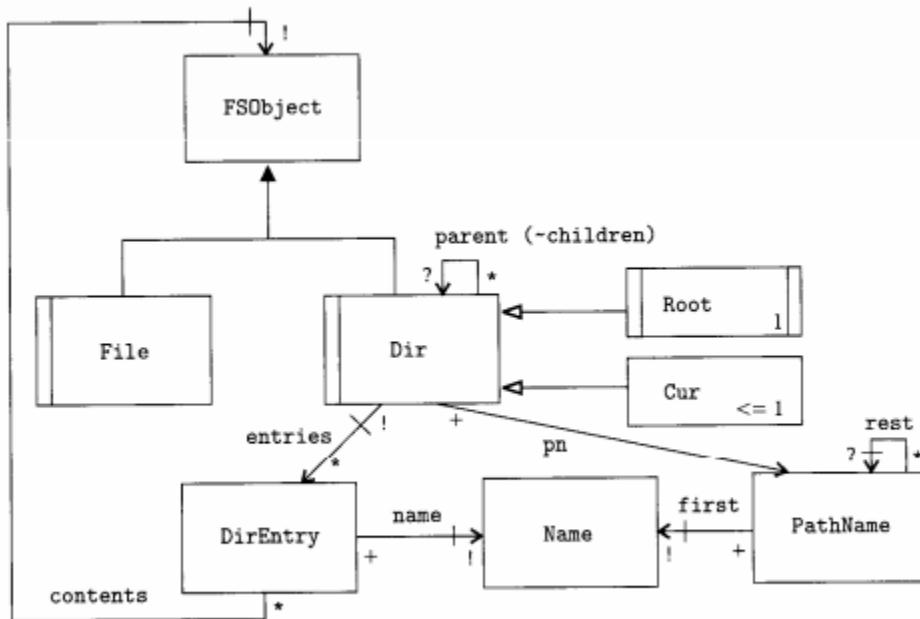
location: Airport ? →| Location !

*socialSecurityNumber: Person ! | → SSN**

*parent (~children): Dir * → Dir ?*

Các quan hệ này có thể được giải thích như sau. Mỗi sân bay *airport* được đặt ở một vị trí riêng biệt, nhưng nhiều vị trí không được gắn kết với sân bay nào cả. Cũng như vậy sân bay không thể dịch chuyển, nhưng một sân bay mới có thể được xây dựng ở một vị trí nào đó trong một thời điểm tương lai. Mỗi người có thể có nhiều số an ninh xã hội và có thể được bổ sung về sau, nhưng bất kỳ một số an ninh xã hội nào cũng được gắn kết với một người duy nhất và sự gắn kết không thể thay đổi. Cuối cùng, một thư mục có nhiều nhất một cha, nhưng có thể là một cha của nhiều thư mục; tiếp theo, cả hai gắn kết này có thể thay đổi. Lưu ý rằng trong trường hợp quan hệ ngược này, chúng ta cho ràng buộc đối với quan hệ chính, nhưng nó cũng lập tức định nghĩa chúng cho cả gắn kết ngược.

Hình 2.2 đưa ra mô hình dữ liệu đơn giản cho hệ thống *file*. Bổ sung thêm vào các tập hợp đã có như *FSObject*, *File*, *Dir*, *Root*, và *Cur*, chúng ta có *DirEntry* mà mô hình các bản nhập trong các thư mục; *Name*, mà mô hình các tên được sử dụng trong các thư mục để đặt tên cho các *files* hoặc các thư mục, và *PathName* mà mô hình tên đường dẫn đến các file hoặc các thư mục.



Hình 2.2 Đồ thị cho hệ thống file

Ở đây cũng có một số quan hệ trong mô hình ngoài quan hệ *parent* và ngược của nó là *children*, mà đã bàn ở trên. Quan hệ *entries* ánh xạ thư mục vào các bản nhập của nó. Một bản nhập *entry* là một gán kết giữa một tên và một đối tượng file của hệ thống; *name* ánh xạ một bản nhập *entry* vào một tên của nó, trong khi *contents* ánh xạ bản nhập vào một đối tượng file hệ thống mà nó tham chiếu đến. Tiếp theo, *first* chọn tên đầu tiên trong một đường dẫn, và *rest* tạo ra phần còn lại của đường dẫn sau khi tên đầu tiên được tách ra.

Sơ đồ đó cũng chỉ ra các quyết định mà chúng ta đã thực hiện về các ràng buộc trên các quan hệ. Chẳng hạn, một thư mục chứa không hoặc nhiều hơn các bản nhập, và các bản nhập không được chia sẻ giữa các thư mục. Mỗi bản nhập chứa chính xác một tên và một đối tượng của hệ thống file, nhưng cùng một tên hoặc một đối tượng của hệ thống file có thể xuất hiện trong nhiều bản nhập. Tiếp theo, mỗi bản nhập *entry* là không thay đổi theo nghĩa: ánh xạ của nó đến *Name* và *FSObject* không thể thay đổi. Điểm cuối cùng phản ánh quyết định là: người sử dụng không thể thay đổi gán kết của một tên bên trong một bản nhập thư mục. Tuy nhiên, họ có thể thay một bản nhập với một bản mới chứa một gán kết khác (vì bản nhập là thay đổi được).

Đánh dấu các quan hệ buộc chúng ta suy nghĩ về các vấn đề mà có thể phải xem xét thêm, nhưng là quan trọng đối với các yêu cầu. Chẳng hạn, trong quyết định ký hiệu gán kết nào là đánh dấu dành cho quan hệ *parent*, chúng ta cần phải quyết định *parent* của một thư mục

có thay đổi được không. Chúng ta đã quyết định là nó có thể thay đổi được (vì chúng ta không đánh dấu là quan hệ này là không thay đổi được).

Các quan hệ ánh xạ các phần tử nguồn vào các phần tử đích không cần các đối số bổ sung. Khi các đối số cần, thì có hai cách kiểm soát chúng. Trước hết, chúng ta có thể đưa vào một đỉnh bổ sung mà gắn kết đối số với kết quả một cách hiệu quả. Đó là cái chúng ta đã làm khi chúng ta bổ sung đỉnh *DirEntry*; chúng ta sử dụng nó để mô hình ánh xạ từ một tên trong một thư mục cụ thể vào một đối tượng hệ thống *file*. Cách tiếp cận khác là sử dụng đệ qui. Chẳng hạn, chúng ta có thể nghĩ về đường dẫn như một dãy các tên, nhưng mô hình nó có thể yêu cầu một quan hệ mà nhận các số nguyên như các đối số. Trong trường hợp này, chúng ta sẽ sử dụng đệ qui, coi đường dẫn như một danh sách móc nối thay cho một dãy.

Biểu diễn ràng buộc như một phần của định nghĩa lược đồ là rất có ích, bởi vì quá trình đó là rất có phương pháp: chúng ta đơn giản xem xét từng đỉnh tập con và từng quan hệ. Đối với mỗi quan hệ, chúng ta quyết định đối với mỗi đầu số lượng là bao nhiêu và có thay đổi được không. Tương tự đối với mỗi tập con, chúng ta quyết định về tính thay đổi của nó (là cố định, tĩnh hay không ràng buộc), và kích thước của nó. Tuy nhiên, đồ thị thường không bao quát hết các ràng buộc mà mô hình cần phải đáp ứng. Các ràng buộc bổ sung này sẽ được định nghĩa bằng văn bản.

2.1.3 Thông tin văn bản

Thông tin văn bản gồm hai phần. Phần đầu đơn giản giải thích ý nghĩa mong muốn của mỗi tập và quan hệ.

Domains

FSObject: mọi files và directories trong hệ thống File

File: mọi files trong hệ thống File

Dir: mọi directories trong hệ thống File

Root: thư mục gốc

Cur: thư mục hiện tại

DirEntry: các bản nhập trong thư mục

Name: các tên dạng xâu bên trong bản nhập thư mục

PathName: đường dẫn đối với thư mục và files

Quan hệ:

parent (~children): cho thư mục cha của một thư mục
entries: cho các bản nhập (cấp tên/đối tượng của hệ thống file)
name: cho tên của một thư mục hay file bên trong một bản nhập thư mục
contents: cho một đối tượng của hệ thống file gắn kết với một tên bên trong một thư mục
first: cho tên ở đầu một đường dẫn
rest: cho phần đuôi của một đường dẫn
pn: cho tên đường dẫn của mọi đường bắt đầu từ một thư mục

Hình 2.3 Mô tả các tập hợp và các quan hệ cho hệ thống files.

Hình trên chỉ ra thông tin cho một hệ thống file. Lưu ý rằng các giải thích trên không có trên đồ thị, như các quan hệ tập con giữa các tập hợp.

Phần thứ hai định nghĩa các ràng buộc bổ sung. Ở đây có hai dạng ràng buộc. Thứ nhất, một số ràng buộc là suy diễn. Một quan hệ suy diễn là quan hệ mà được định nghĩa trên các thuật ngữ của quan hệ khác. Quan trọng là chỉ ra được các quan hệ suy diễn, vì nó giảm kích thước của mô hình, làm cho mọi người dễ hiểu nó hơn. Ngoài ra, định danh các quan hệ suy diễn giảm số các ràng buộc bổ sung. Các quan hệ suy diễn sẽ tự động bị ràng buộc bởi các ràng buộc trên các quan hệ định nghĩa chúng và ngược lại.

Cách nhận biết các quan hệ suy diễn là xem xét từng quan hệ một và xem nó có được định nghĩa qua các quan hệ khác không. Đối với mô hình hệ thống file, sự phân tích như vậy dẫn đến định danh hai quan hệ suy diễn. Quan hệ thứ nhất là *parent (~ children)*: cha một thư mục là một thư mục mà chứa một bản nhập cho nó:

Cha một thư mục là một thư mục mà chứa một bản nhập cho nó

*Với mọi d: Dir [d.parent = { d2 / d2 trong Dir && có tồn tại e: DirEntry
(e trong d2.entries && d = e.contents) }]*

Định nghĩa trên sử dụng ký hiệu viết gọn: khi một tập hợp chỉ có một phần tử, chúng ta sử dụng tên của tập hợp để chỉ tên của phần tử đó. Như vậy *e.contents* là được sử dụng ký hiệu thư mục mà là phần tử duy nhất của tập đó.

Lưu ý *d.parent* được định nghĩa là một tập hợp, điều đó tất nhiên là cần thiết vì mỗi quan hệ ánh xạ vào một tập hợp. Tiếp theo, tập này cần có nhiều nhất một phần tử, vì đồ thị ràng buộc quan hệ *parent* ánh xạ vào một tập hợp chứa nhiều nhất một phần tử. Như vậy, một

khi chúng ta đã xác định được quan hệ *parent* là được suy diễn từ các quan hệ khác, điều này được áp dụng cho chúng, và do đó chúng ta biết rằng một thư mục có thể được chứa trong nhiều nhất một thư mục khác.

Quan hệ suy diễn thứ hai là *pn*:

d.pn chứa mọi đường dẫn mà đi ra từ thư mục *d*

Với mọi *d*: *Dir* [*d.pn* = { *p* / *p* thuộc *Pathname* &&
mà tồn tại *e*: *DirEntry* (*e* thuộc *d.entries* &&
p.first = *e.name* &&
(*p.rest* = { } // (*e.contents* in *Dir* &&
p.rest thuộc *e.contents.pn*))) }]

Bây giờ chúng ta đã sẵn sàng định nghĩa các ràng buộc. Giả sử bắt đầu bằng ràng buộc cấu trúc của phân cấp thư mục. Chúng ta muốn biểu diễn sự kiện rằng thư mục gốc là tổ tiên của mọi thư mục trừ chính nó và ở đây không có chu trình trong dãy quan hệ tổ tiên. Để nắm bắt chính xác ràng buộc này, sẽ là hữu ích khi định nghĩa hàm hỗ trợ sau:

Một tổ tiên của thư mục là cha của nó và tổ tiên của cha của nó
ancestors (*d*) = if *d* = Root then { }
else *d.parent* + *ancestors* (*d.parent*)

Ở đây phép + ký hiệu là phép hợp tập hợp. Chúng ta sử dụng ký hiệu viết tắt *Root* là thư mục gốc gồm một phần tử duy nhất của nó.

Cho trước định nghĩa trên, chúng ta có thể khẳng định như sau:

Một thư mục không là tổ tiên của chính nó và mỗi thư mục trừ *Root* có *Root* như một tổ tiên *ancestor* của nó

Với mọi *d*: *Dir* [!(*d* thuộc *ancestors(d)*) &&
(*d* = *Root* // *Root* thuộc *ancestors(d)*)]

Ràng buộc này là thú vị, vì nó cho chúng ta biết một hệ thống file chứa chỉ một thư mục gốc và các thư mục khác được truy cập từ thư mục gốc này. Một ràng buộc tương tự hạn chế hạn chế các files được chứa bởi hệ thống:

Mỗi file có một bản nhập trong một thư mục nào đó

Với mọi *f*: *File* [có tồn tại *d*: *Dir*, *e*: *DirEntry*
(*e* thuộc *d.entries* && *f* = *e.contents*)]

Các ràng buộc này cùng nhau nói lên rằng, chỉ các đối tượng file mà tồn tại là được truy cập bởi một đường dẫn từ thư mục gốc. Đối với một người sử dụng, điều này có nghĩa là các file và các thư mục mà không được dẫn từ thư mục gốc là không được sử dụng; đối với người cài đặt, điều này có nghĩa là không cần thiết cung cấp lưu trữ các đối tượng không dẫn được.

Chúng ta cũng muốn ràng buộc nội dung của các thư mục.

Một thư mục chứa nhiều nhất một bản nhập với một tên cho trước

Với mọi $d: Dir, e1, e2: DirEntry [$

$e1, e2 \in d.entries \&& e1.name = e2.name \rightarrow e1 = e2]$

Một thư mục có thể chứa nhiều nhất một bản nhập cho một thư mục con

Với mọi $d: Dir, e1, e2: DirEntry [e1, e2 \in d.entries \&&$

$e1.contents \in Dir \&& e1.contents = e2.contents$

$\rightarrow e1 = e2]$

Lưu ý rằng chúng ta phải viết các ràng buộc cả không hình thức bằng ngôn ngữ tự nhiên và cả hình thức bằng ký hiệu toán học. Như với trường hợp bất biến biểu diễn, chỉ chấp nhận các định nghĩa không hình thức, khi các định nghĩa đó là chính xác và dễ hiểu.

Tóm lại Hình 2.4 sau nêu các ràng buộc cho một hệ thống file. Các ràng buộc này không chứa tất cả mọi thứ mà đúng về hệ thống file. Chẳng hạn, điều sau là đúng: thư mục gốc không có cha và mỗi thư mục khác có đúng một cha. Tuy nhiên sự kiện này có thể được chứng minh từ các ràng buộc đã khảng định, chẳng hạn, ràng buộc về tổ tiên và ràng buộc số lượng trên quan hệ *parent*.

Quan hệ suy diễn và các hàm hỗ trợ:

Cha một thư mục là một thư mục mà chứa một bản nhập cho nó

Với mọi $d: Dir [d.parent = \{ d2 / d2 \in Dir \&& \text{có tồn tại } e: DirEntry$

$(e \in d2.entries \&& d = e.contents) \}]$

d.parent chứa mọi đường dẫn mà đi ra từ thư mục d

Với mọi $d: Dir [d.parent = \{ p / p \in Pathname \&&$

$\text{có tồn tại } e: DirEntry (e \in d.entries \&&$

$p.first = e.name \&&$

$(p.rest = \{ \} // (e.contents \in Dir \&&$

$p.rest \in e.contents.pn))) \} \}$

Một tổ tiên của thư mục là cha của nó và tổ tiên của cha của nó

$ancestors(d) = \text{if } d = \text{Root} \text{ then } \{ \}$

$\text{else } d.parent + ancestors(d.parent)$

Các ràng buộc:

Một thư mục không là tổ tiên của chính nó và mỗi thư mục trừ Root có Root như một tổ tiên ancestor của nó

Với mọi $d: Dir$ [$!(d \in ancestors(d)) \&&$

$(d = \text{Root} \text{ || Root} \in ancestors(d)) \]$

Mỗi file có một bản nhập trong một thư mục nào đó

Với mọi $f: File$ [có tồn tại $d: Dir, e: DirEntry$

$(e \in d.entries \&& f = e.contents) \]$

Một thư mục chứa nhiều nhất một bản nhập với một tên cho trước

Với mọi $d: Dir, e1, e2: DirEntry$ [

$e1, e2 \in d.entries \&& e1.name = e2.name \rightarrow e1 = e2 \]$

Một thư mục có thể chứa nhiều nhất một bản nhập cho một thư mục con

Với mọi $d: Dir, e1, e2: DirEntry$ [$e1, e2 \in d.entries \&&$

$e1.contents \in Dir \&& e1.contents = e2.contents$

$\rightarrow e1 = e2 \]$

Hình 2.4 Các ràng buộc cho hệ thống file

Dễ bị quên hơn về các ràng buộc văn bản so với những cái được biểu diễn trực tiếp trên đồ thị, vì không có phương pháp nào hỗ trợ chúng. Thay vào đó, bạn cần suy nghĩ về mọi nhóm quan hệ và có hay không các ràng buộc khác cần cho chúng.

2.2 Đặc tả yêu cầu

Đặc tả yêu cầu là mô tả chương trình tương tác như thế nào với các trạng thái của nó để thực hiện các thao tác, ở đó mỗi thao tác tương ứng với một yêu cầu của người sử dụng. Chẳng hạn, đặc tả hệ thống file cần giải thích cái gì xảy ra với các file và các thư mục, khi người sử dụng đọc và viết file và truy cập hoặc thay đổi các thư mục.

Tóm tắt nội dung Đặc tả yêu cầu:

- Một đặc tả yêu cầu chứa các đặc tả cho các thao tác tĩnh và động.
- Nếu đặc tả là dành cho một chương trình tương tác, các thao tác được hạn chế đến các đối số xâu và không có kết quả, và các đặc tả của chúng có thể bao gồm mệnh đề check để xác định các điều kiện mà cần phải thỏa mãn cho thao tác vận hành.
- Các đặc tả thao tác dựa trên mô hình dữ liệu và chúng cần thỏa mãn các ràng buộc của nó.

Ở đây có hai loại thao tác. Thao tác tĩnh, bắt đầu từ khi ứng dụng chưa chạy. Thao tác động được sử dụng chỉ sau khi ứng dụng đã chạy. Thao tác tĩnh tương ứng với các phương thức tĩnh và hàm tạo, trong khi các thao tác động tương ứng chủ yếu với các phương thức cập nhật. Chẳng hạn, trong một hệ thống file, có các thao tác tĩnh để tạo hệ thống file mới hoặc khôi phục hệ thống file sau lỗi hệ thống và có các thao tác động tương tác với hệ thống file khi nó đang chạy, chẳng hạn, tạo và xóa thư mục hoặc tạo một thư mục cho thư mục hiện tại.

Trước khi chúng ta viết đặc tả, chúng ta cần xem xét hai vấn đề. Thứ nhất liên quan đến việc ứng dụng được dụng ý sử dụng như thế nào. Nếu các thao tác được chú ý gọi bởi chương trình, mỗi thao tác sẽ được định nghĩa là phương thức tĩnh, hàm tạo hoặc phương thức khởi tạo. Trong trường hợp này, đặc tả sẽ giống như những gì chúng ta đã thấy trước kia, ngoại trừ nó sẽ tham chiếu đến một mô hình, từ đó nó xác định trạng thái của các đối tượng trùu tượng.

Tuy nhiên, nếu một chương trình định hướng chỉ sử dụng tương tác, chúng ta sẽ không đặc tả nó theo cách thông thường. Trong trường hợp này, các thao tác bị hạn chế. Chúng chỉ cho phép có các đối số xâu, nhưng chúng không có các kiểu đối số khác và không có kết quả; thay vào đó kết quả của chúng được làm cho người sử dụng nhìn thấy được bởi một thay đổi nào đó trong giao diện người sử dụng. Tiếp theo đặc tả không cho thông tin chi tiết về các thao tác được sử dụng như thế nào thông qua giao diện người dùng; chúng ta tránh cho chi tiết này, vì chúng thường thay đổi (chẳng hạn, nếu ứng dụng được cung cấp cho nhiều hơn một nền tảng).

Một chương trình đôi khi được dự kiến sử dụng tương tác và được gọi bởi một chương trình khác. Trong trường hợp này, chúng ta cần cung cấp đặc tả cho cả hai: các thao tác tương tác và các phương thức thông thường.

Vấn đề thứ hai liên quan đến việc lựa chọn định dạng dữ liệu bên ngoài để sử dụng trong việc trao đổi với người sử dụng; mọi định dạng như vậy sẽ là các ràng buộc trên các xâu. Chẳng hạn, chúng ta muốn định dạng cho tên và đường dẫn trong một hệ thống file. Đôi với một số ứng dụng nào đó, việc lựa chọn định dạng sẽ để lại tùy thuộc nhà cung cấp giao diện người sử dụng, nhưng nói chung sẽ là ý tưởng tốt xác định định dạng trong đặc tả yêu cầu để cung cấp nhất quán các giao diện người sử dụng trong hệ thống.

Hình 2.5 sau định nghĩa một số định dạng đơn giản cho một hệ thống file và sau đó thể hiện các đặc tả của một số thao tác hệ thống file đơn giản mà dự kiến được sử dụng từ giao diện người sử dụng. Chúng ta sử dụng ký hiệu và các qui tắc khác trong các đặc tả này, vì các thao tác này là hơi khác so với các phương thức thông thường. Thứ nhất, các thao tác hầu như thay đổi trạng thái, và do đó chúng ta bỏ qua mệnh đề thay đổi *modifies*, vì nó mang rất ít thông tin. Thứ hai, các thao tác này cần là tổng thể, nếu có vấn đề, chúng cần tìm ra và cảnh báo người sử dụng. Vì các thao tác này là tổng thể, các đặc tả của chúng sẽ không chứa các mệnh đề yêu cầu *Requires*. Tuy nhiên chúng sử dụng một dạng mệnh đề mới là kiểm tra *check*. Mệnh đề này xác định các điều kiện mà thao tác cần kiểm tra; nếu các điều kiện này không thỏa mãn, thao tác cần đưa ra vấn đề cảnh báo người sử dụng thông qua giao diện người sử dụng. Chúng ta không đặc tả dạng tương tác với người sử dụng vì nó không giống với dạng của giao diện người sử dụng.

```
// Hạn chế định dạng
// NAME: Một xâu khác rỗng các ký tự in được mà không chứa ký tự /
// PATHNAME: Một dãy khác rỗng các NAMES được tách bởi / và bắt đầu
// hoặc với / (ý nghĩa là tên bắt đầu từ gốc) hoặc với một tên NAME, ý nghĩa là đường dẫn
// đó bắt đầu từ thư mục hiện tại.

// Các thao tác tĩnh
start()
    // EFFECTS: Tạo một hệ thống file mới bao gồm chỉ từ thư mục gốc và thư mục này là
    // rỗng.

    // Các thao tác động
makeDirInCur (String n)
    // CHECKS: NAME(n) và có thư mục c trong Cur và n không được định nghĩa trong c
    // EFFECTS: Tạo một thư mục mới và nhập nó với tên n trong c.

makeCurFromRoot (String p)
    // CHECKS: p là một đường dẫn đi từ root đến một thư mục d
    // EFFECTS: tạo d là thư mục hiện tại.

deleteDir ()
    // CHECKS: Có một thư mục c trong Cur và c là rỗng và không là gốc root.
    // EFFECTS: Loại bỏ bản nhập cho c từ cha của nó và đặt Cur = {}.
```

Hình 2.5 Các đặc tả một số thao tác của hệ thống file

Trong các đặc tả, các thao tác được chia thành hai loại thao tác tĩnh và động. Các đặc tả tham chiếu đến mô hình dữ liệu (chẳng hạn, đặc tả *makeDirInCur* tham chiếu đến *Cur*) và sử dụng các ràng buộc định dạng cho *Name* và *PathName*, sử dụng ký hiệu *NAME(n)* hoặc *PATHNAME(p)*; Các mệnh đề sẽ trả về *true* nếu đối số của chúng có định dạng đúng.

Mọi thao tác cần duy trì các ràng buộc của mô hình dữ liệu, bao gồm cả hai loại ràng buộc mà được định nghĩa bởi đồ thị và văn bản. Mỗi định nghĩa thao tác cần được kiểm tra và các tác động của nó lên trạng thái đảm bảo các ràng buộc được bảo trì; thông thường trong các phân tích như vậy, các ràng buộc cần được giả thiết là thỏa mãn ở trạng thái bắt đầu thực hiện thao tác. Trong phân tích, bắt đầu với mệnh đề kiểm tra, nó cần làm đủ các kiểm tra để suy ra các điều kiện mà nếu không thỏa mãn sẽ làm cho thao tác vi phạm điều kiện ràng buộc nào đó hoặc ngược lại sẽ không có ý nghĩa. Sau đó xem xét, các thay đổi được mô tả trong mệnh đề tác động *EFFECTS* sẽ dẫn đến trạng thái thỏa mãn các ràng buộc. Mệnh đề *EFFECTS* cần mô tả mọi thay đổi được thực hiện bởi thao tác ngoại trừ các thay đổi đến các quan hệ suy diễn. Các thay đổi đến các quan hệ suy diễn suy ra từ các thay đổi đến các quan hệ mà chúng phụ thuộc.

Chẳng hạn, *makeDirInCur* nhập một thư mục mới dưới tên *n* trong thư mục hiện tại. Nó kiểm tra để suy ra tình huống không lỗi, như vậy không có việc thư mục hiện tại hoặc tên đã được sử dụng. Thao tác sẽ kiểm tra các điều kiện này và cảnh báo người sử dụng nếu chúng xảy ra. Vì mệnh đề *EFFECTS* không khẳng định rằng *Cur* thay đổi, chúng ta có thể giả thiết rằng nó vẫn như vậy (tức là vẫn là thư mục c mà là thư mục hiện tại trước khi thao tác chạy, vẫn là thư mục hiện tại khi nó kết thúc). Tuy nhiên chúng ta cũng biết rằng *c.bn* thay đổi và c là *parent* của thư mục mới, vì các quan hệ này là suy diễn của các quan hệ mà thay đổi. Một ví dụ khác, xét thao tác *deleteDir*, và nhận thấy nó trả lời tường minh như thế nào cho câu hỏi điều gì đã xảy ra đối với *Cur*.

Nếu một thao tác không bảo trì một ràng buộc hoặc không giải thích tác động lên lên trạng thái một cách đầy đủ, đặc tả đó cần thay đổi hoặc một số ràng buộc trong mô hình cần định nghĩa lại. Trong trường hợp khác, tương tác với khách hàng có thể được yêu cầu để chỉ ra cần làm gì. Kết quả sẽ là một đặc tả giống như mô tả điều khách hàng muốn.

Đặc tả trong Hình 2.5 trên là không đầy đủ; một số thao tác khác cần bổ sung; vì một hệ thống file cũng được sử dụng từ các chương trình, cần có các đặc tả của các phương thức được sử dụng bởi các chương trình. Các phương thức này sẽ có các đặc tả mà tuân thủ các qui tắc thông thường của chúng ta: chúng cần có mệnh đề thay đổi *Modifies* nếu chúng thay đổi một điều gì đó; và chúng không có mệnh đề kiểm tra *checks*. Tuy nhiên, các đặc tả của chúng là tương tự với các đặc tả của các thành phần sao mà được sử dụng từ giao diện người sử dụng, với thay đổi tác động đến sự khác biệt rõ ràng (chẳng hạn, chúng nhận các kiểu khác nhau của đối số, và chúng sẽ trả về kết quả hoặc đưa ra ngoại lệ).

Bổ sung thêm cho đặc tả trừu tượng dữ liệu mà tương ứng với toàn bộ ứng dụng, chúng ta cần cho các đặc tả của bất cứ kiểu đối tượng nào mà bên ngoài được nhìn thấy theo nghĩa các đối tượng của chúng sẽ được sử dụng bởi các chương trình khác. Chẳng hạn, đặc tả yêu cầu cho một hệ thống file sẽ định nghĩa các kiểu file và thư mục, với tập đầy đủ các phương thức mà cho phép chúng được sử dụng thuận tiện. Các kiểu này sẽ tương ứng với một tập nào đó trong mô hình này, và mô hình có thể được sử dụng như hướng dẫn để định nghĩa các phương thức. Nhưng khi một người cần đi sâu vào tìm hiểu mô hình, vì cần suy nghĩ về nhu cầu người dùng trong tương tác với kiểu đang được định nghĩa. Các đặc tả cho các kiểu này sẽ giống các đặc tả của các kiểu trừu tượng thông thường không kể chúng có thể được viết theo các thuật ngữ của mô hình dữ liệu. Chẳng hạn, các đặc tả của một phương thức mà tạo ra một thư mục con của thư mục d có thể khẳng định rằng d là thư mục cha của thư mục mới.

2.3 Kiểm thử phần mềm

Dựa vào đặc tả chúng ta sẽ thông nhất với người sử dụng về hành vi của chương trình sẽ thực hiện và chúng ta cũng thỏa thuận với những người cài đặt về việc xây dựng chương trình có các hành vi mong muốn.

2.3.1 Các khái niệm

- Phê chuẩn là quá trình được thiết kế để gia tăng sự tin tưởng của chúng ta là chương trình làm việc theo đúng ý muốn. Nó cần được tiến hành thông qua kiểm chứng hoặc kiểm thử.
- Kiểm chứng là những lý lẽ hình thức hoặc không hình thức đảm bảo chương trình làm việc đúng trên mọi đầu vào có thể.
- Kiểm thử là quá trình chạy chương trình trên một tập các test case và so sánh kết quả thực tế với kết quả mong muốn.

Như vậy có hai cách để đi đến phê chuẩn. Chúng ta cần lập luận rằng chương trình làm việc đúng trên mọi đầu vào có thể. Việc này đòi hỏi suy luận cẩn thận về văn bản chương trình và nói chung được tham chiếu đến là “kiểm chứng”. Phê chuẩn chương trình hình thức nói chung là rất tốn kém với sự hỗ trợ của các công cụ tự động và hiện nay mới chỉ đạt được những sự hỗ trợ ban đầu. Do đó, hầu hết phê chuẩn chương trình đều là không hình thức. Ngay việc kiểm chứng không hình thức cũng là quá trình rất khó khăn.

Một cách khác ngoài kiểm chứng là kiểm thử. Chúng ta có thể dễ dàng tin tưởng là chương trình làm việc đúng trên một tập con nào đó dữ liệu đầu vào bằng cách chạy trên mỗi phần tử của tập con đầu vào đó và kiểm tra kết quả. Nếu tập mọi giá trị đầu vào nhỏ, thì kiểm thử tổng thể có thể được thực hiện. Tuy nhiên đối với hầu hết các chương trình, tập các đầu vào có thể là rất lớn, có thể là vô hạn, khi đó kiểm thử tổng thể là không thể thực hiện được. Cẩn thận chọn tập test case làm gia tăng sự tin tưởng của chúng ta là chương trình làm việc đúng như đặc tả. Nếu làm tốt, có thể phát hiện hầu hết các lỗi của chương trình.

2.3.2 Kiểm thử

Kiểm thử là quá trình chạy chương trình trên tập test case và so sánh kết quả thực tế với kết quả mong muốn. Mục đích của nó hướng tới phát hiện sự tồn tại các lỗi. Kiểm thử không chỉ ra được vị trí lỗi, điều đó được thực hiện thông qua **bắt lỗi** debugging. Khi chúng ta kiểm thử một chương trình, chúng ta sẽ kiểm tra quan hệ giữa đầu vào và đầu ra. Khi bắt lỗi, chúng ta cũng quan tâm quan hệ trên, nhưng ngoài ra còn tập trung kỹ hơn đến các trạng thái trung gian của tính toán.

Cái chính để kiểm thử thành công là chọn dữ liệu kiểm thử đúng. Mục đích là tìm tập nhỏ kiểm thử mà cho phép chúng ta đánh giá thông tin nhận được với giả thiết là kiểm thử tổng thể. Chẳng hạn, đầu vào nhận một số nguyên như đối số, ta có thể kiểm thử trên số số nguyên chẵn bất kỳ, trên số số nguyên lẻ bất kỳ và số 0.

Kiểm thử hộp đen

Các trường hợp kiểm thử test cases được sinh ra bằng việc xem xét cả đặc tả và cài đặt. Trong kiểm thử hộp đen, chúng ta sinh ra dữ liệu kiểm thử chỉ từ đặc tả, không xem xét đến cấu trúc bên trong của module được kiểm thử. Cách tiếp cận này, mà là chung thông qua nhiều nguyên lý công nghệ, có một số ưu điểm rất quan trọng. Ưu điểm quan trọng nhất là thủ tục kiểm thử này không bị tác động bất lợi từ thành phần mà nó kiểm thử. Chẳng hạn, giả sử tác giả của chương trình đưa ra một giả thiết ẩn không đúng là chương trình không bao giờ được gọi với một lớp nào đó của đầu vào. Lập trình với giả thiết này, tác giả có thể mắc sai lầm không viết đoạn code xử lý với lớp đầu vào đó. Nếu dữ liệu kiểm thử được sinh từ việc xem xét chương trình này, người ta có thể dễ dàng bỏ qua dữ liệu đầu vào dựa trên giả thiết sai lầm đó. Ưu điểm thứ hai của kiểm thử hộp đen là bền vững với sự thay đổi cài đặt. Dữ kiện kiểm thử hộp đen sẽ không thay đổi ngay cả khi có những thay đổi đáng kể trong chương trình được kiểm thử. Ưu điểm cuối cùng là kết quả kiểm thử có thể được thực hiện bởi những người không biết cấu tạo bên trong của chương trình được kiểm thử.

- **Kiểm thử các nhánh đọc theo đặc tả:**

Một cách sinh ra dữ liệu kiểm thử hộp đen là khám phá các nhánh có thể có đọc theo đặc tả. Các nhánh này có thể theo cả các mệnh đề yêu cầu REQUIRES và cả các mệnh đề tác động EFFECTS. Một ví dụ dựa theo mệnh đề yêu cầu, khi xem xét đặc tả sau:

```
static float sqrt (float x, float epsilon)
    // REQUIRES: x >= 0 && .00001 < epsilon < .001
    // EFFECTS: Returns sq such that x - epsilon <= sq*sq <= x + epsilon.
```

Mệnh đề yêu cầu của đặc tả này là hội của hai biến logic

1. $x \geq 0$ và
2. $0.00001 < \text{epsilon} < 0.001$

Để khám phá các cách khác nhau để mệnh đề yêu cầu được thỏa mãn, chúng ta cần xét tổ hợp của hai điều kiện thành phần:

1. $x = 0$ và $0.00001 < \text{epsilon} < 0.001$
2. $x > 0$ và $0.00001 < \text{epsilon} < 0.001$

Tập dữ liệu kiểm thử bất kỳ cho `sqrt` cần kiểm thử mỗi một trong các trường hợp trên. Chúng ta luôn cần phải kiểm tra mệnh đề tác động một cách cẩn thận và cố gắng tìm dữ liệu kiểm thử mà áp dụng các cách khác nhau để thỏa mãn nó. Chẳng hạn xét thử tục sau:

```
static boolean isPrime (int x)
// EFFECTS: If x is a prime returns true else returns false.
```

Mệnh đề tác động của đặc tả này là tuyển của x là nguyên tố hoặc không là nguyên tố. Cả hai trường hợp cần được kiểm thử bởi các test case.

Thông thường các nhánh đọc theo mệnh đề tác động để cập đến xử lý lỗi. Như vậy, dữ liệu kiểm thử cần bao quát mọi trường hợp sinh lỗi. Chẳng hạn, xét đặc tả sau:

```
static int search (int[ ] a, int x)
throws NotFoundException, NullPointerException
// EFFECTS: If a is null throws NullPointerException else if x is in a,
//           returns i such that a[i] = x, else throws NotFoundException.
```

Chúng ta cần kiểm thử cả trường hợp ở đó x có trong a và trường hợp x không có trong a , cũng như cả trường hợp khi a là null. Tương tự, nếu căn bậc hai `sqrt` có xử lý lỗi khi $x < 0$, thì chúng ta cũng cần kiểm thử cả trường hợp sẽ ra thông báo lỗi đó.

- **Kiểm thử các điều kiện biến:**

Một chương trình luôn cần được kiểm thử trên các giá trị đầu vào “tiêu biểu” – chẳng hạn một mảng hoặc một tập hợp chứa một số phần tử hoặc một số nguyên nằm giữa giá trị bé nhất và lớn nhất theo yêu cầu của chương trình. Cũng quan trọng phải kiểm thử trường hợp đầu vào không tiêu biểu, mà được gọi là các điều kiện biến.

Xem xét mọi nhánh đọc theo mệnh đề yêu cầu sẽ kiểm thử một số kiểu điều kiện biến, chẳng hạn, ở đó `sqrt` được yêu cầu tìm căn bậc hai của 0. Nhiều điều kiện biến, tuy

nhiên, không hiện ra từ phân tích như vậy. Quan trọng là kiểm tra càng nhiều điều kiện biên càng tốt. Các kiểm tra như vậy sẽ nắm bắt được hai loại lỗi:

1. Các lỗi logic, ở đó một nhánh tương ứng với một trường hợp đặc biệt được thể hiện bởi một điều kiện biên bị bỏ sót.
2. Lỗi kiểm tra các điều kiện mà có thể buộc ngôn ngữ bên dưới hoặc hệ thống phần cứng sinh ra lỗi.

Để sinh ra kiểm thử được thiết kế bắt các lỗi đó, một ý tưởng tốt là sử dụng dữ liệu kiểm thử bao quát được mọi tổ hợp các giá trị lớn nhất và nhỏ nhất cho phép đối với các đối số bị chặn. Chẳng hạn, kiểm thử sqrt cần bao gồm các trường hợp epsilon rất gần 0.001 và 0.00001. Đối với các xâu, kiểm thử cần bao gồm xâu rỗng và xâu một ký; đối với mảng, chúng ta cần kiểm tra mảng rỗng và mảng một phần tử.

- Các lỗi bí danh

Một dạng khác của điều kiện biên xảy ra, khi hai tham số hình thức khác nhau đều tham chiếu đến cùng một đối tượng thay đổi được. Chẳng hạn thủ tục:

```
static void appendVector (Vector v1, Vector v2)
    throws NullPointerException
    // MODIFIES: v1 and v2
    // EFFECTS: If v1 or v2 is null throws NullPointerException else
    // removes all elements of v2 and appends them in reverse order
    // to the end of v1.
```

được cài đặt như sau:

```
static void appendVector (Vector v1, Vector v2) {
    if (v1 == null) throws new NullPointerException
        ("Vectors.appendVector");
    while (v2.size( ) > 0) {
        v1.addElement(v2.lastElement( ));
        v2.removeElementAt(v2.size( ) - 1); }
}
```

Bất cứ dữ liệu kiểm thử nào mà không bao gồm một đầu vào ở đó v1 và v2 tham chiếu đến cùng một mảng khác rỗng đều có thể bị quên dẫn đến lỗi rất nghiêm trọng trong appendVector.

Tóm lại:

- Các kiểm thử hộp đen dựa trên đặc tả của chương trình, chứ không dựa trên cài đặt của nó.
- Các kiểm thử hộp đen cần kiểm thử tất cả nhánh đọc theo đặc tả, nhiều nhất có thể. Ngoài ra chúng cần kiểm thử mọi điều kiện biên và kiểm tra các lỗi bí danh.

Kiểm thử hộp trắng

Trong khi kiểm thử hộp đen nói chung là cách tốt nhất để bắt đầu kiểm thử chương trình thông suốt, nhưng nó không là đủ. Không xem xét cấu trúc bên trong của chương trình, không thể biết được những test case nào sẽ cho thêm thông tin mới. Do đó không thể nói thu được bao nhiêu thông tin từ tập dữ liệu kiểm thử hộp đen. Chẳng hạn, giả sử một chương trình dựa trên tìm kiếm trong bảng đầu vào nào đó và tính toán cho một số cái khác. Nếu dữ liệu kiểm thử hộp đen xảy ra chỉ bao gồm các giá trị tìm kiếm trên bảng sử dụng, thì kiểm thử không cho thông tin nào về phần chương trình tính toán cho các giá trị tính toán đầu khác.

Do đó, cũng cần thực hiện kiểm thử hộp trắng ở đó mã của chương trình được xem xét kiểm thử. Kiểm thử hộp trắng cần hỗ trợ kiểm thử hộp đen với các đầu vào mà áp dụng cho các nhánh khác đọc theo chương trình. Mục tiêu là có tập kiểm thử mà mỗi nhánh được áp dụng bởi ít nhất một phần tử của tập. Chúng ta nói rằng tập kiểm thử là nhánh đầy đủ.

Xét chương trình:

```
static int maxOfThree (int x, int y, int z) {  
    if (x > y)  
        if (x > z) return x; else return z;  
    if (y > z) return y; else return z; }
```

Mặc dù có n^3 đầu vào, ở đó n là phạm vi số nguyên mà chương trình cho phép. Ở đây chỉ có 4 nhánh đọc chương trình. Do đó tính chất nhánh đầy đủ cho phép chúng ta tách dữ liệu kiểm thử thành 4 tập. Trong một lớp, x lớn hơn y và z . Trong tập khác, x lớn hơn y , nhưng nhỏ hơn z , và tiếp tục như vậy. Đại diện của 4 lớp là:

3, 2, 1; 3, 2, 4; 1, 2, 1; 1, 2, 3

Dễ dàng chỉ ra rằng kiểm thử nhánh đầy đủ là không đủ để bắt hết lỗi. Xét chương trình:

```
static int maxOfThree (int x, int y, int z) {  
    return x; }
```

Tập kiểm thử chỉ chứa đầu vào: 2, 1, 1 là nhánh đầy đủ cho chương trình này. Sử dụng tập kiểm thử này có thể dẫn chúng ta đến sai lầm khi tin tưởng chương trình này là đúng, do đó kiểm thử này không phát hiện được lỗi. Tuy nhiên trong trường hợp chương trình có quá nhiều nhánh, thì chúng ta phải tìm cách khác để phân nhánh tập kiểm thử.

Tóm lại:

- Kiểm thử hộp trắng xem xét đến văn bản của chương trình
- Kiểm thử hộp trắng cần hỗ trợ kiểm thử hộp đen sao cho kiểm thử là nhánh đầy đủ, mỗi nhánh trong code được áp dụng bởi ít nhất một kiểm thử
- Đối với vòng lặp với số bước cố định, kiểm thử cần bao gồm một và hai vòng. Với vòng lặp có biến số vòng lặp, kiểm thử cần bao gồm không, một và hai vòng lặp.
- Với đệ qui, kiểm thử cần bao gồm trường hợp cơ sở và một lời gọi đệ qui.

2.3.3 Kiểm thử thủ tục

Để minh họa kiểm thử thủ tục, chúng ta xét ví dụ thủ tục kiểm tra một xâu có phải là từ đọc xuôi ngược đều như nhau không?

```
static boolean palindrome (String s) throws NullPointerException {  
    // EFFECTS: If s is null throws NullPointerException, else returns  
    // true if s reads the same forward and backward; else returns false.  
    // E.g., "deed" and " " are both palindromes.  
    int low = 0;  
    int high = s.length( ) - 1;  
    while (high > low) {  
        if (s.charAt(low) != s.charAt(high)) return false;  
        low ++;  
        high --; }  
    return true; }
```

Hình trên cho đặc tả và cài đặt của thủ tục đó. Bằng việc xem xét đặc tả, chúng ta thấy cần thiết kiểm thử đối số null, kiểm thử mệnh đề trả về là true và false. Ngoài ra chúng ta cần kiểm tra xâu rỗng và xâu gồm một ký tự. Kiểm tra code chỉ ra chúng ta cần test các trường hợp sau:

1. NullPointerException xảy ra khi gọi length
2. Không thực hiện vòng lặp
3. Trả về false trong vòng đầu tiên
4. Trả về true sau vòng đầu tiên
5. Trả về false trong vòng thứ hai
6. Trả về true sau vòng thứ hai

Như vậy có thể tập kiểm tra gồm các xâu: null, “ “, “d”, “deed”, “eedc”, “aabaa” và bổ sung một số trường hợp xâu kiểm tra có độ dài lẻ như “abcba”.

2.3.4 Kiểm thử trừu tượng dữ liệu

Trong kiểm thử kiểu dữ liệu, chúng ta sinh các test case như bình thường bằng việc xem xét đặc tả và cài đặt của từng thao tác. Chúng ta cũng cần kiểm thử các thao tác như một

nhóm, vì một số thao tác (như hàm tạo hoặc thay đổi) tạo ra các đối tượng được sử dụng trong các thao tác khác. Trong các thao tác của Inset, hàm tạo và các phương thức insert, remove cần được sử dụng để sinh đối số cho các phương thức khác hoặc giữa chúng với nhau. Các phương thức observers được sử dụng để kiểm thử hàm tạo và thay đổi mutator; như isIn và size được dùng để kiểm thử insert và remove.

Đặc tả một phần trùu tượng InSet

```

public class IntSet {
    // OVERVIEW: IntSets are mutable, unbounded sets of integers.
    //   A typical IntSet is {x1, ..., xn}.

    // constructors
    public IntSet ( )
        // EFFECTS: Initializes this to be empty.

    // methods
    public void insert (int x)
        // MODIFIES: this
        // EFFECTS: Adds x to the elements of this, i.e., this_post = this + { x }.

    public void remove (int x)
        // MODIFIES: this
        // EFFECTS: Removes x from this, i.e., this_post = this - x .

    public boolean isIn (int x)
        // EFFECTS: If x is in this returns true else returns false.

    public int size ( )
        // EFFECTS: Returns the cardinality of this.

    public Iterator elements ( )
        // EFFECTS: Returns a generator that produces all the elements of
        //   this (as Integers), each exactly once, in arbitrary order.
        // REQUIRES: this must not be modified while the generator is in use.
}

// for IntSet:
public boolean repOk( ) {
    if (els == null) return false;
    for (int i = 0; i < els.size( ); i++) {
        Object x = els.get(i);
        if (!(x instanceof Integer)) return false;
        for (int j = i + 1; j < els.size( ); j++)
            if (x.equals(els.get(j))) return false;
    }
    return true; }
```

RepOk đóng vai trò quan trọng trong kiểm thử: chúng ta sẽ gọi nó sau khi gọi các thao tác của kiểu dữ liệu đó.

Chúng ta xem xét các nhánh trong đặc tả. Đặc tả của isIn và elements có các nhánh rõ ràng để khám phá. Đối với isIn chúng ta cần sinh ra các trường hợp để trả về kết quả true và false. Vì elements là iterator chúng ta cần xem xét các trường hợp InSet có độ dài 0, 1 và 2. Tập InSet rỗng và một phần tử cũng là các điều kiện biên kiểm thử. Do đó để kiểm thử observers, chúng ta cần bắt đầu với các InSets sau:

- InSet rỗng tạo bởi lời gọi hàm tạo InSet
- InSet một phần tử được tạo bởi insert 3 vào tập rỗng
- InSet hai phần tử được tạo bởi insert 3 và 4 vào tập rỗng

Với mỗi một InSet trên chúng ta sẽ gọi isIn, size và elements và kiểm tra kết quả. Trong trường hợp isIn chúng ta gọi với các phần tử ở trong tập hợp và số khác không ở trong tập hợp.

Chúng ta rõ ràng chưa có đủ trường hợp. Chẳng hạn, remove chưa được kiểm thử, các nhánh trong các đặc tả khác cũng chưa được bàn tới. Các nhánh này bị che giấu ở đâu đó trong đặc tả. Chẳng hạn, kích thước của một InSet sẽ không đổi, nếu ta insert một phần tử đã có trong tập hợp. Và tương tự, kích thước sẽ giảm khi ta xóa phần tử chỉ trong trường hợp nó có trong tập đó, do đó chúng ta cần xem xét trường hợp xóa phần tử sau khi đã chèn nó vào và trường hợp xóa phần tử không có trong tập hợp. Chúng ta sẽ sử dụng các InSets bổ sung sau:

- Tập hợp nhận được sau khi insert 3 hai lần vào tập rỗng.
- Tập hợp nhận được bởi insert 3 sau đó remove 3.
- Tập hợp nhận được bởi insert 3 và remove 4.

Để tìm các nhánh bị che giấu, chúng ta cần xem xét tường minh các nhánh trong các phương thức thay đổi mutator. Như, insert cần phải làm việc đúng khi phần tử được chèn đã thuộc tập, hoặc tương tự với remove, mà đã tạo ra 3 trường hợp trên.

Cài đặt một phần InSet

```

public IntSet ( ) { els = new Vector( ); }

public void insert (int x) {
    Integer y = new Integer(x);
    if (getIndex(y) < 0) els.add(y); }

public void remove (int x) {
    int i = getIndex(new Integer(x));
    if (i < 0) return;
    els.set(i, els.lastElement( ));
    els.remove(els.size( ) -1); }

public boolean isIn (int x) {
    return getIndex(new Integer(x)) >= 0; }

private int getIndex (Integer x) {
    // EFFECTS: If x is in this returns index where x appears else returns -1.
    for (int i = 0; i < els.size( ); i++)
        if (x.equals(els.get(i))) return i;
    return -1; }

public int size ( ) { return els.size( ); }
}

```

Ngoài ra, chúng ta cần tìm các nhánh trong cài đặt của các phương thức. Các trường hợp đã xác định sẽ được giải quyết nếu cài đặt sử dụng vector không có lặp làm cấu trúc bên trong cho InSet. Một vấn đề trong isIn mà chưa vòng lặp thông quan gọi getIndex. Để bao quát mọi nhánh trong vòng lặp này, chúng ta cần kiểm tra vector hai phần tử với không sánh và sánh phần tử đầu hoặc phần tử thứ hai. Tương tự, trong remove, chúng ta cần tin tưởng là xóa phần tử thứ nhất và phần tử thứ hai trong vector.

Kiểm thử phân cấp kiểu

- Các kiểu con cần được kiểm thử sử dụng kiểm thử hộp đen của kiểu supertypes của chúng và kiểm thử hộp đen của chính nó. Các kiểm thử supertype cần sử dụng hàm tạo của kiểu con.
- Kiểm thử hộp đen kiểu con bổ sung bao gồm cả các phương thức bổ sung và các phương thức kế thừa thay đổi hành vi.
- Kiểm thử hộp trắng cho lớp cha cần không được sử dụng khi kiểm thử các lớp kế thừa.
- Kiểm thử lớp trừu tượng yêu cầu có lớp con cụ thể. Cặp này được kiểm thử sử dụng hộp đen cho cả sub và supertype và cũng như kiểm thử hộp trắng cho cả lớp cha và lớp kế thừa.

- Kiểm thử phân cấp kiểu mà cung cấp nhiều cài đặt cho supertype có thể yêu cầu kết hợp kiểm thử kiểu con và bổ sung kiểm thử hộp đen để chứng tỏ rằng kiểu con thực sự được chọn cho nhiều đối tượng khác nhau.

2.3.5 Các công cụ kiểm thử

Sẽ có ích tự động quá trình kiểm thử càng nhiều càng tốt. Chúng ta thông thường không thể tự động sinh dữ liệu kiểm thử; sinh các đầu vào phù hợp để kiểm thử một chương trình là quá trình không thuật toán hóa được đòi hỏi những suy nghĩ nghiêm túc. Hơn nữa, tự động quá trình quyết định đầu ra nào là phù hợp cho một tập đầu vào tùy ý thông thường là khó và cũng dễ sinh lỗi như viết chương trình đang kiểm thử.

Cái mà chúng ta có thể tự động là các quá trình triệu gọi một chương trình với dãy đầu vào xác định trước và kiểm tra kết quả với dãy kiểm thử xác định trước để chấp nhận đầu ra hay không. Cơ chế mà làm điều này được gọi là driver. Một driver cần gọi đơn vị cần kiểm thử và theo dõi xem nó thực hiện như thế nào. Cụ thể hơn, nó cần

1. Thiết lập môi trường cần thiết để gọi đơn vị cần kiểm thử. Trong một ngôn ngữ nào đó, điều này có thể tạo và khởi tạo một số biến tổng thể nào đó. Trong đa số các ngôn ngữ, nó có thể thiết lập và mở một số files.
2. Thực hiện một loạt các lời gọi. Các đối số của các lời gọi này có thể được đọc từ file hoặc được nhúng vào code của driver. Nếu các đối số được đọc từ file, chúng cần phải được kiểm tra xem có phù hợp không, nếu có thể.
3. Kiểm tra kết quả và tính phù hợp của chúng.

Driver cho sqrt

```
// accept as inputs the files:  
//   file_of_tests, bad_tests, correct_results, and incorrect_results  
  
for { // each test in file_of_tests  
    if (test.square < 0 || test.epsilon < .00001 || test.epsilon > .001) {  
        // add test to bad_tests  
    }  
    else {  
        result = Num.sqrt(test.square, test.epsilon);  
        if (Num.fabsf(square - result*result) <= epsilon ) {  
            // add <test, result> to correct_results  
        }  
        else {  
            // add <test, result> to incorrect_results  
        }  
    }  
}
```

Cách thông dụng nhất kiểm tra tính phù hợp của kết quả là so sánh chúng với dãy kết quả mong muốn mà đã được lưu trong file. Tuy nhiên, đôi khi tốt hơn là viết chương trình so sánh kết quả trực tiếp với đầu vào. Chẳng hạn, nếu chương trình được đề xuất tìm nghiệm của đa thức, dễ dàng viết driver mà kiểm tra giá trị trả về có phải là nghiệm không, như driver kiểm thử cài đặt đặc tả `sqrt` được nêu trong Hình trên.

Ngoài drivers, kiểm thử thông thường sử dụng các stubs. Một driver mô phỏng các phần của chương trình, mà gọi đơn vị được kiểm tra. Stubs mô phỏng các phần của chương trình mà được gọi bởi đơn vị đang được kiểm tra. Một stub cần

1. Kiểm tra tính hợp lý của môi trường được cung cấp bởi người gọi.
2. Kiểm tra tính hợp lý của các đối số được truyền bởi người gọi.
3. Điều chỉnh các đối số và môi trường và các giá trị trả về sao cho người gọi có thể tiếp tục xử lý. Cách tốt nhất là các tác động này sánh đặc tả của đơn vị mà stub đang mô phỏng. Không may, điều này không phải bao giờ cũng có thể. Đôi khi giá trị “đúng” cần được tìm chỉ bằng cách viết chương trình mà stub được đề xuất thay thế. Trong các trường hợp như thế, chúng ta cần chấp nhận một giá trị “dùng được”.

Các drivers rõ ràng là cần thiết khi kiểm thử các module trước khi các module mà triệu gọi nó được viết. Stubs là cần thiết, khi kiểm thử các modules trước khi các modules mà chúng triệu gọi được viết. Cả hai là cần thiết để kiểm thử đơn vị, ở đó chúng ta muốn cô lập đơn vị mà được kiểm thử càng nhiều càng tốt khỏi các phần khác của chương trình.

Trên thực tế, nói chung là cài đặt drivers và stubs mà dựa trên tương tác với con người. Một cài đặt rất đơn giản của stub có thể chỉ là in ra các đối số mà nó được gọi với và hỏi người đang kiểm thử cung cấp giá trị cần được trả về. Tương tự, một cài đặt đơn giản của driver có thể dựa trên người đang kiểm thử kiểm chứng tính đúng đắn của kết quả được trả về bởi đơn vị đang kiểm thử. Mặc dù các drivers và stubs đơn giản là dễ cài đặt, chúng cần được tránh khi có thể. Chúng rất dễ sinh lỗi so với các drivers và stubs tự động, và chúng khó được xây dựng dựa trên cơ sở dữ liệu kiểm thử tốt và tái sử dụng kiểm thử.

Tái sử dụng kiểm thử là rất quan trọng. Kịch bản sau là rất thông dụng:

1. Chương trình được kiểm thử trên các đầu vào từ 1 đến n không phát hiện có lỗi.
2. Kiểm thử chương trình trên đầu vào $n + 1$ thấy có lỗi
3. Bắt lỗi và sửa chương trình làm việc đúng với $n + 1$
4. Kiểm thử tiếp tục với $n + 2$

Dù bất cứ thay đổi lớn nhỏ nào, quan trọng là tin tưởng chương trình vẫn vượt qua kiểm thử với các đầu vào mà nó đã vượt qua. Điều đó gọi là kiểm thử hồi qui. Kiểm thử hồi qui là thực tế chỉ khi các công cụ là sẵn sàng làm cho nó dễ dàng chạy lại các kiểm thử cũ.

Kiểm thử đơn vị, tích hợp, hồi qui:

- Kiểm thử đơn vị kiểm thử một module cô lập với những cái khác. Nó đòi hỏi:
 - Một driver mà tự động kiểm thử module đó
 - Stubs mà mô phỏng hành vi của mọi modules khác được sử dụng trong module đó
- Kiểm thử tích hợp kiểm thử đồng thời một nhóm các modules
- Kiểm thử hồi qui là chạy lại mọi kiểm thử sau khi lỗi đã được sửa.

Kiểm thử, bắt lỗi và lập trình bảo vệ:

- Kiểm thử là cách công nhận tính đúng đắn của chương trình
- Bắt lỗi là quá trình tìm và loại trừ lỗi
- Lập trình bảo vệ bao gồm chèn kiểm tra để phát hiện lỗi trong chương trình. Nó làm cho bắt lỗi dễ dàng hơn

2.4 Đặc tả yêu cầu cho Search Engine

Phần này khám phá máy tìm kiếm, mà cho phép người dùng truy vấn một họ các tài liệu. Nó mô tả cả phân tích yêu cầu ngắn gọn và đặc tả yêu cầu nhận được.

Như thông thường, chúng ta bắt đầu phân tích bằng một kịch bản thể hiện hành vi cho một trường hợp thông thường. Giả sử một người sử dụng bắt đầu phiên với máy tìm kiếm. Câu hỏi đầu tiên xuất hiện liên quan đến việc máy đã có một họ các tài liệu mà được nhớ từ lần chạy cuối không. Giả sử khách hàng chỉ quan tâm đến việc tìm kiếm mới. Do đó, việc đầu tiên mà người sử dụng cần tìm là định danh các tài liệu cần tìm. Giả sử điều này có thể được thực hiện bởi việc thể hiện địa chỉ URL của một trang chứa các tài liệu; máy tìm kiếm sẽ chạy tìm kiếm mọi tài liệu này. Tiếp theo, khách hàng quan tâm đến tìm kiếm trên nhiều trang, do đó, người sử dụng có thể bổ sung địa chỉ URL của các trang chứa tài liệu, và máy tìm kiếm sẽ tăng họ tài liệu của nó lên. Khách hàng chỉ ra rằng họ tài liệu này có thể tăng lên bất cứ lúc nào, không chỉ đầu phiên, không có lý do gì mà loại bỏ tài liệu từ họ đó.

Khách hàng chỉ ra là người sử dụng có thể tìm kiếm trong họ một tài liệu với một tên cụ thể. Tuy nhiên, mục đích chính của máy tìm kiếm là chạy truy vấn trên một họ, mà có nghĩa là chúng ta phải quyết định truy vấn là gì. Khi trao đổi với khách hàng, chúng ta quyết định rằng, truy vấn bắt đầu bởi người sử dụng với một từ đơn, mà chúng ta gọi là *từ khóa, keyword*. Khách hàng chỉ ra rằng, nhiều từ là không quan tâm (chẳng hạn, như “and” và

“the”) và không được sử dụng như từ khóa. Khách hàng mong muốn máy tìm kiếm biết các từ nào là không quan tâm mà không cần sự can thiệp của bất cứ người sử dụng nào; như vậy, nó cần phải truy cập đến một bộ nhớ, chẳng hạn như file, mà liệt kê các từ không quan tâm.

Hệ thống phản hồi truy vấn bằng việc thể hiện thông tin về những tài liệu chứa từ khóa. Thông tin này được sắp thứ tự bởi số lần từ khóa xuất hiện trong các tài liệu. Hệ thống không thể hiện tài liệu thực tế, mà cung cấp thông tin sao cho người sử dụng có thể kiểm tra các tài liệu sánh nếu như mong muốn.

Tuy nhiên, khả năng truy vấn sử dụng một từ khóa duy nhất là rất hạn chế, và khách hàng cũng yêu cầu khả năng “làm mịn” truy vấn bằng việc cung cấp thêm từ khóa khác. Các tài liệu sánh cần chứa mọi từ khóa. Khách hàng đưa ra các truy vấn có tính toán hơn, như các truy vấn các tài liệu sánh chứa bất cứ một trong các từ khóa hoặc các truy vấn mà yêu cầu các từ khóa đứng cạnh nhau trong văn bản theo thứ tự mà nó cần sánh. Tuy nhiên, các truy vấn như thế là để tách ra một sản phẩm mong muốn riêng.

Bây giờ chúng ta cần xem xét người sử dụng và lỗi hệ thống, và cũng như hiệu năng. Vấn đề hiệu năng chính là làm sao tiến hành các truy vấn đó, khách hàng muốn nó được thực hiện một cách mau lẹ. Yêu cầu này có hai hệ quả. Thứ nhất, chương trình cần chứa cấu trúc dữ liệu làm tăng tốc độ xử lý truy vấn. Thứ hai (quan trọng hơn) là truy vấn có yêu cầu viếng thăm các trang web chứa tài liệu không. Khách hàng chỉ ra rằng điều đó sẽ không xảy ra; thay vào đó truy vấn cần phải dựa trên thông tin đã được biết đối với máy tìm kiếm. Một hệ quả của quyết định này là họ tài liệu không cần phải là mới nhất. Một trang có thể đã được sửa từ lâu, máy tìm kiếm đã được thông tin về điều đó, và truy vấn sẽ không phản ánh các sửa đổi đó: chúng sẽ bỏ qua các tài liệu bổ sung mới hoặc tìm các tài liệu mà không còn tồn tại từ lâu. Khách hàng chỉ ra rằng điều đó là chấp nhận được, nhưng theo dõi được các thay đổi có thể sẽ được quan tâm trong phiên bản tương lai. Khách hàng cũng chỉ ra rằng mọi thông tin về các tài liệu cần được lưu ở máy tìm kiếm, sao cho nếu một truy vấn sánh một tài liệu, người sử dụng có thể xem tài liệu đó ngay cả nó không còn tồn tại nữa tại trang từ đó nó được lấy lên. Một điểm lưu ý về các quyết định này là sự cân bằng được lựa chọn giữa tốc độ xử lý truy vấn và không gian lưu giữ các tài liệu tại máy tìm kiếm.

Bây giờ xem xét các lỗi. Hệ thống có bộ nhớ thường xuyên chứa thông tin về các từ không quan tâm, nhưng bộ nhớ này không thay đổi và khách hàng không liên quan gì đến các lỗi truyền thông. Tiếp theo, khách hàng chỉ ra rằng có thể chấp nhận để máy tìm kiếm đơn giản dừng vì cái gì đó xảy ra không đúng.

Tuy nhiên có một số lỗi của người sử dụng đáng quan tâm. Người sử dụng có thể nhập một từ không quan tâm làm từ khóa hoặc có thể nhập một từ mà không có trong bất cứ tài liệu nào; khách hàng chỉ ra rằng người sử dụng có thể được nhắc nhở trong trường hợp thứ nhất, nhưng trong trường hợp thứ hai, phản hồi đơn giản là một tập rỗng các tài liệu sánh.

Người sử dụng cũng có thể nhập một địa chỉ URL của một trang mà không tồn tại, không chứa các tài liệu hoặc đã được bổ sung vào họ các tài liệu; mọi hành động này cần được phản ánh là nhắc nhở người sử dụng về lỗi. Khách hàng chỉ ra rằng có thể chấp nhận nếu tài liệu được tìm thấy có ở nhiều trang và rằng trong trường hợp này, tài liệu được chỉ ra chỉ trong một trang. Hai tài liệu được xem như nhau nếu chúng có cùng tiêu đề; cũng như vậy, các phiên bản sau sẽ xét trường hợp hai văn bản khác nhau có cùng tên.

Bây giờ chúng ta có ý tưởng thô về việc máy tìm kiếm được hỗ trợ làm gì, chúng ta đã sẵn sàng viết đặc tả yêu cầu. Như chúng ta thường làm, chúng ta sẽ không bao trùm một số vấn đề mà đã xem xét khi phân tích, nhưng sẽ giải quyết khi đi đến đặc tả chính xác. Vì vậy, quá trình viết đặc tả yêu cầu bao gồm định nghĩa mô hình dữ liệu, sẽ là phần thực chất và quan trọng của quá trình phân tích yêu cầu.

Các tập hợp và các quan hệ cho máy tìm kiếm được định nghĩa trên Hình 2.6 và đồ thị được cho trên Hình 2.7. Một tài liệu có một tiêu đề, một số URL (của các trang từ đó nó được lấy lên) và thân văn bản, một thân là một dãy các từ. Đỉnh *NK* thể hiện các từ không quan tâm, tập này là cố định (các thành viên của nó không thay đổi). Tập sánh *Match* thể hiện tập các văn bản mà sánh với truy vấn hiện tại; *Key* là tập các từ khóa được sử dụng trong truy vấn đó. *Key* và *NK* là các tập rời nhau (một từ khóa không bao giờ là từ không quan tâm), nhưng chúng không cùng phủ tập từ *Word* (vì tại một thời điểm bất kỳ, nhiều từ trong các văn bản không là từ khóa mà cũng không là từ không quan tâm). *Cur* là văn bản mà được định danh bởi tên và là tài liệu đang quan tâm; *CurMatch* là một bản sánh *Match* mà hiện tại đang được xem xét.

Các ràng buộc cho máy tìm kiếm được cho trên Hình 2.8. Chúng ta có thể thấy *sum* là quan hệ suy diễn; đây là tổng của các số lần xuất hiện của mỗi từ khóa trong văn bản. Các ràng buộc này chỉ ra rằng các chỉ số trong *Match* và *Entry* là duy nhất, mà các văn bản này có trong *Match* chỉ nếu khi truy vấn xảy ra các văn bản trong *Match* chính xác là cái mà chưa mọi từ khóa trong *Key*, và thứ tự các văn bản trong *Match* phản ánh số lần xuất hiện các từ khóa trong văn bản.

Domains:

Doc: tập các văn bản (tài liệu)

URL: các địa chỉ của các trang mà ở đó các văn bản được tìm thấy

Title: tiêu đề của một văn bản

Entry: các bản nhập (các cặp từ/chỉ số) trong một văn bản

Num: các số nguyên dương

Word: các từ trong các văn bản

NK: các từ không quan tâm (noninteresting keys)

Key: các từ khóa sử dụng trong truy vấn hiện thời

Match: các tài liệu sánh các từ khóa trong truy vấn hiện thời

Cur: văn bản đang xem xét

CurMatch: bản sánh đang xem xét

Các quan hệ

site: địa chỉ URL của các trang chứa văn bản

title: tiêu đề của một văn bản

body: các bản nhập mà tạo nên nội dung của một văn bản

index: chỉ số của một bản nhập trong Entry

wd: từ trong bản nhập trong Entry

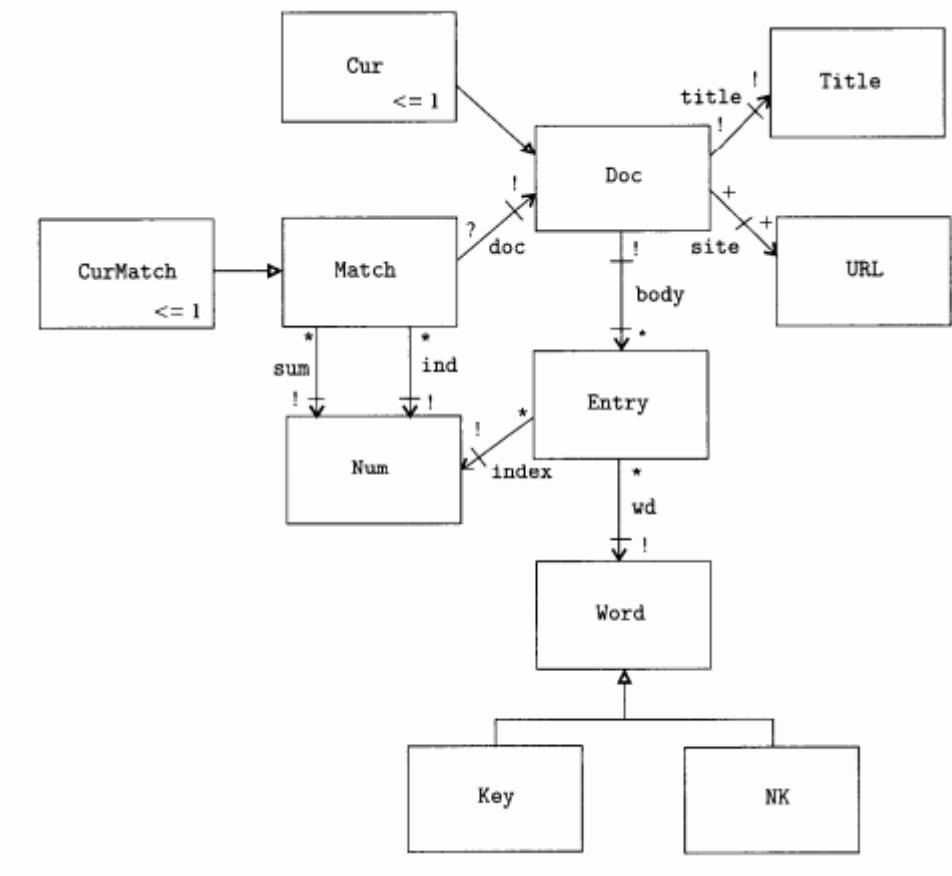
doc: văn bản của bản nhập trong Match

ind: chỉ số của một bản nhập trong Match

sum: đếm số lần xuất hiện của các từ khóa trong một bản sánh

Hình 2.6 Các tập và các quan hệ cho máy tìm kiếm

Đặc tả yêu cầu cho máy tìm kiếm được cho trên Hình 2.9. Đặc tả này không cho thông tin về định dạng, chúng ta giả thiết định dạng chuẩn cho URL và định dạng đơn giản cho các văn bản. Chẳng hạn, các từ trong các văn bản là những gì xuất hiện giữa các khoảng trắng hoặc các ký tự điều khiển HTML, và các phần trong một văn bản (tiêu đề, tác giả, thân bài) được phân tách theo cách thông thường. Đặc tả này chỉ ra rằng máy tìm kiếm biết về các từ không quan tâm qua một file riêng và nó không có trạng thái thường xuyên khác.



Hình 2.7 Đồ thị cho máy tìm kiếm

Có một số mở rộng tiềm năng cho máy tìm kiếm. Chúng ta sẽ liệt kê chúng dưới đây:

- Chúng ta muốn hỗ trợ định nghĩa sâu sắc hơn về từ và văn bản.
- Chúng ta muốn cho phép một số văn bản khác nhau có cùng tên.
- Chúng ta muốn hỗ trợ truy vấn sâu sắc hơn – chẳng hạn như hợp các bản sánh chứa một trong một số từ khóa.
- Chúng ta muốn cung cấp thông tin nhiều hơn về các văn bản chuyên biệt chẳng hạn như đếm số từ khác nhau. Hoặc chúng ta có thể chỉ ra mọi chỗ ở đó một từ cụ thể nào đó xuất hiện trong một văn bản.
- Chúng ta muốn cung cấp cách cho người sử dụng nhìn thấy danh sách mọi từ quan tâm xuất hiện trong các văn bản của họ tài liệu đó.
- Chúng ta muốn cho phép các từ không quan tâm được thay đổi.
- Chúng ta muốn máy tìm kiếm định kỳ viếng thăm lại các trang ở đó các văn bản của nó được lấy lên để nhận được bất cứ văn bản mới nào được bổ sung vào các trang đó.

- Chúng ta muốn máy tìm kiếm lưu lại địa chỉ URL của các trang một cách chắc chắn sao cho nó có thể làm mới các văn bản khi nó khởi động lại hoặc ngay cả lưu giữ các văn bản một cách ổn định.
- Chúng ta có thể muốn máy tìm kiếm không lưu các văn bản, nhưng giữ thông tin về chúng và sau đó làm mới văn bản khi người sử dụng muốn xem xét.

Các quan hệ suy diễn và các hàm bổ trợ

Tổng sum cho mỗi bản sánh là tổng số lần đếm xuất hiện của mọi từ khóa trong văn bản đó

Với mọi m : Match [$m.sum = \sum_{k \in Key} sumkey(m.doc, k)$]

$sumkey(d, k) = | \{e: Entry / e \text{ trong } d.body \&& e.wd = k\} |$

Văn bản d sánh nếu nó chứa mọi từ khóa trong Key

$matches(d) = Key ! = \{ \} \&& (\text{với mọi } k: Key (sumkey(d, k) > 0))$

Các ràng buộc

Mỗi bản nhập trong một văn bản có chỉ số khác nhau

Với mọi $d: Doc, e1, e2: Entry [e1, e2 \text{ trong } d.body \&&$

$e1.index = e2.index \rightarrow e1 = e2$]

Mỗi bản sánh có chỉ số khác nhau

Với mọi $m1, m2: Match [m1.ind = m2.ind \rightarrow m1 = m2]$

Nếu không có từ khóa nào, thì không có bản nhập trong sánh

$Key = \{ \} \rightarrow Match = \{ \}$

Match chứa chính xác các văn bản sánh

Với mọi $m: Match [matches(m.doc)] \&&$

Với mọi $d: Doc [matches(d) \rightarrow \text{tồn tại } m: Match (d = m.doc)]$

Match được sắp xếp theo số đếm xuất hiện từ khóa

Với mọi $m1, m2: Match [m1.ind < m2.ind \rightarrow m1.sum \geq m2.sum]$

Hình 2.8 Các quan hệ và ràng buộc suy diễn cho máy tìm kiếm

```
// Máy tìm kiếm có một file riêng chứa danh sách các từ không quan tâm.  
// static Operations  
startEngine ()  
    // EFFECTS: Bắt đầu máy tìm kiếm chạy với NK chứa các từ trong file riêng.  
    // Mọi tập khác đều rỗng.  
// Dynamic Operations  
Query (String w)  
    // CHECKS: w không có trong NK  
    // EFFECTS: Đặt Key = { w } và làm cho Match chứa các văn bản sánh w,  
    // sắp xếp như yêu cầu. Làm sạch CurMatch.  
QueryMore (String w)  
    // CHECKS: Key != {} và w không thuộc NK và w không thuộc Key.  
    // EFFECTS: Bổ sung w vào Key và làm cho Match gồm các bản đã  
    // ở trong Match sánh bổ sung với w. Sắp xếp Match cho đúng.  
    // Làm sạch CurMatch.  
makeCurrent (String t)  
    // CHECKS: t trong Title  
    // EFFECTS: làm cho Cur chứa văn bản với tiêu đề t.  
makeCurMatch (String i)  
    // CHECKS: NUM(i) và i là chỉ số trong Match.  
    // EFFECTS: Làm cho CurMatch chứa bản nhập thứ i trong Match.  
addDocuments (String u)  
    // CHECKS: u chưa là tên của một trang trong URL và u đặt tên cho trang  
    // mà cung cấp các văn bản.  
    // EFFECTS: Bổ sung u vào URL và các văn bản trong trang u với các tên  
    // mới vào Doc. Nếu Key là khác rỗng, bổ sung các văn bản mà sánh với các
```

// từ khóa vào Match và làm sạch CurMatch.

Hình 2.9 Đặc tả yêu cầu cho máy tìm kiếm

2.5 Tóm tắt chương

Kết quả chính của phân tích yêu cầu là đặc tả yêu cầu. Đặc tả này cần là hoàn chỉnh và chính xác: hoàn chỉnh sao cho nó bao quát mọi quyết định về các yêu cầu, và chính xác sao cho người thiết kế có thể hiểu được sản phẩm cần xây dựng. Chương này đã bàn về cách viết đặc tả yêu cầu mà làm cho chúng ta đáp ứng được mục tiêu đề ra.

Cách tiếp cận này là dựa vào đặc tả trên mô hình dữ liệu. Mô hình này định nghĩa trạng thái yêu cầu và cách nó thay đổi theo thời gian. Nó làm điều đó sử dụng ký hiệu cho phép các ràng buộc được đặc tả. Các ràng buộc được xác định tại một số nơi: khi định nghĩa đồ thị, khi định nghĩa các ràng buộc văn bản, và khi đặc tả các thao tác. Định nghĩa các ràng buộc là hữu ích cả khi phân tích, ở đó giúp nhà phân tích nghĩ chi tiết về những cái đã xem xét kỹ, và trong việc viết đặc tả yêu cầu, ở đó họ có thể cung cấp kiểm tra kép về việc các thao tác được đặc tả có đúng không.

Sự cần thiết định nghĩa các ràng buộc thường chỉ ra nơi mà các yêu cầu cần được suy nghĩ kỹ. Nó đưa nhà phân tích đến các bàn luận tiếp với khách hàng để đưa ra các chi tiết. Như vậy, sự cần thiết định nghĩa các ràng buộc dẫn trực tiếp đến đặc tả hoàn chỉnh hơn. Các ràng buộc này có thể cũng dẫn đến sự hiểu biết tốt hơn về phần của người thiết kế, vì việc sử dụng mô hình dữ liệu cho phép đặc tả chính xác hơn. Như kết quả, chúng ta có thể chuyển qua pha thiết kế với niềm tin là sản phẩm mà đang xây dựng sẽ là cái mà khách hàng mong muốn.

CHƯƠNG 3: ĐẶC TẢ THIẾT KẾ

Trong chương trước, chúng ta đã bàn về đặc tả về trùu tượng. Chúng ta nhấn mạnh trùu tượng, vì chúng xây dựng nên các khối mà tạo nên chương trình. Chúng ta sẽ bàn về việc sáng tạo ra trùu tượng như thế nào và làm sao kết hợp chúng để xây dựng một chương trình tốt. Tiếp cận của chúng ta sẽ dựa trên các tài liệu đã có ở pha trước, đặc biệt bàn về các trùu tượng và đặc tả tốt.

3.1 Tổng quan về quá trình Thiết kế

Mục đích của thiết kế là định nghĩa cấu trúc chương trình bao gồm một số modules mà có thể được cài đặt độc lập và sau khi cài đặt sẽ cùng nhau thỏa mãn đặc tả yêu cầu. Cấu trúc này cần cung cấp hành vi yêu cầu và cũng đáp ứng các ràng buộc hiệu năng. Tiếp theo, cấu trúc này cần phải là tốt: nó cần phải là tương đối đơn giản và tránh lặp (chẳng hạn, nó không chứa hai modules mà làm các việc gần giống nhau). Cuối cùng, cấu trúc này cần hỗ trợ cả phát triển chương trình ban đầu và bảo trì và thay đổi. Sự dễ dàng thay đổi sẽ phụ thuộc vào kiểu thay đổi mong muốn. Một sự phân rã cụ thể không thể hỗ trợ mọi thay đổi dễ dàng như nhau. Điều này nói lên tại sao các thay đổi cần được định danh trước khi thiết kế - sao cho một sự phân rã cần được phát triển để làm thuận tiện cho chúng. Bảng 3.1 sau tổng kết mục tiêu của thiết kế.

- *Đáp ứng các yêu cầu chức năng và hiệu năng*
- *Xác định cấu trúc chương trình sao cho*
 - *Các thành phần là các trùu tượng tốt*
 - *Một cấu trúc là đơn giản chấp nhận được và tương đối dễ dàng cài đặt và sửa đổi*
 - *Một cấu trúc làm cho nó tương đối dễ dàng kết hợp với các thay đổi đã được xác định khi phân tích yêu cầu.*

Bảng 3.1 Các mục tiêu thiết kế

Chúng ta bắt đầu một thiết kế với các đặc tả yêu cầu. Đôi khi nó mô tả một trùu tượng duy nhất, đôi khi là một số trùu tượng mà cùng nhau tạo nên hệ thống (chẳng hạn trùu tượng hệ thống file cộng thêm trùu tượng file và thư mục). Chúng ta chọn trong số các trùu tượng để bắt đầu làm việc. Nó trở thành trùu tượng đích ban đầu.

Trong khi thiết kế một chương trình, chúng ta thực hiện ba bước sau cho một đích:

- Xác định các trùu tượng hỗ trợ, hoặc gọi tắt là các hỗ trợ, mà sẽ là hữu ích trong cài đặt đích và sẽ làm thuận tiện việc phân rã bài toán.
- Đặc tả hành vi của mỗi hỗ trợ.
- Phác thảo cài đặt của đích.

Bước đầu tiên bao gồm sáng tạo một số trùu tượng hỗ trợ mà là hữu ích trong lĩnh vực bài toán của đích. Các hỗ trợ sẽ được nghĩ ra như việc tạo một máy trùu tượng mà cung cấp các đối tượng và các thao tác hướng tới cài đặt đích. Ý tưởng là nếu máy đó là sẵn sàng, việc cài đặt đích để chạy trên đó sẽ là trực tiếp.

Tiếp theo, chúng ta định nghĩa các hỗ trợ một cách chính xác bằng việc cung cấp một đặc tả cho mỗi hỗ trợ. Khi một trùu tượng là cái xác định đầu tiên, ý nghĩa của nó thường là hơi mơ hồ. Bước thứ hai bao gồm làm chi tiết và sau đó viết các quyết định trong một đặc tả mà là hoàn chỉnh và không nhập nhằng nhất có thể.

Khi hành vi của một hỗ trợ được định nghĩa chính xác, chúng ta cần sử dụng chúng để viết các chương trình. Về nguyên tắc, một đích bây giờ có thể được cài đặt, nhưng chúng ta sẽ không làm điều đó trong thời gian thiết kế. Thay vào đó, chúng ta sẽ phác thảo đủ để tin tưởng cài đặt của một thành phần hiệu quả sẽ được xây dựng. Chẳng hạn, đối với một trùu tượng dữ liệu, chúng ta có thể liệt kê một số mục mà cần thiết trong việc thể hiện nó.

Nếu máy trùu tượng tồn tại, chúng ta có thể kết thúc; nhưng trên thực tế, các hỗ trợ cần được xem xét cụ thể hơn. Bước tiếp theo là lựa chọn một hỗ trợ và thiết kế cài đặt của nó. Quá trình này tiếp tục cho đến khi mọi hỗ trợ được nghiên cứu xong.

- *Chọn một trùu tượng đích cài đặt của nó chưa được nghiên cứu.*
- *Xác định các trùu tượng hỗ trợ mà sẽ có ích trong cài đặt đích và mà làm thuận tiện phân rã bài toán đã cho.*
- *Phác thảo các hỗ trợ được sử dụng như thế nào trong cài đặt đích.*
- *Lặp cho đến khi các cài đặt của mọi trùu tượng được nghiên cứu xong.*

Bảng 3.2 Quá trình thiết kế

Quá trình thiết kế liên quan đến tính hiệu quả hai loại. Thứ nhất là tìm một thiết kế kinh tế: cái mà không phức tạp và tương đối dễ cài đặt và dễ thay đổi. Cũng quan trọng như vậy, tuy nhiên, là tìm một thiết kế mà dẫn đến một chương trình thực thi tốt. Liên quan đến hiệu năng thẩm vào cả quá trình: ở mỗi giai đoạn chúng ta muốn cài đặt đích một cách hiệu quả. Điều này sẽ dẫn chúng ta đến lựa chọn một số hỗ trợ và không lựa chọn số khác; đặc biệt chúng ta muốn lựa chọn một số hỗ trợ mà sẽ làm việc được yêu cầu bởi một đích theo một cách hiệu quả.

Đối với các chương trình lớn, chúng ta thường không biết trước cấu trúc chương trình cần phải như thế nào. Thay vào đó, khám phá cấu trúc này là mục đích chính của thiết kế. Như các quá trình thiết kế, đôi khi một lựa chọn cần phải được thực hiện trong một số các cấu trúc, không cái nào trong số đó được hiểu một cách rõ ràng. Sau này, quyết định được phát hiện thấy là sai. Điều này đặc biệt giống các quyết định mà được đưa ra sớm trong thiết kế, khi một cấu trúc chương trình còn được hiểu biết ít và ít được ràng buộc bởi các quyết định khác, và khi tác động của một lỗi lên cấu trúc tổng thể là hầu như rất quan trọng.

Khi các lỗi được phát hiện, chúng ta cần hiệu chỉnh chúng bằng cách thay đổi thiết kế. Chúng ta thường bỏ qua mọi thứ mới làm mà phụ thuộc vào lỗi. Điều này giải thích tại sao chúng ta bất đắc dĩ phải bắt đầu cài đặt sau khi đã có một bản thiết kế hoàn chỉnh, hoặc ít nhất một thiết kế hoàn chỉnh cho một phần của chương trình đang cần cài đặt. Tất nhiên, không quan trọng, chúng ta cần thận thiết kế như thế nào, chúng ta cũng không bao quát hết mọi vấn đề với thiết kế trong suốt quá trình cài đặt. Khi điều đó xảy ra, chúng ta cần suy nghĩ lại phần thiết kế liên quan đến vấn đề này.

Trong khi thiết kế chúng ta không được sao nhãng một số câu hỏi liên quan, như:

- Phân rã chương trình được thực hiện như thế nào? Tức là, chúng ta định danh các trừu tượng phụ trợ mà sẽ giúp cho phân tách bài toán như thế nào?
- Chúng ta chọn đích tiếp theo như thế nào?
- Chúng ta biết như thế nào về việc có bước tiến triển. Chẳng hạn, các hỗ trợ có dễ dàng cài đặt hơn đích mà buộc chúng phải xem xét không?
- Các yêu cầu hiệu năng và chỉnh sửa tác động như thế nào đến một thiết kế?
- Phân rã cần được thực hiện với giá như thế nào?

Các câu hỏi này và các câu hỏi khác sẽ được trả lời thông qua ví dụ. Trước hết chúng ta xem xét việc viết một thiết kế như thế nào.

3.2 Sổ tay thiết kế

Các quyết định được thực hiện trong khi thiết kế cần được ghi chép lại. Tài liệu này cần được thực hiện một cách có hệ thống và lưu giữ trong sổ tay thiết kế. Sổ tay này chứa phần mở đầu mô tả thiết kế chung chương trình và mỗi phần cho mỗi trừu tượng.

3.2.1 Phần mở đầu sổ tay

Phần mở đầu liệt kê mọi trừu tượng đã được định danh. Nó cũng chỉ ra hỗ trợ nào là được sử dụng trong cài đặt đích trừu tượng nào. Tài liệu này có dạng một lược đồ phụ thuộc module (module dependency diagram), mà định danh mọi trừu tượng được kể đến trong khi thiết kế (mà được thể hiện khi thiết kế bắt đầu hoặc đưa vào như hỗ trợ) và chỉ ra quan hệ giữa chúng. Lược đồ phụ thuộc module chỉ ra các modules sẽ được cài đặt (chẳng hạn,

các lớp và các giao diện) mà sẽ tồn tại trong chương trình khi cài đặt. Nó cũng chỉ ra sự phụ thuộc của chúng, ở đó module M1 phụ thuộc vào module M2 nếu thay đổi đặc tả của M2 sẽ buộc thay đổi trong M1. Lược đồ phụ thuộc module sẽ đặc biệt có ích để theo dõi tác động của một sự thay đổi trong đặc tả, vì nó sẽ cho phép chúng ta định danh mọi modules mà cần phải xem xét vì sự thay đổi đó.

Sô tay thiết kế bao gồm:

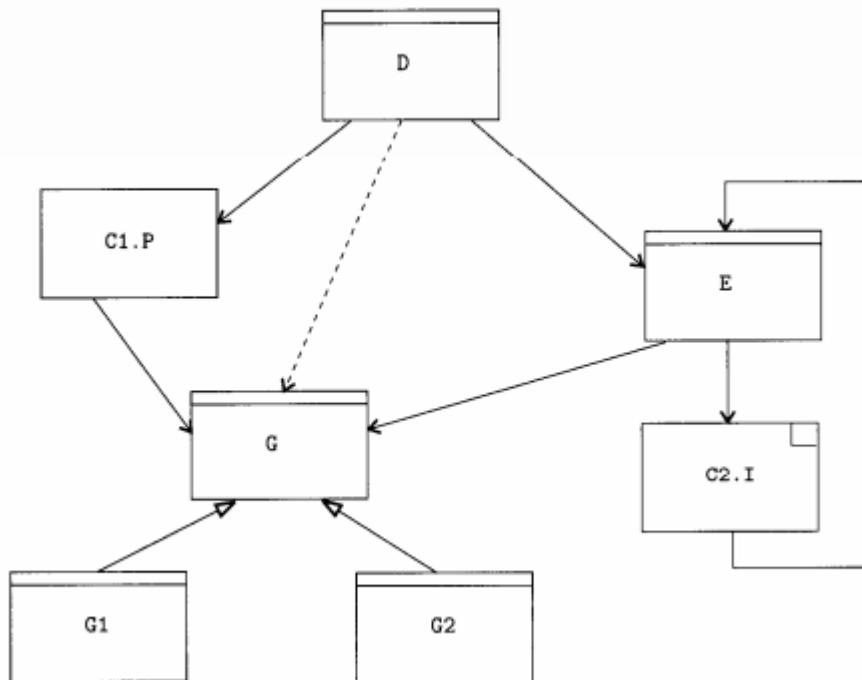
- *Lược đồ phụ thuộc module*
- *Mỗi phần cho mỗi trùu tượng gồm:*
 - *Đặc tả của nó*
 - *Các yêu cầu hiệu năng của nó*
 - *Phác thảo cài đặt của nó*
 - *Thông tin khác bao gồm sự biện minh các quyết định thiết kế và trao đổi về các giải pháp khác, các mở rộng tiềm năng hoặc các điều chỉnh khác, và thông tin về ngữ cảnh sử dụng mong muốn.*

Bảng 3.3 Sô tay thiết kế

Một lược đồ phụ thuộc module bao gồm các đỉnh, mà thể hiện các trùu tượng, và các cung. Ở đây có ba loại đỉnh, mỗi một cho một loại trùu tượng mà chúng ta sử dụng. Mỗi đỉnh đặt tên cho trùu tượng của nó. Các đỉnh giống như đỉnh gán nhãn C1.P trên Hình 3.4 biểu diễn một thủ tục, đỉnh giống C2.I thể hiện bộ duyệt iterator, và cái còn lại biểu diễn trùu tượng dữ liệu. Một trùu tượng dữ liệu sẽ được cài đặt bởi một lớp hoặc một giao diện. Các thủ tục và bộ duyệt được cài đặt bởi các phương thức tĩnh trong một lớp nào đó; tên đỉnh sẽ chỉ ra lớp cũng như tên phương thức (chẳng hạn, C.P cho biết thủ tục P được cài đặt trong lớp C).

Có hai kiểu cung. Kiểu thứ nhất, một cung với đầu mở, là cung sử dụng, nó chỉ ra modules nào được sử dụng trong cài đặt modules khác nào. Cung này đi từ trùu tượng gốc đến một trùu tượng hỗ trợ. Nó có nghĩa là cài đặt của trùu tượng gốc sẽ sử dụng hỗ trợ đó. Chúng ta nói rằng trùu tượng gốc sử dụng hoặc phụ thuộc vào các hỗ trợ đó. Trong trường hợp cung đến một trùu tượng dữ liệu, chúng ta không chỉ ra phương thức nào sẽ được sử dụng, vì thông thường trùu tượng gốc sẽ sử dụng nhiều phương thức của hỗ trợ. Ghi thông tin chi tiết quá sẽ làm cho lược đồ phân tách có thể khó đọc.

Đôi khi, một trùu tượng sử dụng các đối tượng của một trùu tượng dữ liệu mà không goi bất cứ phương thức nào của nó. Trong trường hợp này, chúng ta nói trùu tượng gốc *sử dụng yếu hỗ trợ* đó. Một cung đứt nét được sử dụng để chỉ ra kiểu phụ thuộc này.



Hình 3.4 Một lược đồ phụ thuộc module

Một lược đồ phụ thuộc module là một đồ thị có hướng. Nó không phải là cây, vì một trùu tượng có thể được sử dụng một số cái khác. Đồ thị có thể chứa chu trình, mà xảy ra khi có đệ qui. Chúng ta đưa ra chỉ đệ qui đa chiều; nếu một trùu tượng cụ thể nào đó là đệ qui, đệ qui đó không được chỉ ra trên lược đồ. Nói chung, là chấp nhận được đối với một chương trình có chứa đệ qui, nếu một bài toán được giải quyết có cấu trúc đệ qui tự nhiên. Tuy nhiên, nếu chu trình rất dài và bao gồm nhiều cung, sẽ là khôn ngoan để nghi ngờ xem xét lại cấu trúc chương trình.

Trong Hình 3.4, trùu tượng dữ liệu D là được cài đặt nhờ ba hỗ trợ, thủ tục P, và các trùu tượng dữ liệu E và G. Tiếp theo, E sử dụng G và bộ duyệt I. Cả E và I là đệ qui đa chiều.

Kiểu thứ hai của cung, một cung mở rộng, chỉ ra một quan hệ kiểu con; cung này có đầu đóng. Các cung mở rộng xảy ra chỉ giữa hai đỉnh trùu tượng dữ liệu. Chúng đi từ kiểu con đến kiểu cha và chỉ ra rằng một kiểu con mở rộng hành vi của một kiểu cha. Chẳng hạn, Hình 3.4 chỉ trùu tượng dữ liệu G có hai kiểu con, G1 và G2. Không thể có bất cứ chu trình nào chỉ gồm các cung mở rộng.

Lược đồ phụ thuộc module là có ích khi các lỗi được phát hiện. Một lỗi thiết kế chỉ ra một sai lầm trong một trùu tượng – chẳng hạn, một cài đặt hiệu quả trả về không thể hoặc các đối số cần thiết bị bỏ qua. Kết quả là giao diện của một trùu tượng, và khi đó, đặc tả của nó cần phải thay đổi. Tác động tiềm năng của thay đổi có thể được xác định từ lược đồ. Mọi trùu tượng mà sử dụng trùu tượng lỗi cần được xem xét lại theo giao diện mới và đặc tả của nó. (Việc xem xét này có thể tìm được một số lỗi trùu tượng và tiếp tục như vậy). Một trùu tượng mà sử dụng yếu một trùu tượng lỗi sẽ bị tác động nếu trùu tượng đó biến mất hoàn toàn, nhưng không bị ảnh hưởng, nếu đặc tả của nó thay đổi hoặc được thay thế bởi kiểu khác. Chẳng hạn, nếu chúng ta tìm ra một vấn đề với G trong Hình 3.4, chúng ta sẽ phải suy nghĩ lại về cài đặt của E và P, nhưng D có thể không bị ảnh hưởng. Tuy nhiên, nếu việc suy nghĩ lại E nhắc chúng ta thay đổi đặc tả của nó, chúng ta có thể buộc phải kiểm tra lại cài đặt của D.

Các mũi tên mở rộng được sử dụng theo cách tương tự. Nếu đặc tả của một trùu tượng dữ liệu với các kiểu con thay đổi, mọi kiểu con của nó cần được kiểm tra lại; hoặc các đặc tả của chúng cũng sẽ phải thay đổi, hoặc chúng có thể không còn là kiểu con của nó nữa. Như vậy, một thay đổi trong đặc tả của G có nghĩa là chúng ta cần phải kiểm tra lại G1 và G2. Cũng như vậy, nếu một đặc tả có kiểu cha mà thay đổi, chúng ta cần kiểm tra lại kiểu cha. Có thể là một thay đổi sẽ không có tác động đến kiểu cha; cũng có thể xảy ra, nếu thay đổi chỉ tác động đến các phương thức mới của kiểu con, và kiểu con vẫn thỏa mãn nguyên lý thay thế. Nếu nguyên lý thay thế không còn thỏa mãn nữa, kiểu cha cần được định nghĩa lại, hoặc trùu tượng thay đổi không còn là kiểu con của nó nữa, và lược đồ cần thay đổi để phản ánh điều này. Như vậy, nếu đặc tả của G1 thay đổi, chúng ta cần kiểm tra G; hoặc nguyên lý thay thế còn thỏa mãn hoặc chúng ta cần định nghĩa lại G, hoặc G1 không còn là kiểu con của G.

Mặc dù một lược đồ phụ thuộc module trông giống như đồ thị mô hình dữ liệu, hai cái khác hẳn nhau. Một lược đồ phụ thuộc module mô tả cấu trúc chương trình; các định của nó là các module của chương trình, và các cung định nghĩa quan hệ phụ thuộc giữa các modules đó. Một mô hình dữ liệu, mặt khác, là một trùu tượng; các định của nó xác định các tập hợp và không tương ứng với module của chương trình.

3.2.2 Các bộ phận trùu tượng

Chúng ta chia bản nhập số tay cho một trùu tượng cụ thể thành bốn phần:

1. Đặc tả hành vi chức năng của nó
2. Đặc tả ràng buộc hiệu năng mà cần quan sát thấy
3. Thông tin về nó được cài đặt như thế nào
4. Thông tin vụn vặt khác mà không nằm trong ba loại thông tin trên

Đặc tả, tất nhiên, là phần quan trọng nhất trong danh sách trên. Tuy nhiên, chúng ta đã nói nhiều về nó.

Ràng buộc hiệu năng nói chung được lan tỏa từ trên xuống dưới xuyên suốt chương trình. Các đặc tả yêu cầu có thể ràng buộc thời gian chương trình cần để thực hiện một số nhiệm vụ, dung lượng bộ nhớ chính và phụ, mà nó có thể sử dụng, và sử dụng các tài nguyên khác như mạng. Để tin tưởng rằng cài đặt một trừu tượng đích đáp ứng các ràng buộc hiệu năng của nó, chúng ta cần giả thiết về hiệu năng của các bộ trợ của nó. Giả thiết này chỉ ra các ràng buộc hiệu năng trong các phần của sổ tay thiết kế chi tiết mỗi bộ trợ.

Các ràng buộc hiệu năng có thể được biểu diễn theo nhiều cách. Chúng ta thường biểu diễn chúng như các chức năng của kích thước đầu vào. Chúng ta có thể, chẳng hạn, yêu cầu rằng một trừu tượng tập hợp sử dụng bộ nhớ tuyến tính với kích thước của tập (Order (n), trong đó n là số phần tử trong tập); điều này có nghĩa là dung lượng bộ nhớ chiếm bởi đối tượng tập hợp cần không vượt quá một số lần kích thước của tập. Tiếp theo, chúng ta có thể ràng buộc thời gian để thực hiện các phương thức khác nhau trên tập hợp, như *lookup* và *insert*, là hằng số (Order(1)). Các ràng buộc như vậy có thể được áp đặt sao cho một trừu tượng mà sử dụng tập hợp có thể đáp ứng yêu cầu hiệu năng của nó. Các ràng buộc đó lại tác động đến cách tập hợp được cài đặt; cụ thể, chúng ta cần sử dụng bảng băm *HashMap* để đáp ứng yêu cầu đặt ra.

Đối với nhiều ứng dụng, là đủ để cung cấp ràng buộc hiệu năng tương đối như những gì vừa cho. Đôi khi, tuy nhiên, là hữu ích khi giới hạn hằng số nhân hoặc ngay cả áp đặt giới hạn tuyệt đối. Một giới hạn cận trên tuyệt đối với thời gian là cần thiết trong các ứng dụng thời gian thực. Tương tự, chúng ta đặt các cận trên về không gian sử dụng trong chương trình mà được chạy trên máy nhỏ hoặc các máy không hỗ trợ bộ nhớ ảo.

Phần của bản nhập cho sổ tay chứa thông tin về việc trừu tượng được cài đặt, cần bao gồm danh sách các bộ trợ. Nó có thể cũng chứa mô tả về cài đặt làm việc như thế nào. Mô tả này có thể được bỏ qua nếu cài đặt là trực tiếp, nhưng là cần thiết nếu cài đặt là khôn ngoan. Chẳng hạn, đối với trừu tượng tập, phần cài đặt cần phải chỉ ra là bảng băm cần được sử dụng.

Như tên của nó suy ra, phần này của bản nhập trừu tượng có nhãn là “vụn vặt” có thể chứa hầu hết mọi thứ. Các thứ thông thường là

- Biện minh cho các quyết định được ghi lại đâu đó trong bản nhập
- Bàn luận về các lựa chọn khác mà được xem xét và loại bỏ
- Các mở rộng tiềm năng hoặc các sửa đổi khác của trừu tượng
- Thông tin về bối cảnh ở đó người thiết kế mong muốn trừu tượng được sử dụng

Tóm lại, chúng ta cần lưu ý rằng, nếu thiết kế là rất lớn, nó có thể hữu ích cấu trúc số tay bằng cách đưa ra các số tay phụ trợ. Trong lược đồ phụ thuộc module, bất cứ đồ thị con có thể được xem xét như một hệ thống độc lập. Tuy nhiên, lựa chọn hầu như thuận tiện là một đồ thị con ở đó chỉ có một đỉnh được sử dụng từ bên ngoài. Trong trường hợp này, cấu trúc con toàn bộ đơn giản tương ứng với một trùu tượng duy nhất khi phần còn lại của chương trình liên quan tới.

3.3 Cấu trúc các chương trình tương tác

Trong các phần tiếp theo, chúng ta sẽ thể hiện quá trình thiết kế bằng việc phát triển một chương trình java để cài đặt máy tìm kiếm đơn giản mô tả trong chương trước.

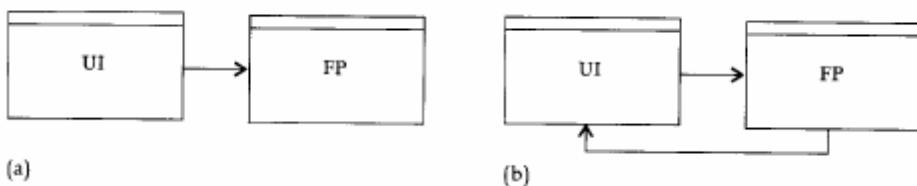
Đặc tả yêu cầu cho một ứng dụng tương tác như máy tìm kiếm mô tả các thao tác mà hướng tới được sử dụng thông qua giao diện người sử dụng. Do đó, một cài đặt của một đặc tả như vậy sẽ bao gồm cả code để cài đặt giao diện người sử dụng và cả code để thực hiện công việc của ứng dụng. Mặc dù có thể xây dựng cấu trúc nguyên khôi ở đó hai phần không tách nhau, thay vào đó chúng ta sẽ làm sự tách bạch giữa chúng. Vì vậy, thiết kế của chúng ta sẽ có hai phần: phần chức năng (FP – Functional Part) và phần giao diện người sử dụng (UI – User Interface). Phần UI sẽ lĩnh trách nhiệm tương tác với người dùng; nó sẽ hiện thông tin cho người sử dụng (chẳng hạn, danh sách hiện tại các từ khóa) và nhận đầu vào của người sử dụng (chẳng hạn, một từ khóa bõ sung để tìm kiếm). Phần FP sẽ tiến hành các lệnh của người sử dụng khi được thông báo từ đầu vào của người dùng và sẽ thông báo cho UI kết quả.

Có một số lý do quan trọng để tách một chương trình thành các phần UI và FP. Thứ nhất là cấu trúc này cho phép chúng ta giữ FP được tự do khỏi chi tiết UI và UI tự do khỏi chi tiết FP. Điều đó có nghĩa là chúng ta có thể thay đổi cài đặt FP (chẳng hạn, tăng tốc nó lên) mà không cần thay đổi UI. Cũng như vậy, chúng ta có thể thay đổi UI (chẳng hạn sử dụng các thư viện khác nhau để tương tác với màn hình và bàn phím) mà không tác động đến FP. Do đó, nếu thư viện thay đổi (chẳng hạn, nó được thay đổi với thư viện khác với nhiều đặc trưng tốt hơn), ứng dụng có thể thay đổi sử dụng thư viện mới bởi thay đổi chỉ phần UI code.

Lý do thứ hai có việc tách này là nó cho phép chúng ta thay đổi cách giao diện người sử dụng thấy và cảm giác mà không thay đổi phần chức năng. Chẳng hạn, nó làm cho tương đối dễ dàng hơn khi thay một UI đơn giản bằng một giao diện tốt hơn. Trên thực tế, chúng ta có một số UI khác nhau, mọi cái trong chúng có thể sử dụng cùng một FP.

Lý do thứ ba để tách là nó cho phép chúng ta phát triển thông qua các kiêm chứng hồi qui phần chức năng. Code của kiêm chứng hồi qui sẽ được lập trình để tương tác với FP giống như UI thực hiện, nhưng thay vì tương tác với người sử dụng, nó tương tác với một bộ dẫn của FP.

Có hai cách kết nối UI với FP, như được thể hiện bởi sơ đồ phụ thuộc module trên Hình 3.5. Trong cấu trúc thứ nhất, FP cung cấp các phương thức mà UI gọi khi đầu vào của người sử dụng được nhập; các phương thức FP thực hiện yêu cầu của người sử dụng và trả thông tin về cho UI mà thông báo về kết quả. Cấu trúc thứ hai mở rộng cái thứ nhất: UI vẫn gọi FP thông báo về đầu vào người sử dụng, nhưng FP có thể hoặc trả về kết quả hoặc gọi bất cứ một phương thức thuận tiện nào của UI. Khi tạo một thiết kế, chúng ta luôn bắt đầu với cấu trúc thứ nhất, vì nó sẽ là phù hợp hơn với nhiều ứng dụng. Chúng ta sẽ chuyển sang cấu trúc thứ hai nếu thấy cần thiết để cải tiến thiết kế.



Hình 3.5 Hai cấu trúc tương tác

Chúng ta bắt đầu thiết kế bằng việc xem xét UI, vì nó tiến hành ứng dụng bằng việc tương tác với người sử dụng. Chúng ta không trói buộc mình với dạng tương tác này của người sử dụng, và trên thực tế thiết kế UI thực thi trong phạm vi văn bản. Vấn đề của chúng ta là đi đến một thiết kế của FP mà là độc lập với bất cứ UI nào, sao cho chúng ta có thể thay đổi UI như mong muốn.

Để xác định giao diện FP, chúng ta xem xét mỗi thao tác trong đặc tả yêu cầu. Trong hầu hết các trường hợp, chúng ta sẽ tạo ra một phương thức FP, mà sẽ được gọi bởi UI thực hiện thao tác đó. FP sẽ làm mọi công việc của ứng dụng, còn UI có trách nhiệm tương tác giữa người sử dụng và FP.

Hình 3.6 cho một đặc tả cho FP máy tìm kiếm. Đặc tả này được suy ra từ đặc tả yêu cầu của máy tìm kiếm trên Hình 2.9. Điểm thứ nhất lưu ý là các phương thức này là tương ứng với các thao tác gắn kết, nhưng có hai khác biệt. Thứ nhất, chúng không có mệnh đề kiểm tra CHECKS (hoặc không có mệnh đề REQUIRES), thay vào đó, chúng đưa ra NotPossibleException khi kiểm tra bị vi phạm. Giá trị của ngoại lệ này là xâu giải thích vấn đề xảy ra, ý tưởng là UI chỉ đơn giản thể hiện xâu giải thích cho người sử dụng vấn đề là gì. Khác biệt thứ hai so với thao tác là các phương thức trả về kết quả mà có thể được sử dụng bởi UI để thể hiện thông tin cho người sử dụng.

Class Engine {

// TÔNG QUAN: Một máy tìm kiếm có trạng thái mà được mô tả trong mô hình dữ liệu của máy tìm kiếm. Các phương thức quăng ra ngoại lệ NotPossibleException // khi có vấn đề; ngoại lệ chứa một xâu giải thích vấn đề. Mọi phương thức xử lý // thay đổi trạng thái của this.

// Các hàm tạo

Engine () throws NotPossibleException

// EFFECTS: Nếu các từ không quan tâm không được lấy từ một trạng thái lưu // trữ sẽ quăng ra NotPossibleException, ngược lại tại NK và khởi tạo trạng thái // ứng dụng một cách phù hợp.

// Các phương thức

Query queryFirst (String w) throws NotPossibleException

// EFFECTS: Nếu $\neg \text{Word}(w)$ hoặc w không trong NK quăng ra ngoại lệ // NotPossibleException, ngược lại đặt Key = { w }, thực hiện truy vấn mới và // trả về kết quả.

Query queryMore (String w) throws NotPossibleException

// EFFECTS: Nếu $\neg \text{Word}(w)$ hoặc w không trong NK hoặc Key = { } hoặc w ở // trong Key quăng ra ngoại lệ NotPossibleException, ngược lại bổ sung w vào // Key, thực hiện truy vấn mới và trả về kết quả.

Doc findDoc (String t) throws NotPossibleException

// EFFECTS: Nếu t không ở trong Title quăng ra ngoại lệ NotPossibleException, // ngược lại trả về tài liệu với tiêu đề t .

Query addDocs (String u) throws NotPossibleException

// EFFECTS: Nếu u không là một URL của một trang chứa các tài liệu hoặc u // đang có trong URL quăng ra ngoại lệ NotPossibleException, ngược lại bổ sung // các tài liệu mới vào trong Doc, nếu không có truy vấn nào đang xử lý, trả về // kết quả của truy vấn rỗng, ngược lại trả về kết quả truy vấn mà bao gồm cả các // tài liệu mới sánh truy vấn đó.

Hình 3.6 Đặc tả của Máy tìm kiếm

Điểm thứ hai là đặc tả của Máy tìm kiếm sử dụng mô hình dữ liệu. Chúng ta sẽ tiếp tục sử dụng mô hình dữ liệu này để phát triển thiết kế.

Điểm thứ ba là các phương thức trả về các đối tượng của hai trùu tượng dữ liệu, Doc và Query, và chúng ta cần định nghĩa các kiểu dữ liệu này. Doc là cách mà UI giữ một tài liệu; để hiển thị một tài liệu, nó cần tuy cập Tiêu đề và văn bản. Query là cách mà UI giữ kết quả truy vấn. Ở đây cần truy cập đến các từ khóa của truy vấn và các văn bản mà sánh truy vấn, nhưng không cần biết sum cho mỗi sánh, vì thông tin này không được thể hiện cho người sử dụng. Hình 3.7 cho đặc tả cho hai kiểu dữ liệu này. Các đặc tả là sơ bộ: chúng bỏ qua hàm tạo và các phương thức mà sẽ tạo ra khi chúng ta xem xét cài đặt của Máy tìm kiếm.

Điểm thứ tư là một số thao tác không có các phương thức tương ứng của Máy tìm kiếm. Điều này xảy ra khi UI có thể dễ dàng làm việc đó tự nó hoặc gọi các phương thức của kiểu dữ liệu phụ trợ. Trong máy tìm kiếm, chẳng hạn, FP không cần có phương thức cho thao tác makeCurMatch, vì UI có thể cài đặt thao tác này sử dụng các phương thức của Query và Doc. Tuy nhiên, như trước đã khẳng định, mọi công việc của ứng dụng được thực hiện bởi FP. Chẳng hạn, UI cho máy tìm kiếm nhận được một từ khóa từ người sử dụng, nhưng nó gọi phương thức của FP để tính toán kết quả truy vấn.

```
class Doc {  
    // TỔNG QUAN: Một tài liệu chưa tiêu đề và thân văn bản.  
    // Các phương thức  
    String title ()  
        // EFFECTS: Trả về tiêu đề của tài liệu this.  
    String body ()  
        // EFFECTS: Trả về thân văn bản của tài liệu this  
}  
  
class Query {  
    // TỔNG QUAN: cung cấp thông tin về các từ khóa của một truy vấn và các tài liệu  
    // mà sánh với các từ khóa đó. Phương thức size trả về số tài liệu sánh. Các tài liệu  
    // có thể được truy cập với các chỉ số từ 0 đến size -1. Các tài liệu được sắp xếp bởi  
    // số sánh mà chúng chứa, với tài liệu 0 chứa số sánh nhiều nhất.  
    // Các phương thức  
    String [ ] keys ()  
        // EFFECTS: trả về các từ khóa của this
```

```
Int size( )  
    // EFFECTS: trả về số các bản sánh của truy vấn này  
  
Doc fetch (int i) throws IndexOutOfBoundsException  
    // EFFECTS: Nếu  $0 \leq i < size - 1$  trả về bản sánh thứ  $i$ , ngược lại tung ra  
    // IndexOutOfBoundsException.  
}
```

Hình 3.7 Đặc tả sơ bộ của Doc và Query

Điểm cuối cùng là chúng ta không đưa các phương thức `toString` và `repOk` vào trong các đặc tả. Tuy nhiên, mọi trùu tượng đều có các phương thức này.

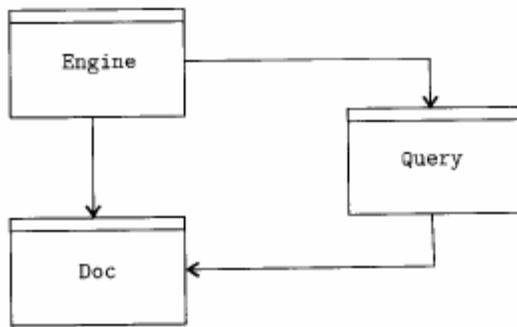
3.4 Bắt đầu thiết kế

Bước đầu tiên để xây dựng lược đồ phụ thuộc module và nhập nó và các đặc tả FP và các trùu tượng khác vào sổ tay thiết kế. Lược đồ phụ thuộc module ban đầu đã được đưa ra trên Hình 3.8.

Lược đồ này được trình bày sao cho, nếu một trùu tượng sử dụng một trùu tượng khác, thì trùu tượng được sử dụng sẽ ở vị trí thấp hơn trong lược đồ so với trùu tượng sử dụng. Chúng ta suy diễn quan hệ sử dụng đối với các bộ trợ từ đặc tả của chúng, cụ thể từ các tiêu đề của các thủ tục và bộ duyệt riêng lẻ, hoặc trong trường hợp trùu tượng dữ liệu là từ các tiêu đề của các phương thức và các hàm tạo của chúng.

Nếu một tiêu đề cho một trùu tượng chỉ ra rằng một đối tượng của một kiểu nào đó được truyền như một đối số hoặc trả về như một kết quả, trùu tượng đó sử dụng kiểu dữ liệu này. Như vậy, Engine sử dụng cả hai `Query` và `Doc`, và `Query` sử dụng `Doc`. Chúng ta giả thiết rằng phụ thuộc mạnh giữa các bộ trợ (tức là, các trùu tượng gọi các phương thức của kiểu chúng sử dụng). Khi chúng ta nghiên cứu cài đặt của một trùu tượng, chúng ta có thể khám phá ra rằng sự phụ thuộc đó là yếu, chứ không phải là mạnh.

Khi chúng ta chọn mục tiêu thứ nhất, FP cho ứng dụng. Bước đầu tiên là tạo ra các bộ trợ. Có một cảm giác trực quan chính chỉ dẫn cho chúng ta trong quá trình này là: Để cấu trúc bài toán quyết định cấu trúc chương trình. Điều này có nghĩa là chúng ta cần tập trung để hiểu bài toán cần giải quyết (chẳng hạn, cài đặt đích như thế nào) và sử dụng hiểu biết nhận được về bài toán để giúp chúng ta phát triển cấu trúc chương trình.



Hình 3.8 Lược đồ phụ thuộc module đầu tiên cho Engine

Một cách tốt để nghiên cứu cấu trúc chương trình là tạo một danh sách các bài toán cần được hoàn thành. Trong trường hợp này, một cấu trúc trùu tượng như Engine, mỗi phương thức là một nhiệm vụ, như vậy chúng ta bắt đầu bằng việc xem xét lần lượt từng cái một. Chúng ta có thể xem xét chúng theo thứ tự bất kỳ, nhưng một số trong số đó thú vị hơn một số khác. Chẳng hạn, xem xét *queryFirst* sẽ buộc chúng ta nghĩ về việc chúng ta sẽ xử lý truy vấn như thế nào, và xem xét *findDoc* sẽ buộc chúng ta nghĩ về tìm một tài liệu cho trước tiêu đề như thế nào.

Khi một phương thức có nhiều việc phải làm, chúng ta liệt kê nhiệm vụ cần phải hoàn thành. Đây là danh sách cho *queryFirst*:

1. Kiểm tra xâu đầu vào w để tin tưởng nó là một từ.
2. Tin tưởng đây là một từ quan tâm.
3. Bắt đầu truy vấn mới với w như một từ khóa.
4. Với mỗi tài liệu, xác định xem nó có sánh không (chứa w).
5. Với mỗi bản sánh, xác định số lần xuất hiện của w.
6. Sắp xếp bản sánh theo số lần xuất hiện của w.
7. Trả thông tin về các bản sánh và truy vấn.

Chúng ta không giả thiết chương trình cuối cùng sẽ có các phần tương ứng với các nhiệm vụ đã liệt kê. Liệt kê các nhiệm vụ chỉ là bước đầu tiên tiến tới thiết kế. Cũng như vậy, mặc dù chúng ta đã liệt kê các nhiệm vụ gần đúng theo thứ tự mà chúng cần phải được thực hiện, chúng ta không giả thiết rằng thứ tự đó sẽ tồn tại trong chương trình cuối cùng. Như chúng ta tiếp tục thiết kế, chúng ta cần phải tổ chức cách mà các nhiệm vụ được thực thi.

Đáng lẽ, bước tiếp theo là sử dụng danh sách trên như chỉ dẫn tạo ra các trùu tượng mà xác định cấu trúc chương trình. Khi tìm kiếm các trùu tượng, chúng ta tìm cách che giấu các chi tiết của việc xử lý mà không quan tâm đến mức độ hiện tại của thiết kế. Mặc dù chúng

ta sử dụng các thủ tục và bộ duyệt để che giấu các chi tiết, các kiểu dữ liệu (và đôi khi họ các kiểu dữ liệu) thường là hữu ích cho mục đích này.

Giả sử bắt đầu bằng việc xem xét nhiệm vụ 4, vì nó rất quan trọng đến hiệu năng của phương thức *queryFirst*. Một cách để xử lý là kiểm tra mọi từ của mọi tài liệu, hoặc ít nhất mọi từ của một tài liệu cho đến khi tìm được một bản sánh. Tuy nhiên, vấn đề là rất tốn thời gian nếu họ các tài liệu là lớn. Tiếp theo khi chúng ta xem xét một tài liệu, chúng ta xem nhiều từ mà không là từ khóa của truy vấn hiện tại, nhưng có thể là từ khóa của các truy vấn về sau. Nếu chúng ta có thể ghi lại thông tin mà chúng ta biết về chúng, chúng ta có thể tránh được công việc trong truy vấn tương lai. Để ghi lại thông tin này, chúng ta cần một trùu tượng dữ liệu mới, *Itable*, mà theo dõi các từ quan tâm trong các tài liệu. Thông tin mà có thể bổ sung vào *ITable* hoặc khi các tài liệu được đưa vào (bằng *addDoc*) hoặc khi truy vấn đang chạy sau khi tài liệu được đẩy vào; quyết định thứ nhất là tốt hơn, vì nó tránh kiểm tra trong khi *queryFirst*. Quyết định này có nghĩa là *queryFirst* không cần duyệt qua các tài liệu nữa. Nó chỉ cần sử dụng thông tin được lưu trước bằng việc gọi phương thức *lookup* của *Itable*.

Bây giờ chúng ta xem xét nhiệm vụ thứ 5. Mặc dù giả thiết trước là số tài liệu sánh với truy vấn nhỏ hơn nhiều so với số tài liệu trong họ, xử lý tài liệu sánh để đếm số lần xuất hiện từ khóa vẫn còn nhiều việc phải làm. Tiếp theo, đây là công việc thừa: khi thông tin về tài liệu được bổ sung vào *ITable*, chúng ta đã xem từng từ trong từng tài liệu, như vậy chúng ta có thể đồng thời đếm số lần xuất hiện của các từ và lưu thông tin này vào bảng *ITable*. Phương thức *lookup* có thể sau đó trả về thông tin các số đếm cũng như kết quả sánh.

Tiếp theo chúng ta xem xét nhiệm vụ 6. Chúng ta cần sắp xếp các bản sánh theo số đếm xuất hiện, nhưng điều này có thể được thực hiện theo nhiều cách. Thay vì quyết định thực hiện nó như thế nào bây giờ, chúng ta sẽ đưa ra một trùu tượng dữ liệu mới, *MatchSet*, nhận trách nhiệm về các chi tiết. Trên thực tế, điều này có thể là ý tưởng tốt cho *MatchSet* thực hiện truy vấn: nó cung cấp tính linh hoạt cho cài đặt của nó (chẳng hạn, cơ hội để sắp xếp, khi các tài liệu được bổ sung vào tập hợp) và cho phép chúng ta trì hoãn quyết định về kiểu thông tin mà phương thức *lookup* của *ITable* trả về cho đến khi chúng ta xem xét dạng nào sẽ là có ích nhất để cài đặt *MatchSet*. Có *MatchSet* thực hiện truy vấn có nghĩa là mọi nhiệm vụ 3-6 sẽ được xử lý bởi một phương thức của *MatchSet*; phương thức này có thể trả về đối tượng *Query* mà chưa kết quả cho một truy vấn.

Cuối cùng, chúng ta xem xét hai nhiệm vụ đầu. Đối với nhiệm vụ thứ hai, chúng ta cần tiền xử lý thông tin về các từ không quan tâm khi chương trình bắt đầu khởi động sao cho chúng ta có thể nhanh chóng xem một từ để xuất có phải không quan tâm không. Thông tin này có thể được lưu trữ trong một trùu tượng dữ liệu riêng hoặc chúng ta có thể trộn thông tin này với những gì về các từ quan tâm. Lựa chọn sau cho chúng ta một bảng giữ thông tin về mọi từ, cả quan tâm và không quan tâm; phương thức *lookup* của nó có thể trả về mọi thông tin này. Ưu thế của lựa chọn này là nó cho phép mỗi từ trong một tài liệu mới được

bổ sung vào bảng với chỉ một lần gọi, thay vì trước hết xem nó có ở trong một bảng không để kiểm tra nó có phải từ không quan tâm không. Do đó, lấy lựa chọn đó và đổi tên *ITable* thành *WordTable* để sánh tốt hơn với chức năng của nó. Một điểm khác là các phương thức của *WordTable* có thể phát hiện các xâu không phải là từ và các không từ này chỉ là một dạng đặc biệt của từ không quan tâm.

Như vậy, bằng việc xem xét phương thức *queryFirst*, chúng ta đã đưa vào một số trừu tượng bổ trợ. Các trừu tượng này cho phép phương thức *queryFirst* được cài đặt rất đơn giản, bằng việc gọi một phương thức của *WordTable* và sau đó một phương thức của *MatchSet*. Cài đặt này cũng là hiệu quả, với giả thiết *WordTable* và *MatchSet* được cài đặt hiệu quả.

Để tiếp tục thiết kế, chúng ta cần xem xét các phương thức khác, *queryMore* cũng tương tự như *queryFirst*, chỉ khác là bây giờ chúng ta quan tâm đến các tài liệu mà đã sánh (và như vậy ở trong *MatchSet*). Tuy nhiên, có hai cách xử lý khóa mới. Chúng ta có thể xây dựng một *MatchSet* khác sánh với khóa mới và trộn hai tập lại để nhận các bản sánh chung. Hoặc chúng ta có một phương thức *MatchSet* khác mà bổ sung khóa mới vào trong truy vấn hiện tại. Lựa chọn sau trông có vẻ tốt hơn, vì nó cũng cung cấp sự linh hoạt cho cài đặt *MatchSet* mà có thể dẫn đến hiệu năng tốt hơn.

Tiếp theo chúng ta xem xét phương thức *findDoc*. Phương thức này không thể được cài đặt bằng việc sử dụng *WordTable* để tìm mỗi từ trong tiêu đề, vì nó sẽ tạo ra nhiều bản sánh, mà không phải tài liệu quan tâm. Do đó, chúng ta cần một cách khác để tìm tài liệu cho trước tiêu đề. Vì nó sẽ là không hiệu quả nếu xem xét mọi tài liệu khi *findDoc* được gọi, chúng ta muốn tiền xử lý các tài liệu khi chúng được bổ sung vào họ các tài liệu. Gọi cấu trúc dữ liệu trừu tượng mà lưu thông tin về các tiêu đề là *TitleTable*.

Cuối cùng xét phương thức *addDocs*. Phương thức này cần để nhận các tài liệu mới bằng việc lấy qua trang Web có tên địa chỉ URL. Tương tác với trang có thể được xử lý bằng bộ duyệt, *getDocs*, mà cung cấp lần lượt các văn bản tài liệu. Phương thức *addDocs* tạo một tài liệu mới và bổ sung nó vào *TitleTable* và *WordTable*; bản thân tài liệu sẽ đơn giản tồn tại trên heap và có thể được truy cập từ cả hai bảng. Sau đó *addDocs* cần bổ sung tài liệu vào một truy vấn nếu nó đang được xử lý; điều này sẽ được làm bởi lời gọi một phương thức của *MatchSet*. Cuối cùng, *addDocs* cần trả về kết quả truy vấn mới, hoặc truy vấn rỗng, nếu không có truy vấn nào đang xử lý; điều này suy ra rằng chúng ta cần một đối tượng Query rỗng.

Bây giờ chúng ta đã xem xét mọi phương thức, chúng ta đã sẵn sàng cho bước thứ hai của thiết kế, được gọi là, khẳng định lại và viết tài liệu các đặc tả này. Trong quá trình viết tài liệu, chúng ta thường chưa khám phá hết mọi thứ. Một số đơn giản là các chi tiết cần tìm hiểu tiếp, nhưng một số có thể là các lỗi thiết kế và yêu cầu chỉnh sửa. Cần phải nói, nhiều

công việc thiết kế sẽ được thực hiện trong bước này; quá trình này giống những gì đã xảy ra trong phân tích yêu cầu khi chúng ta xác định đặc tả yêu cầu.

Hình 3.9 cho đặc tả của *getDocs*. Lưu ý rằng thủ tục này không biết gì về URL của các trang mà cung cấp các tài liệu trước đó. Do đó, *addDocs* cần kiểm tra *URL* được cung cấp so với danh sách *URL* mà nó bảo trì.

Class Comm {

static Iterator getDocs (String u) throws NotPossibleException

*// EFFECTS: Nếu u không là URL hợp lệ hoặc trang tên đó không phản hồi
// như mong muốn quăng ra NotPossibleException ngược lại trả về một
// bộ sinh generator mà tạo ra các tài liệu từ trang như các văn bản.*

}

Hình 3.9 Đặc tả cho bộ duyệt *getDocs*

Hình 3.10 cho đặc tả của *WordTable* và *TitleTable*. Đối với *WordTable*, chúng ta có thể xử lý file của các từ không quan tâm trong Engine và bổ sung các từ vào bảng đồng thời hoặc chúng ta có thể có hàm tạo *WordTable* xử lý tiến trình này. Có hàm tạo *WordTable* xử lý tiến trình này sẽ hạn chế hiểu biết về tên file và định dạng cho *WordTable* sao cho chúng ta có thể thay đổi chúng dễ dàng. Điểm khác về *WordTable* là phương thức *addDoc* của nó yêu cầu rằng đối số của nó khác null; điều này là chấp nhận được, vì phương thức được gọi bởi máy tìm kiếm chỉ sau khi nó tạo tài liệu. Một yêu cầu tương tự tồn tại đối với phương thức *addDoc* của *TitleTable* (và trong nhiều đặc tả được cho về sau). Lưu ý rằng phương thức *addDoc* của *TitleTable* kiểm tra tài liệu đó có bị lặp lại không.

Class WordTable {

*// TỔNG QUAN: Giữ thông tin về các từ quan tâm và không quan tâm. Các từ
// không quan tâm nhận được từ một file riêng. Ghi số lần xuất hiện của mỗi từ
// quan tâm trong mỗi tài liệu.*

// Các hàm tạo

WordTable () throws NotPossibleException

// EFFECTS: Nếu file riêng không đọc được thì quăng ra ngoại lệ

```
// như các từ không quan tâm.  
// Các phương thức  
boolean isInteresting (String w)  
    // EFFECTS: Nếu w là null hoặc không phải từ hoặc không phải từ quan tâm  
    // trả về false, ngược lại trả về true.  
void addDoc (Doc d)  
    // REQUIRES: d không là null  
    // MODIFIES: this  
    // EFFECTS: Bổ sung mọi từ quan tâm của d vào this với số đếm lần xuất  
    // hiện của chúng.  
}  
  
class TitleTable {  
    // TỔNG QUAN: Giữ thông tin về các tài liệu với các tiêu đề của chúng.  
    // Các hàm tạo  
    TitleTable ()  
        // EFFECTS: Khởi tạo this là một bảng rỗng.  
    // Các phương thức  
    void addDoc (Doc d) throws DuplicateException  
        // REQUIRES: d là không null  
        // MODIFIES: this  
        // EFFECTS: Nếu một tài liệu với tiêu đề d đã có trong this quăng ra ngoại lệ  
        // DuplicateException, ngược lại bổ sung d với tiêu đề của nó vào this  
        Doc lookup (String t) throws NotPossibleException  
            // EFFECTS: Nếu t là null hoặc không có tài liệu nào với tiêu đề là t trong this  
            // quăng ra ngoại lệ NotPossibleException ngược lại trả về tài liệu với tiêu đề t
```

Hình 3.10 Các đặc tả của WordTable và TitleTable

Đặc tả này cho WordTable là không đầy đủ. Khi tạo ra các trùu tượng dữ liệu, chúng ta chỉ định nghĩa các thao tác được sử dụng bởi các cài đặt đã được nghiên cứu. Các thao tác bổ sung sẽ được định nghĩa khi chúng ta nghiên cứu các cài đặt khác mà sử dụng kiểu này; điều đó là những gì đã xảy ra cho Query như kết quả nghiên cứu cài đặt của Engine.

Bây giờ chúng ta xem xét đặc tả cho *MatchSet*. Một câu hỏi mà xuất hiện là *MatchSet* liên quan với *Query* như thế nào. Không có cảm giác là lý do tốt để có hai trùu tượng riêng biệt ở đây. Thay vào đó, các phương thức của *MatchSet* mà chúng ta đã định danh trong thiết kế của *Engine* đơn giản là các cách xây dựng các truy vấn, trong kho các phương thức chúng ta nhắc đến trước đó là các cách truy cập thông tin về các truy vấn khi mà chúng đã được tạo ra. Do đó, chúng ta cần nhập hai trùu tượng này thành một trùu tượng mà chúng ta gọi là *Query* để nhất quán với đặc tả của *Engine*.

Cuối cùng, chúng ta cần xem xét đặc tả *Doc*. Tất cả những gì chúng ta cần xác định là sự cần thiết cho một hàm tạo, mà nhận tài liệu như một xâu và trả về một *Doc* cung cấp xâu được tiếp nhận như một tài liệu. Yêu cầu xử lý của chúng ta ở đây rất đơn giản: hàm tạo cần xác định rằng tài liệu có một tiêu đề và một thân, nhưng không cần thiết phải quét thân. Điều này là quan trọng; chúng ta không muốn hàm tạo xử lý toàn bộ tài liệu, vì chúng ta sẽ làm điều đó khi chúng ta tìm các từ khóa của tài liệu.

```
Class Query {  
    // TỔNG QUAN: như trước kia bổ sung thêm  
    // Các hàm tạo  
    Query( )  
        // EFFECTS: Trả về truy vấn rỗng.  
    Query (WordTable wt, String w)  
        // REQUIRES: wt và w khác null  
        // EFFECTS: Tạo một truy vấn với một từ khóa w.  
        // Các phương thức  
        Void addKey (String w) throws NotPossibleException  
            // REQUIRES: w khác null  
            // MODIFIES: this  
            // EFFECTS: Nếu this là rỗng hoặc w là từ khóa đã có trong truy vấn thì
```

```
// quăng ra NotPossibleException, ngược lại thay đổi this để truy vấn chưa
// thêm w và giữ mọi từ khóa đã có trong this.

Void addDoc (Doc d)

    // REQUIRES: d khác null

    // MODIFIES: this

    // EFFECTS: Nếu this là rỗng và w chưa mọi từ khóa của this thì bỏ sung
    // nó vào this như kết quả truy vấn, ngược lại không làm gì.

}

Class Doc {

    // TỔNG QUAN: Giống như trước bỏ sung thêm

    // Các hàm tạo

    Doc (String d) throws NotPossibleException

        // EFFECTS: Nếu d không được xử lý như một tài liệu thì quăng ra ngoại lệ
        // NotPossibleException, ngược lại tạo this là Doc tương ứng với d.

}
```

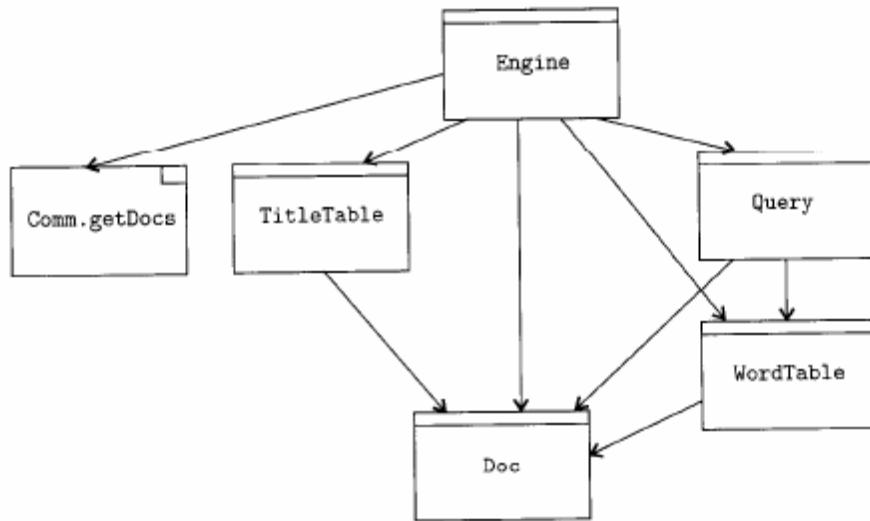
Hình 3.11 Các đặc tả của Query và Doc

Hình 3.11 chỉ ra đặc tả của *Query* và *Doc*. Lưu ý rằng một truy vấn dựa trên một từ khóa mới trên thực tế là được tính toán quan việc sử dụng một hàm tạo, không phải một phương thức và rằng ở đây có hàm tạo thứ hai để tạo một query rỗng (mà cần thiết cho phương thức *addDocs* của *Engine*). Lưu ý rằng các thao tác *Query* không đòi hỏi một từ khóa quan tâm; nếu từ khóa là không quan tâm, kết quả truy vấn sẽ không có bản sánh nào.

Đặc tả này trên các hình vẽ không cho bất cứ ràng buộc hiệu năng nào. Tuy nhiên, để *Engine* thực thi tốt, quan trọng là đối với mọi bảng phải cung cấp các cách hiệu quả tìm kiếm thông tin. Trên thực tế, chúng cần cung cấp tìm kiếm thời gian hằng số (và khi đó chúng ta có thể dựa trên các bảng băm để tiếp tục thiết kế này).

Lược đồ phụ thuộc module mở rộng được chỉ ra trên Hình 3.12. Lưu ý rằng chúng ta không chỉ ra *Engine* có sự phụ thuộc vào *Iterator* mặc dù nó sử dụng một bộ sinh; phụ thuộc này là bỏ qua được vì chúng ta coi bộ duyệt là một phần của ngôn ngữ cơ sở. Các kiểu ngoại lệ

là cũng được xử lý theo cách như vậy, vì chúng là hầu hết các trùu tượng được định nghĩa trong các thư viện chuẩn, như `java.util`, chẳng hạn, chúng ta dùng `Vector` là đương nhiên.



Hình 3.12 Lược đồ phụ thuộc module mở rộng cho Engine

Cũng như trước, chúng ta định nghĩa các mức trong lược đồ dựa trên các phụ thuộc. Cũng như vậy chúng ta giả thiết phụ thuộc mạnh, khi chúng ta tiếp tục thiết kế, chúng ta có thể khám phá ra rằng một số phụ thuộc là yếu.

Như vậy, để khảng định tiếp thiết kế của Engine, sau đây là phác thảo các thẻ hiện:

```
WordTable wt;  
TitleTable t;  
Query q;  
String [ ] urls;
```

Như vậy, các thẻ hiện của nó chỉ lưu trạng thái của máy và các bảng mà sẽ được sử dụng để xử lý các yêu cầu của người sử dụng trong tương lai. Các thẻ hiện thực tế có thể là khác so với những gì chỉ ra đây, nhưng sẽ chứa thông tin trong phác thảo trên.

3.5 Phương pháp thiết kế

Khi xây dựng thiết kế, chúng ta sử dụng phương thức đơn để tập trung chú ý vào điều cần phải thực hiện – chúng ta tách công việc thành các nhiệm vụ con và sau đó nghiên cứu thực hiện các nhiệm vụ nhỏ như thế nào. Chúng ta không đơn giản đưa ra các thủ tục cho mỗi nhiệm vụ con. Thay vào đó, chúng ta xem xét các trùu tượng, đặc biệt các kiểu dữ liệu,

quan tâm đến chi tiết của các nhiệm vụ. Điều này cho kết quả thiết kế tốt hơn, làm cho nó dễ dàng che giấu các chi tiết cho đến các giai đoạn về sau của thiết kế và khai phá các kết nối giữa các nhiệm vụ khác nhau.

Chúng ta đưa ra các trùu tượng để che giấu chi tiết mà chúng ta nghĩ rằng chưa phù hợp ở mức hiện tại. Điều này làm nảy sinh câu hỏi là thế nào để quyết định cái gì phù hợp ở mức độ hiện tại. Quyết định như vậy phần lớn là công việc suy luận, nhưng cũng có một số chỉ dẫn. Một cài đặt cần phải thực hiện một điều gì đó, nhưng nó không cần làm quá nhiều. Mục đích của chúng ta kết thúc với các modules nhỏ; các cài đặt các trùu tượng dữ liệu cần không được dài hơn vài trang và các cài đặt phương thức thông thường ngắn hơn một trang. Chẳng hạn, nếu Engine lĩnh trách nhiệm về mọi chi tiết xử lý yêu cầu của người sử dụng, nó cần phải là rất lớn.Thêm vào đó, các phần khác nhau của cài đặt của một module sẽ là ở cùng một mức chi tiết. Cuối cùng, mỗi trùu tượng cần tập trung vào một mục đích duy nhất;

Cách tiếp cận cơ bản của chúng ta là để cấu trúc bài toán xác định cấu trúc chương trình:

- *Liệt kê các nhiệm vụ cần phải thực hiện*
- *Sử dụng danh sách này trợ giúp tạo ra các trùu tượng, đặc biệt các trùu tượng dữ liệu, để hoàn thành các nhiệm vụ.*
- *Đưa ra các trùu tượng che giấu chi tiết: trùu tượng từ làm như thế nào đến cái gì.*
- *Mỗi trùu tượng cần tập trung vào một mục đích duy nhất, và cài đặt của nó thực tế cần làm một cái gì đó.*

Bảng 3.13 Phương pháp thiết kế

Thiết kế *Engine* là tiêu biểu cho thiết kế một trùu tượng ở mức độ cao trong một hệ thống. Các cài đặt của các trùu tượng như vậy là liên quan chủ yếu với tổ chức tính toán, trong khi các chi tiết thực hiện các bước được xử lý bởi các bộ phận. Đưa ra các kiểu dữ liệu được đặc tả bộ phận giống như *Doc* và *Query* cũng là tiêu biểu. Trong các giai đoạn sớm của thiết kế, chúng ta thường biết rằng hai modules cần trao đổi với nhau, nhưng không biết chính xác trao đổi đó được tiến hành như thế nào. Cụ thể, chúng ta biết rằng, các modules trao đổi qua các đối tượng của một kiểu nào đó, nhưng chúng ta không biết được phương thức nào trên đối tượng được sử dụng. Do đó, đặc tả của kiểu dữ liệu chia sẻ là cần thiết chưa hoàn chỉnh. Nó sẽ được hoàn chỉnh khi chúng ta tiếp tục thiết kế.

3.6 Tiếp tục thiết kế

Bây giờ chúng ta có lược đồ phụ thuộc module chứa một số trùu tượng. Chúng ta lựa chọn bước tiếp theo như thế nào. Điều đầu tiên nhận thấy là không phải mọi trùu tượng đều phù hợp cho bước tiếp theo; những cái mà phù hợp chúng ta sẽ gọi là các ứng cử viên. Rõ ràng, bản thân Engine không phải là ứng cử viên, vì chúng ta đã thiết kế xong cài đặt của nó. Ngoài ra, *Doc* và *WordTable* cũng không phải là các ứng cử viên, vì chúng ta chưa nghiên cứu làm sao cài đặt các modules sử dụng chúng; và do đó chúng ta không thể chắc chắn các đặc tả của chúng là hoàn chỉnh. Thực tế, chúng ta sẽ lựa chọn tùy thuộc một trùu tượng chưa hoàn chỉnh như một ứng cử viên.

- *Định danh mọi ứng cử viên; đây là các trùu tượng mà cài đặt của chúng chưa được nghiên cứu kỹ, nhưng đặc tả đã hoàn chỉnh.*
- *Chọn mục tiêu T trong số các ứng cử viên. Nguyên nhân chọn một mục tiêu cụ thể bao gồm khai phá sự chưa rõ ràng, tăng hiểu biết bên trong cấu trúc chương trình, hoặc kết thúc một phần thiết kế.*

Bảng 3.14 Lựa chọn mục tiêu tiếp theo

Vì vậy, chúng ta có ba ứng cử viên: *getDocs*, *TitleTable* và *Query*. Chúng ta sẽ chọn giữa chúng như thế nào? Ở đây không có các qui tắc rõ ràng; Ứng cử viên nào cũng có thể được chọn nghiên cứu tiếp. Tuy nhiên, có một số lý do, tại sao chúng ta ưu tiên một ứng cử viên hơn các ứng cử viên khác:

1. Chúng ta không chắc chắn về cài đặt trùu tượng đó. Chẳng hạn, chúng ta cần cài đặt một trùu tượng rất hiệu quả, và chúng ta không tin tưởng làm thế nào để đạt được hiệu quả như vậy, không biết nó có đạt được không.
2. Chúng ta không tin tưởng về tính phù hợp của nó.
3. Một ứng cử viên có thể là trung tâm hơn những cái khác, như vậy nghiên cứu cài đặt của nó sẽ giúp thấu hiểu hơn về thiết kế.
4. Chúng ta có thể tiếp tục làm việc trên một lĩnh vực nào đó của thiết kế và sẽ kết thúc lĩnh vực đó.

Hai qui tắc đầu liên quan đến việc nghiên cứu các lĩnh vực đã được xem xét là đáng ngờ; lựa chọn một trùu tượng về một trong hai lý do, đó là giống như khám phá nhanh lối thiết kế. Nếu một sự không chắc chắn như vậy tồn tại. Nó hầu như luôn là cách tốt nhất nghiên cứu ngay tức thì. Trên thực tế, chúng ta có thể cần nghiên cứu ngay một cấu trúc dữ liệu mà không là ứng cử viên, nếu chúng ta cảm giác chưa đủ chắc chắn về nó. Hai qui tắc khác thực tế là đối ngược: kết thúc một lĩnh vực là không giống như tìm hiểu sâu về bên trong, nhưng nó có thể là hữu ích nhận một phần thiết kế đầy đủ. Tiếp theo, làm điều này có thể cho phép chúng ta bắt đầu cài đặt phần này của chương trình.

Trong ví dụ của chúng ta, mọi ứng cử viên đều là trung tâm của thiết kế, mặc dù *getDocs* gây nên các vấn đề về làm sao trao đổi với các trang khác, trong khi hai ứng cử viên sẽ khám phá chi tiết về các cấu trúc dữ liệu. Để ngắn gọn, chúng ta sẽ không chọn xét thiết kế của *getDocs*; thay vào đó, chúng ta sẽ giả thiết, nó được cung cấp bởi một thư viện nào đó (chẳng hạn, một lớp *Comm*). Do đó chúng ta cần lựa chọn giữa *TitleTable* và *Query*. Cả hai sẽ cung cấp hiểu biết bên trong; chúng ta sẽ bắt đầu với *TitleTable*, vì nó trông đơn giản hơn, và nó cũng làm sáng tỏ về *Doc*.

Ở đây có hai vấn đề trong *TitleTable*: làm sao lấy tiêu đề từ một tài liệu (trong phương thức *addDoc*) và làm sao tô chúc bảng này để tìm kiếm sẽ nhanh. Tiêu đề có thể nhận được bằng cách gọi phương thức *title* của *Doc*. Một tìm kiếm *lookup* nhanh có thể đạt được sử dụng bảng băm, hoặc chúng ta có thể cài đặt trực tiếp bên trong *TitleTable* hoặc nó được cung cấp bởi *java.util*. Tuy nhiên, chúng ta cần quyết định kiểu nào được sử dụng cho khóa và giá trị. Trong trường hợp này, giá trị là *Docs*. Chúng ta có thể sử dụng một xâu cho khóa hoặc đưa vào một trùu tượng mới là *Title*. Quyết định đầu là chấp nhận, nếu giả thiết rằng các tiêu đề là đủ ngẫu nhiên để phương thức băm cho xâu sẽ tạo nên các kết quả đủ ngẫu nhiên. Chúng ta sẽ chấp nhận giả thiết này, vì nó có vẻ hợp lý.

Chúng không cần mở rộng lược đồ phụ thuộc module tại thời điểm này, vì chúng ta chưa đưa ra bất cứ trùu tượng dữ liệu nào. Ngay cả nếu chúng ta quyết định sử dụng bảng băm trong *java.util*, không cần phải bổ sung nó vào lược đồ, vì nó có ở trong thư viện chuẩn. Tuy nhiên, chúng ta cần giải thích trong tài liệu về cài đặt của *TitleTable* rằng chúng ta sẽ sử dụng bảng băm ánh xạ từ các xâu vào *Docs*. Và nếu chúng ta quyết định dùng bảng băm trong *java.util*, chúng ta cũng cần khẳng định điều này. Trong trường hợp này, sử dụng bảng băm trong *java.util* sẽ là thích hợp.

3.6.1 Trùu tượng Query

Bây giờ chúng ta sẽ chỉ còn một trùu tượng, *Query*, để xem xét, vì không còn ứng cử viên nào khác. *Query* có bốn nhiệm vụ lý thú sau để thực hiện:

1. Tính toán truy vấn mới cho trước một khóa.
2. Mở rộng truy vấn với từ khóa mới (phương thức *addKey*).
3. Mở rộng truy vấn với tài liệu mới (phương thức *addDoc*).
4. Cung cấp truy cập đến thông tin về truy vấn. Một quan tâm cụ thể là cung cấp truy cập các tài liệu sắp xếp theo thứ tự số lần xuất hiện các từ khóa.

Ngoài ra, *Query* cần tạo ra truy vấn rỗng, nhưng nhiệm vụ này hiển nhiên.

Có thể điểm bắt đầu tốt là xem xét tạo truy vấn như thế nào nếu cho trước một từ khóa. Để làm điều đó, *Query* cần phải

1. Tìm tất cả các tài liệu chứa từ khóa với số đếm lần xuất hiện.
2. Giữ thông tin về từ khóa

3. Sắp xếp các tài liệu dựa trên số lần xuất hiện các từ khóa.

Nhiệm vụ thứ nhất có thể hoàn thiện bằng việc gọi phương thức *lookup* của *WordTable*. Nếu *lookup* trả về các tài liệu sánh sắp xếp theo bộ đếm sánh, nhiệm vụ thứ ba sẽ là hiển nhiên. Tuy nhiên, có việc này được làm trong *WordTable* không là ý tưởng tốt vì hai lý do. Thứ nhất, nhiều từ sẽ không được sử dụng như từ khóa, nên sắp xếp sự xuất hiện của chúng không cần thiết. Thứ hai, sắp xếp không hữu ích khi mở rộng truy vấn, vì chúng ta quan tâm đến tổng các bộ đếm của mọi từ khóa. Do đó, *lookup* sẽ trả về các tài liệu không sắp xếp, và *Query* cần linh trách nhiệm sắp xếp.

Nếu chúng ta có thể giả thiết rằng số các tài liệu sánh là nhỏ, thì kỹ thuật được sử dụng để sắp xếp là không quan trọng. Tuy nhiên, giả thiết này có vẻ không đúng, và do đó chúng ta cần cẩn thận hơn. Một cách tích cực để sắp xếp là sử dụng một cây được sắp xếp. Kỹ thuật này làm việc hiệu quả, nếu thứ tự được cung cấp của các tài liệu trong danh sách trả về bởi *WordTable* là ngẫu nhiên theo số sánh: giả thiết này là chấp nhận được. Do đó, sử dụng cây sắp xếp là một lựa chọn có lý.

Bây giờ chúng ta xem xét phương thức *addKey*. Ở đây các nhiệm vụ là:

1. Kiểm tra xem khóa mới đã sử dụng chưa.
2. Tìm các tài liệu chứa khóa mới.
3. Tìm các tài liệu sánh khóa mới mà đã có trong truy vấn
4. Sắp xếp các tài liệu còn lại theo tổng sánh.

Nhiệm vụ thứ 2 có thể được thực hiện sử dụng phương thức *lookup* của *WordTable*. Để làm tốt nhiệm vụ 3, chúng ta cần một cách nhanh chóng kiểm tra một tài liệu sánh khóa mới có ở trong truy vấn không. Điều này không thể được thực hiện hiệu quả nếu sử dụng cây sắp xếp, vì là sắp xếp theo số sánh với các từ khóa biết trước đó, mà không dựa trên cái mà định danh tài liệu. Thay vào đó, chúng ta cần lưu các bản sánh trước (hoặc các bản sánh mới) vào bảng băm sao cho chúng ta có thể tìm kiếm chúng một cách hiệu quả. Sử dụng bảng băm cho chúng ta tìm kiếm thời gian hằng số hơn là tuyến tính.

Tiếp theo chúng ta xem xét phương thức *addDoc*. Phương thức này cần kiểm tra mỗi từ khóa có trong tài liệu mới không; nếu chúng có, cần bổ sung tài liệu đó vào tập sánh theo đúng thứ tự sắp xếp của nó (tức là bổ sung tài liệu với tổng sánh vào cây sắp xếp). Tuy nhiên, có vấn đề với sơ đồ này: làm sao *addDoc* kiểm tra được các từ khóa có trong tài liệu mới không? Chúng ta có thể tìm kiếm theo mỗi từ khóa, và sau đó xác định xem tài liệu mới có ở trong danh sách kết quả không; nhưng vì các danh sách này rất dài, cách kiểm tra này sẽ phải trả giá. Như một lựa chọn, chúng ta có thể cung cấp phương thức khác của *WordTable* để tìm kiếm một tài liệu và trả về các từ quan tâm của nó, nhưng phương thức này đòi hỏi cài đặt phức tạp hơn đối với *WordTable*, hoặc nó sẽ là rất tốn thời gian. Cũng có thể, chúng ta không cần phương thức này; chỉ thời gian mà chúng ta quan tâm tìm kiếm các từ trong tài liệu là lúc đầu tiên bổ sung nó vào cơ sở dữ liệu. Do đó, thay vào đó ta có

phương thức *addDoc* của *WordTable* trả về bảng băm mà giám sát các từ khóa trong tài liệu mới. Bảng này được xây dựng trong thời gian tuyến tính với kích thước của tài liệu, và kiểm tra tài liệu có sánh truy vấn đang có cũng là tuyến tính với số từ khóa. Bảng này có thể thực tế ánh xạ mỗi từ trong tài liệu vào số lần xuất hiện của nó sao cho trong trường hợp tài liệu sánh truy vấn, tổng sánh có thể tính dễ dàng. Bảng này sẽ cần là đối số cho phương thức *addDoc* của *Query*.

Như vậy, chúng ta đã xác định sự cần thiết thay đổi đặc tả của *Query* và của *WordTable*, mà có nghĩa là chúng ta cần kiểm tra *Engine* để xác định tác động của các thay đổi này. Nhưng không có vấn đề gì: *Engine* đơn giản là truyền bảng trả về bởi lời gọi phương thức *addDoc* của *WordTable* cho phương thức *addDoc* của *Query*.

Cuối cùng, chúng ta xem xét các phương thức quan sát *Observers*. Giữa các tài liệu trong cây sắp xếp hoặc trong bảng băm sẽ không cho phép cài đặt hiệu quả cho *fetch*.

Mặc dù chúng ta có thể tưởng tượng thay thế phương thức *fetch* bằng một bộ duyệt các phần tử (trên cây sắp xếp), *fetch* thực tế là cần thiết cho giao diện UI. Do đó sau khi sắp xếp, chúng ta cần chuyển các tài liệu vào một mảng sao cho *fetch* có thể được cài đặt một cách hiệu quả.

Tuy nhiên, nếu chúng ta định chuyển các tài liệu vào mảng, không có nghĩa nữa nếu sử dụng cây sắp xếp để sắp xếp chúng. Thay vào đó, chúng ta có thể lưu chúng vào mảng và sắp xếp chúng tại chỗ, sử dụng một thuật toán sắp xếp tốt như *quickSort*. Thực tế, chúng ta sẽ sử dụng một vector sao cho trong phương thức *addDoc*, chúng ta có thể đơn giản chèn tài liệu mới vào vị trí đúng của nó trong vector (sử dụng phương thức *insertElementAt* của vector). Cài đặt của phương thức *addKey* sẽ chuyển các tài liệu vào một bảng băm, lưu các tài liệu chứa mọi khóa vào một vector, và sau đó trong hàm tạo, sắp xếp vector.

Như vậy chúng ta kết thúc với các thể hiện sau của một truy vấn khác rỗng:

```
WordTable k;  
Vector matches; // các phần tử và các đối tượng DocCnt  
String [ ] keys;
```

DocCnt là kiểu bản ghi với hai trường, tài liệu và số đếm xuất hiện của các từ khóa trong tài liệu. Bất biến thể hiện sẽ khẳng định rằng *matches* được sắp xếp theo số đếm, và các tổng là đúng đắn, và các sánh chứa mọi từ khóa – nói cách khác, nó sẽ khẳng định một số ràng buộc từ mô hình dữ liệu.

Vì cài đặt của *Query* không rõ ràng, sẽ là hữu ích ghi lại các quyết định của chúng ta bằng cách phác thảo một số phương thức. Hình 3.15 nêu ra các phác thảo này. Chúng có thể

được sử dụng để hỗ trợ phát triển các đặc tả cho các phương thức và trừu tượng mới định nghĩa và chúng có thể được nhập vào sổ tay thiết kế.

Đối với hàm tạo (của query khác rỗng):

Tìm khóa key trong WordTable

Sắp xếp các bản sánh sử dụng quicksort

Đối với phương thức addKey

Tìm kiếm khóa mới new key trong WordTable

Lưu thông tin các bản sánh trong bảng băm

Đối với mỗi bản sánh hiện thời, tìm kiếm tài liệu trong bảng băm và nếu có ở đó, lưu nó vào trong vector

Sắp xếp vector sử dụng quicksort

Đối với phương thức addDoc

Sử dụng bảng băm đổi số để nhận số xuất hiện của mỗi từ khóa hiện thời

Nếu tài liệu có mọi từ khóa, tính tổng và chèn cặp <doc, sum> vào vector các bản sánh.

Hình 3.15 Phác thảo của một số phương thức Query

Các đặc tả mở rộng cho *Query* và *WordTable* được cho trên Hình 3.16. Hình 3.17 cho đặc tả của *quicksort* và các trừu tượng liên quan. *quicksort* là một trừu tượng tổng quát mà đòi hỏi các phần tử của *vector* thuộc một kiểu mở rộng giao diện *Comparable*. Vì *vector* thực tế được sử dụng trong *Query* chứa các đối tượng *DocCnt*, điều này có nghĩa *DocCnt* cần mở rộng giao diện này. Lưu ý rằng, đặc tả của *DocCnt* không chứa chi tiết về các phương thức truy cập đến các thành phần, vì chúng là chuẩn cho các kiểu loại bản ghi.

```
Class Query {  
    // Như trước kia ngoại trừ đặc tả phương thức addDoc có thay đổi.  
    void addDoc (Doc d, HashTable h)
```

```
// REQUIRES: d không null và h ánh xạ xâu (các từ quan tâm
// trong d) vào số nguyên (số đếm lần xuất hiện của từ đó trong
// d).
// MODIFIES: this
// EFFECTS: Nếu mỗi từ khóa của this có trong h, bổ sung d vào
// các bản sah của this.
}

class WordTable {
    // Như trước kia
    Vector lookup (String k)
        // REQUIRES: k không là null
        // EFFECTS: Trả về một vector của DocCnt ở đó Doc chứa k cnt
        // lần.
    HashTable addDoc (Doc d)
        // REQUIRES: d không là null
        // MODIFIES: this
        // EFFECTS: Bổ sung thông tin về các từ quan tâm của d và số
        // lần xuất hiện của nó vào this và trả về bảng băm ánh xạ mỗi
        // từ quan tâm trong d vào số lần xuất hiện của nó trong d.
}
```

Hình 3.16 Các đặc tả mở rộng cho Query và WordTable

```
class Sorting {
    // TỔNG QUAN: Cung cấp một số thủ tục cho sắp xếp vector.
    static void quicksort (Vector v) throws ClassCastException,
    NullPointerException
        // MODIFIES: v
        // EFFECTS: Nếu một thành viên của v là null throws
```

```
// NullPointerException; nếu các phần tử của v là không so sánh
// được throws ClassCastException, ngược lại sử dụng thuật toán
// quicksort để sắp xếp các phần tử của v sao cho các phần tử với
// chỉ số lớn hơn là nhỏ hơn so với các phần tử có chỉ số nhỏ hơn
}

class DocCnt implements Comparable {
    // TỔNG QUAN: DocCnt là kiểu loại bẩn ghi với hai trường Doc và
    // số nguyên.

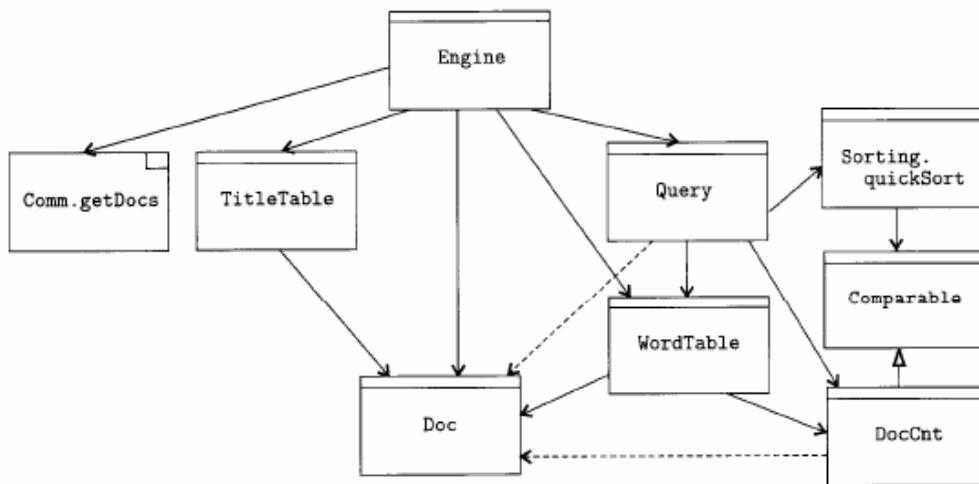
    // Các phương thức
    int compareTo (Object x)      throws ClassCastException,
        NullPointerException

    // EFFECTS: Nếu x là null throws NullPointerException;
    // nếu x không là đối tượng DocCnt throws ClassCastException,
    // ngược lại, nếu this.cnt < x.cnt trả về -1, nếu this.cnt = x.cnt trả
    // về 0, ngược lại trả về 1.

}
```

Hình 3.17 Các đặc tả cho quicksort và DocCnt

Lược đồ phụ thuộc module mở rộng được nêu ra trên Hình 3.18. Điểm thú vị nhất ở đây là sử dụng phân cấp để giải thích làm sao *DocCnt* thỏa mãn sự phụ thuộc của *quicksort* vào *Comparable*. Điểm khác là chúng ta biết rằng *Query* có sự phụ thuộc yếu vào *Doc* (vì nó không gọi phương thức nào của *Doc*), như trên Hình 3.18, chúng ta đã giả thiết là phụ thuộc mạnh. *DocCnt* có sự phụ thuộc yếu vào *Doc*.



Hình 3.18 Lược đồ phụ thuộc module mở rộng

3.6.2 Trùu tượng *WordTable*

Tại thời điểm này, chúng ta chỉ còn một ứng cử viên là *WordTable*, với giả thiết chấp nhận được là *quicksort* được cung cấp trong thư viện. Việc cài đặt *WordTable* là tương đối rõ ràng. Nó sẽ sử dụng bảng băm để ánh xạ các xâu thể hiện các từ không quan tâm vào null và các xâu là các từ quan tâm vào vector, ở đó mỗi thành phần của vector là một *DocCnt*.

Phương thức *addDoc* sẽ sẽ lấy mỗi từ từ *Doc* và sử dụng bảng băm để xác định nó có là quan tâm không. Nó sẽ lưu các từ quan tâm vào cả bảng băm, *H*, mà sẽ trả về và vào bảng băm mà lưu trạng thái của *WordTable*. Có thể cách tiếp cận tốt nhất cho *addDoc* là lưu thông tin trong bảng *H* khi nó xử lý tài liệu, và sau đó bổ sung thông tin vào trạng thái của nó ở cuối.

Cài đặt *WordTable* đòi hỏi cách duyệt trên các từ trong tài liệu. Như vậy, *Doc* cần cung cấp một bộ duyệt mà có thể được sử dụng cho mục đích này.

Chúng ta chưa xem xét một vấn đề nữa đối với *WordTable*: nó liên quan đến dạng chuẩn cho các từ. Chẳng hạn, chúng ta không muốn nhập cả hai từ “hash” và “Hash” vào trong *WordTable*. Một khả năng là có kiểm tra sự bằng nhau mở rộng mà được thừa nhận, chẳng hạn, như “hash” và “Hash” là hai từ như nhau, nhưng có vẻ khá tốn công. Giải pháp tốt hơn là có định dạng chuẩn cho mỗi từ - chẳng hạn, mọi chữ đều là chữ thường. Một điểm khác là chuẩn hóa này cần cả ở những chỗ khác – chẳng hạn, trong các phương thức *queryFirst* và *queryMore* của *Engine* và cũng như trong phương thức *addDocs* của *TitleTable*. Nó đề xuất rằng một cách tiếp cận tốt là có một thủ tục *canon* mà tạo ra dạng chuẩn. Thủ tục này có thể được gọi bởi các phương thức của *Engine* sao cho một từ được

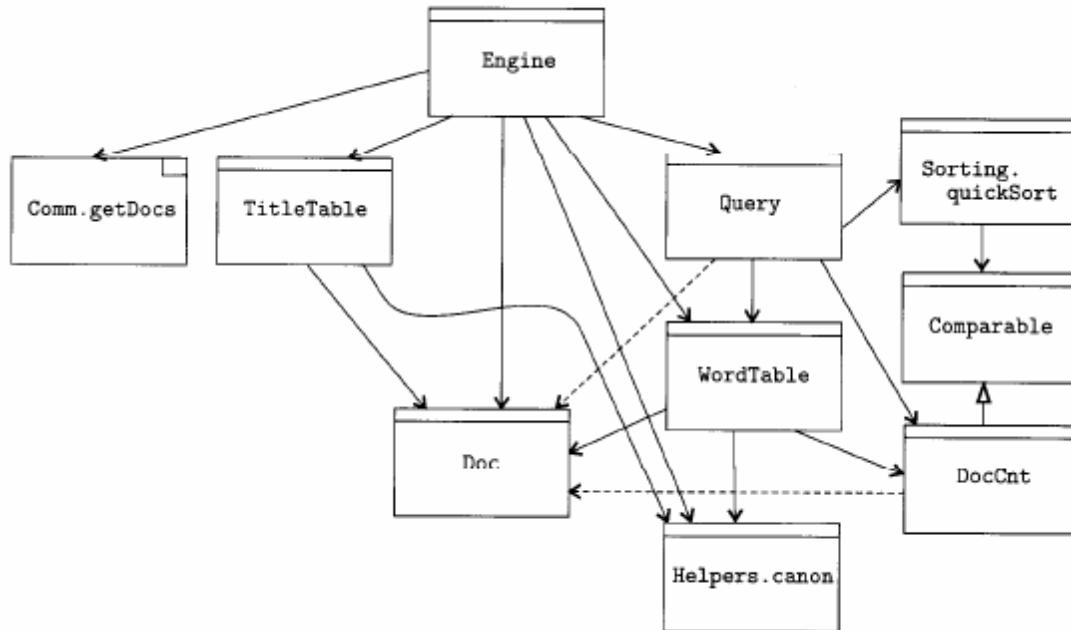
nhập vào bởi người sử dụng có thể được chuyển về dạng chuẩn chỉ một lần và bởi *WordTable* khi nó nhập từ vào bảng, bao gồm cả trong hàm tạo của nó, vì nó cần giữ cả các từ không quan tâm trong dạng chuẩn. Có một thủ tục duy nhất, mà được sử dụng mọi nơi để chuẩn hóa cho phép chúng ta địa phương hóa kiến thức dạng chuẩn là gì, là cái sao cho chúng ta có thể thay đổi nó dễ dàng nếu cần thiết. Như vậy, thiết kế này là tốt hơn ở chỗ, chẳng hạn, cả hai *Engine* và *WordTable* cần tạo các dạng chuẩn.

Điểm khác là bột duyệt từ của *Doc* đơn giản trả về các từ. Nó không xác định các từ đó có là từ khóa hay không. Thiết kế này là mong muốn hơn thiết kế ở đó *Doc* biết về các từ khóa, vì nó thể hiện sự tách bạch không liên quan.

```
class Doc {  
    // Như trước bổ sung thêm chúng ta bây giờ biết Doc là không thay đổi và  
    // cung cấp bộ duyệt.  
    Iterator words()  
        // EFFECTS: Trả về một bộ sinh mà tạo ra các từ trong tài liệu như các  
        // xâu theo thứ tự mà chúng xuất hiện trong văn bản.  
}  
  
class Helpers {  
    // Cung cấp các thủ tục trợ giúp; hiện tại chỉ có canon.  
    String canon(String s) throws NotPossibleException  
        // EFFECTS: Nếu s là null throws NotPossibleException  
        // ngược lại trả về một xâu mà là dạng chuẩn của s.  
}
```

Hình 3.19 Các đặc tả cho *Doc* và *canon*

Đặc tả của *canon* và các đặc tả mở rộng cho *Doc* được cho trên Hình 3.19. Hình 3.20 chỉ ra lược đồ phụ thuộc module mở rộng.



Hình 3.20 Lược đồ phụ thuộc module mở rộng

3.6.3 Kết thúc thiết kế

Bây giờ chúng ta chỉ còn lại các trùu tượng *canon* và *Doc*. *canon* được cài đặt hiển nhiên và chúng ta không cần xem xét tiếp. *Doc* truyền đầu vào là xâu như cần thiết. Khi một *Doc* được tạo, hàm tạo tìm tiêu đề bằng phân tích cú pháp phần đầu tiên của xâu; nếu nó không tìm thấy tiêu đề, nó sẽ tung ra ngoại lệ *NotPossibleException*. Phần sau của phân tích được thực hiện bởi bộ duyệt các từ; tại mỗi bước lặp nó tìm từ tiếp theo và trả về nó. Khi kết thúc, nó đạt đến cuối tài liệu.

3.6.4 Tương tác giữa FP và UI

Thiết kế của chúng ta cho máy tìm kiếm sử dụng chỉ dạng đơn giản của tương tác giữa FP và UI; UI gọi các phương thức của FP (hoặc các phương thức của các kiểu hỗ trợ nào đó). Chúng ta sẽ xem xét một tương tác phức tạp hơn.

Thao tác addDocs có thể dùng thời gian lâu để hoàn thành, vì nó bao gồm tương tác với một trang ở xa, và thêm vào đó có rất nhiều tài liệu để lấy xử lý. Khi một thao tác xử lý lâu, người sử dụng nói chung muốn cam đoan một lần nữa là chương trình đang tiến triển. Sự cam đoan này có thể có dạng nói cho người sử dụng định kỳ cái gì đang tiến triển – chẳng hạn, mỗi khi phương thức *addDocs* của *Engine* nhận một tài liệu khác từ *getDocs*,

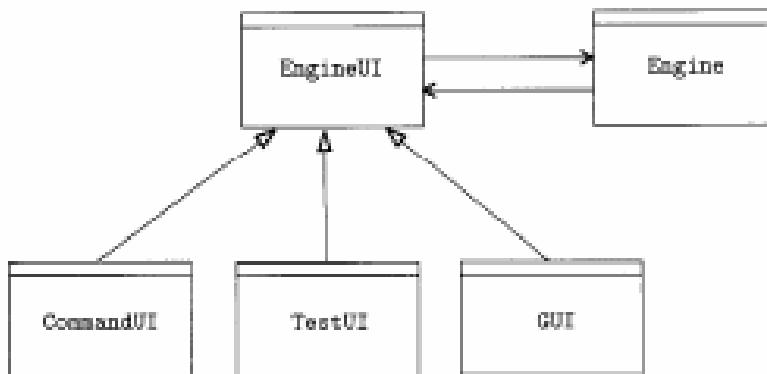
nó có thể gọi phương thức của UI để báo cáo về số đếm hiện tại; UI sau đó có thể hiển thị thông tin này cho người sử dụng. Phương thức UI có thể là

```
void docProgress (int n)  
// REQUIRES: Thao tác addDocs đang xử lý  
// EFFECTS: Thông báo cho người sử dụng là n tài liệu đã được lấy xử lý
```

Bây giờ chúng ta có cấu trúc (b) trong Hình 3.5.

Khi chúng ta có cấu trúc (b), FP sẽ cần truy cập đến đối tượng UI để gọi các phương thức của nó. Điều này có thể được thực hiện bằng cách truyền đối tượng UI như đối số cho hàm tạo của FP.

Khi UI và FP tương tác như trong cấu trúc (a) của Hình 3.5, FP là không phụ thuộc vào UI. Với cấu trúc (b) FP là phụ thuộc vào UI. Vì mục tiêu của chúng ta là có nhiều UI, chúng ta cần có cách để liên hệ chúng sao cho FP sẽ làm việc với bất kỳ giao diện nào trong số đó. Điều này dễ dàng được thực hiện sử dụng phân cấp sau: FP phụ thuộc vào UI trừu tượng, và các UI thực tế là các kiểu con của UI trừu tượng. Một lược đồ phụ thuộc module thể hiện tình huống này cho máy tìm kiếm được chỉ ra trên Hình 3.21 ở đó chúng ta thấy có ba giao diện UI thực tế: một cái thực hiện kiểm thử hồi qui cho FP, một cái cung cấp giao diện dòng lệnh, và một cái cung cấp một giao diện đồ họa. Trong lược đồ này, chúng ta bỏ qua cấu trúc của *Engine*, một cách để nghĩ về điều này là trong lược đồ này, đỉnh được gán nhãn *Engine* thay thế cho toàn bộ lược đồ phụ thuộc module mà đã chỉ ra trên Hình 3.20.



Hình 3.21 Phân cấp UI

Trên thực tế, chúng ta có thể cũng muốn có một phân cấp cho FP, sao cho UI có thể được sử dụng với nhiều phương án của FP.

Mở rộng máy tìm kiếm để thông báo cho người sử dụng về tiến triển khi đẩy tài liệu vào không chưa đáp ứng yêu cầu khác của người sử dụng: khi thao tác thực hiện rất lâu, người sử dụng muốn dừng nó. Khi điều này xảy ra, người sử dụng nói chung yêu cầu phản hồi nhanh chóng cho lệnh dừng. Thiết kế hiện tại của chúng ta cho phép dừng xảy ra mỗi khi *Engine* gọi phương thức *doProgress*, nhưng nó có thể không đủ nhanh, chẳng hạn, truyền thông là rất chậm hoặc trang chứa tài liệu không phản hồi đủ tốt.

Để cung cấp phản hồi nhanh hơn, tuy nhiên, yêu cầu sử dụng song song. Phương thức FP mà chạy lâu cần phải tách thành một luồng để là công việc dài sau đó trả về kết quả cho UI. UI sau đó có thể gọi một phương thức khác của FP, nếu người sử dụng chỉ ra rằng đây là thời điểm dừng công việc này. Chẳng hạn, trong máy tìm kiếm, thân của phương thức *addDocs* cần phải tách một luồng; luồng này sẽ thực hiện lấy và xử lý tài liệu, trong khi phương thức *addDocs* có thể trả về ngay tức thì. *Engine* có thể cần cung cấp phương thức khác mà UI có thể gọi để dùng lấy các tài liệu.

```
void stopFetch( )
```

// EFFECTS: Kết thúc việc đang lấy các tài liệu.

Cấu trúc này cho phép máy tìm kiếm tiếp tục nhận lệnh tiếp theo của người sử dụng ngay lập tức, mặc dù ngay cả khi việc lấy tài liệu đang được tiến hành.

3.7 Lược đồ phụ thuộc module với mô hình dữ liệu

Một mô hình dữ liệu định nghĩa trạng thái của một chương trình ở mức độ của đặc tả yêu cầu. Nó không nói gì về cấu trúc thành phần của chương trình. Các tập hợp trong mô hình không phải là các kiểu. Chúng ta không tạo ra thiết kế chương trình bằng việc đưa ra một trừu tượng dữ liệu cho mỗi tập hợp. Thay vào đó, chúng ta xem xét chương trình cần làm gì và đưa ra trừu tượng mà cho phép nó được hoàn thành theo cách kinh tế nhất. Tiếp theo, trong quá trình thiết kế, chúng ta không quá tập trung vào mô hình dữ liệu ngoại trừ một cách gián tiếp. Mô hình dữ liệu giúp giữ thiết kế tập trung vào những gì mong muốn, vì nó đóng vai trò quan trọng làm rõ các đặc tả yêu cầu. Kết quả cuối cùng của thiết kế là một cấu trúc mà hoàn toàn khác mô hình này. Nó được nhìn thấy cho máy tìm kiếm bằng cách so sánh Hình 2.7 và Hình 3.20.

Tuy nhiên, mô hình dữ liệu có thể được sử dụng để kiểm tra tính đúng đắn của thiết kế. Trước hết, mỗi tập trong mô hình dữ liệu cần phải được chỉ ra ở đâu đó trong chương trình, hoặc thiết kế không đúng. Chẳng hạn, ở đây có sự giải thích cái gì xảy ra với các tập hợp cho máy tìm kiếm:

Cur, Curmatch: chúng có ở trong UI

Doc: ở đây không có tập hợp tất cả tài liệu, nhưng mọi tài liệu có thể được truy cập từ WordTable và Title.

Title: TitleTable

Entry: được kiểm soát trong trừu tượng Doc

Word: phần của WordTable

NK: phần của WordTable

Key: được kiểm soát bởi Query

Match: được kiểm soát bởi Query

URL: như một phần của thẻ hiện của Engine

Num: int

Ngoài ra, chúng ta cần giải thích cái gì xảy ra với các quan hệ và các ràng buộc. Một số quan hệ biến mất; chẳng hạn điều gì xảy ra với quan hệ chỉ số (vì Doc cung cấp bộ duyệt hơn là truy cập bằng chỉ số) và quan hệ site (vì ở đây không cần xác định site mà cung cấp tài liệu). Các ràng buộc trên các quan hệ mà còn, cần phải được tuân thủ. Chẳng hạn, trừu tượng Query chịu trách nhiệm tin tưởng rằng ở đây không có sánh khi Key là rỗng. Là lý tưởng khi một ràng buộc kết thúc vẫn được tuân thủ bên trong một trừu tượng duy nhất, nhưng đây không phải trường hợp thông thường. Chẳng hạn, các ràng buộc TitleTable chưa bao nhập cho mỗi tài liệu chỉ nếu Engine nhập mỗi tài liệu vào bảng và bảng lưu mỗi tài liệu mà Engine nhập vào. Như vậy, là hữu ích xác định các module nào cần làm việc với nhau để bắt tuân theo mỗi ràng buộc. Ở đây có một số thông tin cho máy tìm kiếm:

Mỗi tài liệu có một tiêu đề: buộc tuân thủ bởi Engine và TitleTable

Mỗi từ trong mỗi tài liệu có trong Word: buộc tuân thủ bởi Doc, WordTable và Engine

NK là tập con của Word: buộc tuân thủ bởi WordTable

Key là tập con của Word – NK: buộc tuân thủ bởi Engine, Query và WordTable

Chính xác các tài liệu sánh ở trong Match: buộc tuân thủ bởi Query, WordTable, Doc và Engine

Key = { } → Match = { }: buộc tuân thủ bởi Query

matches được sắp xếp đúng: buộc tuân thủ bởi, Query, WordTable, Doc và Engine

Lưu ý rằng một số module thường không buộc tuân thủ ràng buộc, và lại máy Engine thường tham gia để buộc phải tuân thủ. Chẳng hạn, Engine tham gia giữ Key tách rời khỏi

NK vì nó kiểm tra một từ có là quan tâm hay không và tạo ra truy vấn nếu nó là quan tâm; WordTable cũng tham gia vì nó cần cung cấp câu trả lời đúng khi phương thức isInteresting được gọi. và Query cũng tham gia vì nó cần đưa ra câu trả lời cho từ cho trước như một đối số và không đổi với một từ khác nào đó. Lý do Engine tham gia thường xuyên như vậy vì nó điều tiết việc chạy toàn bộ chương trình; đây là kiểu module ở lớp trên cùng, và như kết quả, các module như vậy sẽ đóng vai trò buộc tuân thủ cho rất nhiều ràng buộc. Bảng 3.22 tóm tắt phần thảo luận trên.

- *Một lược đồ phụ thuộc module xác định các module của chương trình mà được sử dụng trong cài đặt. Mô hình dữ liệu định nghĩa các tập mà không tương ứng với các module của chương trình.*
- *Mỗi tập trong mô hình dữ liệu cần chỉ ra có ở đâu đó trong thiết kế. Một kiểm chứng về tính hoàn thiện của thiết kế là giải thích nó kiểm soát mỗi tập hợp như thế nào.*
- *Tiếp theo, một thiết kế cần buộc tuân thủ các ràng buộc trong mô hình. Một kiểm tra tốt về tính đúng đắn của thiết kế là giải thích nó được thực hiện hiện như thế nào.*

Bảng 3.22 Các mô hình dữ liệu và thiết kế

3.8 Đặc tả chương trình

Sau khi có lược đồ phụ thuộc giữa các module, chúng ta cần đưa ra đặc tả chương trình, để chỉ dẫn chương trình cần được viết như thế nào. Thông thường các thành viên đội dự án viết đặc tả cho mỗi module, sau đó đưa chúng cho các lập trình viên, những người sẽ viết code trong pha cài đặt. Đặc tả cần phải rõ ràng và dễ hiểu, nếu không sẽ ảnh hưởng đến tốc độ của các lập trình viên viết code các lệnh mơ hồ, chưa hoàn chỉnh. Khi viết đặc tả cụ thể có thể họ sẽ tìm được cách tốt hơn sắp xếp các module hoặc có thể phát hiện sự gắn kết bị bỏ sót hoặc không cần thiết.

Nói chung không có một cú pháp hình thức chung nào để viết đặc tả chương trình, mỗi công ty sử dụng định dạng của riêng mình. Hầu hết các đặc tả chương trình gồm bốn thành phần bao hàm các thông tin mà lập trình viên cần để viết code đúng đắn.

- **Thông tin chương trình:** như tên module, mục đích, bối cảnh dùng, người sử dụng, môi trường lập trình, thư viện hỗ trợ.
- **Các sự kiện:** liệt kê các sự kiện mà kích hoạt chức năng của chương trình. Một sự kiện là điều gì đó sẽ xảy ra hoặc được thực hiện. Đó hầu như là mọi thứ mà người sử dụng buộc sự kiện phải xảy ra. Các chương trình sự kiện điều khiển bao gồm các thủ tục mà được thực hiện phản hồi đến sự kiện được khởi tạo bởi người dùng, hệ thống hoặc code chương trình.

- **Các đầu vào và đầu ra:** Ở đây mô tả dữ liệu đầu vào và đầu ra, mà được xác định bởi quan hệ đầu vào và đầu ra; và quan hệ điều khiển. Các lập trình viên cần phải hiểu thông tin nào sẽ được truyền và tại sao các thông tin này được đưa vào các biến và các cấu trúc dữ liệu trong chương trình.
- **Giả code:** trình bày thuật toán cho chương trình dưới dạng giả code. Nêu các thủ tục con hỗ trợ được cài đặt. Giải thích đầu vào được đưa vào chương trình như thế nào, tương tác với cơ sở dữ liệu và các module khác ra sao và tạo đầu ra như thế nào. Có thể nêu rõ môi trường thực hiện module như thư viện sử dụng, các giao diện tương tác và lý giải module đáp ứng yêu cầu chức năng và phi chức năng như thế nào.

3.9 Đánh giá phương pháp thiết kế

Thiết kế Engine được thực hiện theo từng giai đoạn. Chúng ta chọn một mục tiêu mới ở mỗi giai đoạn và nghiên cứu cài đặt của nó bằng cách thực hiện các hành động chính. Trước hết, chúng ta tạo ra một số trừu tượng hỗ trợ mà là có ích trong giải quyết bài toán mục tiêu. Sau đó chúng ta xác định các tính chất của các hỗ trợ và mô tả các tính chất này trong đặc tả.

3.8.1 Tạo ra các hỗ trợ

Phản khó nhất của thiết kế chương trình, và chõ đòi hỏi sự sáng tạo lớn nhất, là việc tạo ra các hỗ trợ. Phương thức chủ yếu sử dụng ở đây là nghiên cứu cấu trúc bài toán và suy diễn từ đó ra cấu trúc chương trình. Nghiên cứu bài toán của chúng ta có dạng danh sách các nhiệm vụ cần được hoàn thành. Tuy nhiên, chúng ta không đơn giản sáng tạo cấu trúc chương trình mà thực hiện lần lượt các nhiệm vụ này. Thay vào đó, chúng ta sử dụng danh sách này như một cách tập trung chú ý vào cái cần phải thực hiện và sau đó tạo ra các trừu tượng để hoàn thành công việc cần thiết.

Các trừu tượng được đưa ra trong mỗi trường hợp để che giấu chi tiết. Chẳng hạn, Query che giấu chi tiết truy vấn được xử lý như thế nào, trong khi Doc che giấu chi tiết một tài liệu được thể hiện như thế nào. Mong muốn che giấu chi tiết vì hai lý do sau: kiểm soát độ phức tạp và tính thay đổi. Chúng ta kiểm soát độ phức tạp bằng cách trì hoãn các quyết định và như vậy hạn chế số việc mà cần giải quyết ở một mức cụ thể bất kỳ của thiết kế. Chúng ta hỗ trợ tính thay đổi vì các chi tiết được che giấu có thể thay đổi về sau mà không ảnh hưởng đến các trừu tượng khác.

Trong phát triển thiết kế, chúng ta được chỉ dẫn bởi một số yêu tố:

- Hiểu biết về các trừu tượng nào đã sẵn sàng có. Nó bao gồm các trừu tượng có trong ngôn ngữ lập trình và trong các thư viện chương trình có sẵn, cũng như những cái đã được xác định trong thiết kế này. Chẳng hạn, chúng ta biết rằng `java.util` cung cấp các bảng băm và rằng `quicksort` luôn sẵn sàng trong một thư viện.

- Kiến thức về các thuật toán và các cấu trúc dữ liệu đã tồn tại trước. Nó là cần thiết để biết về các phương pháp mà đã được phát triển. Chẳng hạn, người thiết kế cần làm quen với các kỹ thuật sắp xếp và tìm kiếm đã tồn tại và không phải sáng tạo chúng nữa.
- Kiến thức về các chương trình liên quan và cấu trúc của chúng. Như chúng ta đi sâu hơn về thiết kế, kiến thức này như trở thành ngày càng hữu ích hơn, vì ngay cả các ứng dụng rất khác nhau cũng có những nhiệm vụ con tương tự.

Chẳng hạn, quyết định xử lý các từ trong một tài liệu của chúng ta là chúng ta bổ sung chúng vào bảng *WordTable* được dựa trên kiến thức của chúng ta rằng các thuật toán chạy qua một lần thường là tốt hơn về không gian và thời gian so với các thuật toán chạy qua hai lần. Cũng như vậy, chúng ta dựa trên sự tồn tại các kiểu của Java, bao gồm cả các kiểu trong các gói như *java.io*, và lựa chọn tường minh giữa chúng và các trùu tượng dữ liệu; chẳng hạn, chúng ta có thể đã định nghĩa một trùu tượng cho tiêu đề của một tài liệu, nhưng nó có vẻ đã đủ là dùng xâu trong trường hợp này.

Vai trò các kiểu dữ liệu

Các kiểu thường có tác động lớn đến dạng của thiết kế, với các thủ tục và bộ duyệt cũng như phương thức, nhưng các thủ tục và bộ duyệt độc lập là quan trọng. Chúng ta tìm kiếm các kiểu trong bốn lĩnh vực: đầu vào, đầu ra, cấu trúc dữ liệu bên trong và các mục dữ liệu riêng. Hầu hết các trùu tượng dữ liệu trong Engine che giấu các cấu trúc dữ liệu bên trong. (Thiết kế Engine là có một chút bất thường trong việc sử dụng lặp của cùng một cấu trúc dữ liệu – bảng băm; thông thường hơn, một chương trình sẽ sử dụng các cấu trúc dữ liệu khác nhau). Tuy nhiên, *Doc* là một trùu tượng của một mục dữ liệu, và trùu tượng *WordTable* từ chi tiết đầu vào (các trúc file mà định nghĩa các từ không quan tâm). Chúng ta có thể có một trùu tượng đầu ra, nếu ở đây có cách cho người sử dụng yêu cầu máy tìm kiếm viết thông tin về các tài liệu sánh vào một file; trong trường hợp này, *InfoStream*, ở đó thông tin về toàn bộ một tài liệu sánh sẽ được bổ sung một lần, có thể là có ích.

Bao đóng đầu vào và đầu ra cho phép chúng ta tiếp nhận đầu vào hoặc tạo đầu ra dưới dạng số lượng tổng quát. Đầu vào và đầu ra là thường được kiểm soát với các thủ tục hoặc bộ duyệt, vì chúng ta thông thường chỉ đọc hoặc viết mục tiếp theo. Bộ duyệt *getDocs* là một ví dụ. Nó có thể tạo bộ đệm tưởng tượng – chẳng hạn, lấy các tài liệu theo gói hoặc lấy tài liệu tiếp theo trong khi chúng ta đang sử dụng cái hiện tại. Vì, nó là một trùu tượng, các chi tiết này là được che giấu, chúng ta không cần liên quan đến chúng.

3.8.2 Đặc tả các bộ trợ

Khi đặc tả bộ trợ, chúng ta liên quan chỉ với các khái niệm này; các chi tiết của khái niệm đó thường được định nghĩa sau. Trong bước thứ hai của thiết kế, chúng ta khẳng định các chi tiết này và viết tài liệu về chúng bằng việc đặc tả. Theo hoạt động này, chúng ta nhìn chung khám phá các trường hợp mà đã bị bỏ qua, đặc biệt các trường hợp lỗi và các trường

hợp hiển nhiên (chẳng hạn, đầu vào rỗng). Chúng ta có thể khám phá các đối số hoặc kết quả bị bỏ sót, hoặc tìm các đối số và kết quả thuộc loại kiểu sai. Trong trường hợp các kiểu dữ liệu, chúng ta có thể khám phá các thao tác bị sót. Ngoài ra, nếu một trùu tượng là quá phức tạp, điều này có thể là tự nhiên, khi chúng ta thử viết đặc tả. Một đặc tả phức tạp hiển nhiên dẫn tới câu hỏi làm thế nào để các vật trở nên đơn giản. Thông thường nhận được kết quả là một trùu tượng đơn giản hơn và tốt hơn.

Chúng ta sẽ sử dụng các phác thảo cài đặt như hỗ trợ định dạng và đặc tả các trùu tượng hỗ trợ và viết tài liệu các quyết định cài đặt của chúng ta. Các phác thảo này chưa mô tả các bài toán con cần hoàn thành. Sau đó đối với mỗi nhiệm vụ con chúng ta cần định danh thủ tục hoặc bộ duyệt, hoặc thao tác của một kiểu, mà được sử dụng và các đối số là gì. Không giống như danh sách các nhiệm vụ được sử dụng để bắt đầu thiết kế, thứ tự ở đó các nhiệm vụ con được thực hiện trong phác thảo là rất quan trọng. Các phác thảo này đáng giá để làm chỉ khi có cái gì đó phức tạp đang tiếp diễn; chúng ta sẽ sử dụng chúng chỉ cho trùu tượng *Query*.

Sau khi chúng ta đặc tả mọi bộ trợ chính xác, chúng ta viết các quyết định thiết kế của chúng ta trong sổ tay bằng việc mở rộng lược đồ phụ thuộc module và bằng việc bổ sung và mở rộng các định nghĩa bộ trợ. Mặc dù chúng ta đã chỉ ra chỉ các phần mới của đặc tả trên các hình vẽ, toàn bộ đặc tả sẽ xuất hiện như một mục trong sổ tay. Chúng ta cũng sẽ bao gồm một phác thảo cài đặt của mỗi mục tiêu nếu nó được thực hiện; như đã nêu, chúng ta sẽ ghi lại các quyết định về các trúc dữ liệu nào được sử dụng trong thể hiện kiểu và chúng ta có thể cung cấp một phác thảo của thể hiện.

Một vấn đề mà chúng ta chưa bàn tới là làm thế nào một thiết kế của một cài đặt trùu tượng cần được thực hiện. Ở mỗi bước, một nghiên cứu cần được thông suốt vừa đủ để mọi bộ trợ được định danh và đặc tả chính xác. Điều này không cần thiết đi xa hơn nữa. Chẳng hạn, chúng ta đã xem xét kỹ cài đặt *WordTable*, vì nó đã rõ ràng cần được làm gì. Trái ngược lại, một thiết kế của *Query* được thực hiện chi tiết, vì chúng ta cần hiểu cái gì được làm và các bộ trợ cần được định nghĩa như thế nào. Tuy nhiên, nói chung, chúng ta mong đợi hơn thiết kế là cần thiết xuyên suốt cài đặt, bao gồm có thể cả nhận biết sự cần thiết các phương thức riêng trong các trùu tượng dữ liệu.

3.8.3 Tiếp tục thiết kế

Ngay sau khi một mục tiêu đã được nghiên cứu xong, bước tiếp theo là lựa chọn một mục tiêu mới. Trước hết, mọi ứng cử viên đã được xác định, các ứng cử viên này là các trùu tượng mà cài đặt của chúng chưa được nghiên cứu nhưng việc chúng sử dụng các trùu tượng đã được phân tích. Trong khi đã có các chỉ dẫn về việc thực hiện lựa chọn này, ở đây vẫn chưa có qui tắc. Chẳng hạn, không có yêu cầu nào mà mọi trùu tượng ở một mức độ thiết kế cần được nghiên cứu xong trước khi đi đến mức thấp hơn. Thay vào đó, ý niệm chung cần được sử dụng, với mục tiêu kết thúc toàn bộ thiết kế càng nhanh càng tốt. Đó là

cách tại sao chúng ta xem xét các trùu tượng có vấn đề càng sớm càng tốt và ngay cả nghiên cứu chúng trước khi chúng được hoàn thiện.

Nếu trong khi nghiên cứu cài đặt mục tiêu nào đó, chúng ta khám phá ra một lỗi trong chính trùu tượng đó (chẳng hạn có một thay đổi trong đặc tả của nó), chúng ta cần sửa lỗi đó trước khi tiếp tục với mục tiêu hiện tại. Chúng ta sử dụng các cung trong lược đồ phụ thuộc module để khám phá mọi cài đặt bị tác động bởi lỗi này. Sau đó chúng ta hiệu chỉnh thiết kế cho các cài đặt đó. Trong tiến trình này, chúng ta có thể khám phá ra nhiều lỗi hơn mà cũng cần phải hiệu chỉnh. Chúng ta nhìn thấy một số ví dụ về lỗi như vậy trong thiết kế của chúng ta, nhưng mọi cái trong số đó đều rất dễ dàng kiểm soát. Trong thiết kế trên thực tế, chúng ta thường như tìm được các vấn đề quan trọng hơn mà có thể dẫn chúng ta quay trở lại một số trùu tượng trước khi vấn đề đó có thể được chỉnh sửa. Trên thực tế, một số lỗi thiết kế có thể không được tìm thấy cho đến khi cài đặt đang được thực hiện. Mô hình dữ liệu có thể được sử dụng để kiểm tra các lỗi thiết kế như đã bàn trước đây.

Khi không còn ứng cử viên nào, thì thông thường thiết kế được kết thúc. Tuy nhiên ở đây có một trường hợp đặc biệt. Khi hai hay nhiều hơn các trùu tượng là đệ qui qua lại, thì không cái nào có thể là một ứng cử viên, vì mỗi cái được sử dụng trong trùu tượng kia mà chưa được nghiên cứu. Khi đụng chạm đến các trùu tượng đệ qui qua lại, chúng ta cần xử lý cẩn thận. Một cái cần được chọn như một ứng cử viên để nghiên cứu trước. Vì ứng cử viên này được sử dụng bởi một trùu tượng khác mà chưa được nghiên cứu, chúng ta có thể khám phá về sau là hành vi của nó không phải là cái cần thiết. Vấn đề này là một chỉ dẫn khác về tại sao đệ qui qua lại cần được xem xét kỹ càng.

3.8.4 Sổ tay thiết kế

Kết quả của một thiết kế kết thúc là một sổ tay thiết kế chứa lược đồ phụ thuộc module hoàn chỉnh và các phần cho mỗi trùu tượng. Điều quan trọng là sổ tay thiết kế này chưa chỉ các lý giải cho chúng. Lý tưởng là, phần này của tài liệu sẽ giải thích cả các vấn đề nào được giải quyết bởi một cấu trúc cụ thể và các vấn đề nào đã tránh được mà có thể được nảy sinh bởi các cấu trúc khác đã được nghiên cứu trong thiết kế.

Vì tài liệu như vậy là khó và tốn kém thời gian để hoàn thành, nó thường được bỏ qua. Quan trọng là trong các giai đoạn về sau của phát triển chương trình, bằng cách nào đó, khi nào đó một tình huống nảy sinh mà một quyết định thiết kế cần được xem xét lại hoặc thay đổi. Quyết định mới cần là tốt nhất có thể làm được bởi một người hiểu biết đầy đủ thiết kế đó. Tài liệu làm cho thông tin cần thiết này sẵn sàng, cả cho người khác nhưng và ngay cả với những người thiết kế ban đầu, mà sẽ bị quên theo thời gian.

3.8.5 Thiết kế trên xuống

Quá trình thiết kế của chúng ta là trên xuống, đó là chúng ta luôn suy luận xuất phát từ cái mong muốn đến làm sao để đạt được điều đó. Theo cách này, chúng ta luôn giữ chặt mục

đích trong xem xét và thoái mái sử dụng cảm giác của chúng ta về chương trình để chỉ dẫn chúng ta đến một giải pháp. Thông qua kinh nghiệm trong viết chương trình, các lập trình viên phát triển cảm giác về cái gì là cài đặt được với tính hiệu quả như thế nào. Chúng cũng dẫn tới việc biết các cấu trúc chương trình như thế nào là phù hợp cho các bài toán khác nhau. Kiến thức này có thể được sử dụng cho tác động hiệu quả tốt trong thiết kế trên xuống.

Một cách khác so với thiết kế trên xuống là thiết kế dưới lên, mà bắt đầu từ những gì đã được biết là sẽ cài đặt được và bằng cách nào đó tiếp tục từ đó để đạt được cái mà mong muốn. Thiết kế dưới lên không thực tế là quá trình biện hộ được cho bất cứ cái gì trừ những chương trình nhỏ nhất. Đối với các chương trình lớn lỗ hổng giữa cái gì sẵn sàng và cái gì mong muốn là lớn và cần được bắc cầu bởi việc đưa vào nhiều trùu tượng. Lỗ hổng này là dễ dàng bắc cầu trên xuống vì chúng ta có thể tập trung vào việc cái gì được hiểu ít nhất (chương trình mà được mong muốn) và sử dụng cái gì mà chúng ta biết là sẽ hỗ trợ. Với thiết kế trên xuống, cảm giác của chúng ta cái gì cài đặt được nó sẽ cho chúng ta biết có thực hiện bước tiến không: các bộ trợ cần được dễ dàng cài đặt hơn so với đích. Với thiết kế dưới lên, nó là khó hơn nhiều đo sự tiến bộ, vì nó khó được đánh giá làm thế nào biết được chúng ta gần với cái mong muốn.

Trên thực tế, chúng ta hướng tới quay lui và tiến giữa thiết kế trên xuống và dưới lên. Chẳng hạn, chúng ta có thể nghiên cứu cài đặt các trùu tượng chính hoặc trùu tượng còn nghi ngờ ngay cả khi mọi sử dụng là chưa được hiểu kỹ. Tuy nhiên, quan trọng là thiết kế đó được dẫn dắt từ trên và rằng chúng ta thực tế tránh cài đặt bất cứ trùu tượng nào cho đến khi thiết kế của họ và các bộ trợ của chúng là hoàn chỉnh; thực hiện cài đặt trước là thực sự lãng phí, vì các cơ hội của mọi chi tiết sẽ đúng là rất nhỏ.

Tóm tắt

Chương này chúng ta đã bàn về thiết kế chương trình. Thiết kế được tiến triển bằng việc phân rã thành phần dựa trên ghi nhận các trùu tượng có ích. Chúng ta đã bàn làm sao phân rã này xảy ra và thể hiện quá trình thiết kế với một ví dụ. Chúng ta cũng đã bàn về một phương pháp viết thiết kế vào một số tay.

Ví dụ đã sử dụng là một máy tìm kiếm đơn giản, và chương trình kết quả là nhỏ. Thêm vào đó, thể hiện quá trình thiết kế là không thực tế: chúng ta đã có rất ít lỗi khi chúng ta thiết kế, và các lỗi này có tác động rất ít lên toàn bộ thiết kế. Trong thực tế, bất cứ thiết kế nào, ngay cả chương trình đơn giản, cũng đòi hỏi xử lý nhiều lần và nhiều lỗi sẽ xuất hiện và được hiệu chỉnh khi nó tiến triển. Tuy nhiên, các phương pháp cơ bản mà chúng ta áp dụng ở đây, cũng đúng đối với các chương trình lớn hơn nhiều.

Chúng ta không tuyên bố rằng thiết kế máy tìm kiếm này là tốt nhất. Trên thực tế, mục đích của quá trình thiết kế không bao giờ là thiết kế tốt nhất. Thay vào đó, nó là một thiết kế

“phù hợp”, một cái mà thỏa mãn mọi yêu cầu và các mục đích thiết kế và có một cấu trúc tốt chấp nhận được.

Chương này tiến hành hoàn toàn về thiết kế, không có cài đặt nào xảy ra cho đến khi thiết kế được hoàn thành. Thông thường mong muốn cài đặt sớm hơn. Nếu một phần của thiết kế được hoàn thiện, các module đó có thể được cài đặt trong khi phần còn lại của thiết kế đang được thực hiện. Chẳng hạn, chúng ta có thể cài đặt *TitleTable* trong khi chúng ta đang làm việc về thiết kế xử lý truy vấn. Bắt đầu cài đặt sớm là mong muốn vì nó có thể dẫn tới hoàn thành sớm chương trình hoặc cho ra sớm sản phẩm mà cung cấp một số đặc trưng. Tuy nhiên, trước khi thực hiện bắt cứ cài đặt nào, quan trọng là đánh giá thiết kế xem nó có đúng không – cái mà sẽ dẫn tới một chương trình đáp ứng mọi yêu cầu. Các kỹ thuật xác định sẽ bàn đến trong chương sau.

CHƯƠNG 4: ĐÁNH GIÁ THIẾT KẾ

Trong chương này chúng ta sẽ bàn ngắn gọn về hai suy xét được nảy sinh giữa hoàn thiện một thiết kế và bắt đầu cài đặt – mà được gọi là đánh giá thiết kế hay phản biện thiết kế và lựa chọn một chiến lược phát triển chương trình.

4.1 Phản biện thiết kế

Trong khi thiết kế một chương trình lớn, rất đáng để lùi một bước định kỳ và thử ước lượng toàn diện về thiết kế đã có. Quá trình này được gọi là xem xét lại thiết kế. Xem xét lại thiết kế cần luôn được thực hiện bởi một đội ngũ gồm một số người đã tham gia thiết kế và những người chưa tham gia. Những người chưa tham gia thiết kế cần được làm quen với cả yêu cầu chương trình và công nghệ mà sẽ được sử dụng trong cài đặt thiết kế. Họ cũng được làm quen với bản thân thiết kế.

Quan trọng là cả hai, đội ngũ thiết kế và những người xem xét bên ngoài đều hiểu rằng xem xét lại thiết kế không phải là khám phá một thiết kế hoàn chỉnh, nhưng là khám phá xem thiết kế hiện tại có phù hợp không – luôn làm công việc với hiệu năng và giá chấp nhận được. Trong khi đội ngũ thiết kế không tránh được sẽ tự tìm cách bảo vệ các quyết định thiết kế của họ trước những người xem xét bên ngoài, những người này sẽ không xem xét điều này là mục đích chính của mình. Thêm vào đó, cả hai người phản biện và người thiết kế cần giữ trong đầu là mục đích của xem xét chỉ là tìm các lỗi, chứ không phải hiệu chỉnh chúng. Các lỗi cần được ghi lại trong mục lỗi, và sau đó phản biện cần được tiếp tục (trừ khi rất nhiều lỗi được tìm ra rằng tiếp tục là không hiệu quả nữa).

Sẽ là có ích cho người thiết kế thể hiện không chỉ thiết kế mà còn các phương án mà đã xem xét và loại bỏ. Nó sẽ cho những người phản biện bên ngoài một ngữ cảnh để đánh giá thiết kế được chọn. Nó cũng có thể giúp những người phản biện tìm ra thiếu sót trong thiết kế. Vấn đề chung là thất bại nếu áp dụng tiêu chuẩn thiết kế một cách đồng nhất. Giải thích một phương án mà đã bác bỏ vì nó thất bại đáp ứng tiêu chuẩn nào đó có thể nhắc những người phản biện lưu ý rằng một số phần khác của thiết kế cũng thất bại đáp ứng cùng tiêu chuẩn này.

Sau đây là ba vấn đề cần quan tâm trong đánh giá một thiết kế:

1. Mọi cài đặt của thiết kế thể hiện chức năng mong muốn không? Nếu có, chương trình có đúng đắn không?
2. Có các cài đặt của thiết kế mà là hiệu năng chấp nhận được không?
3. Thiết kế có mô tả cấu trúc chương trình mà sẽ làm cho cài đặt, kiểm thử và bảo trì khá dễ dàng được thực hiện không? Cũng như vậy, khó như thế nào để nâng cấp thiết kế nhằm đáp ứng các thay đổi trong tương lai, đặc biệt những cái mà đã được định danh trong pha phân tích yêu cầu?

4.1.1 Tính đúng đắn và hiệu năng

Kiểm thử và kiểm chứng phi hình thức là hai cách tiếp cận để làm tăng sự tin tưởng của chúng ta rằng chương trình sẽ có hành vi như mong muốn. Không may mắn, không cách tiếp cận nào đó có thể áp dụng cho thiết kế. Vì thiết kế không thể chạy, kiểm thử không áp dụng được. Nếu thiết kế được thể hiện hoàn toàn trong ngôn ngữ hình thức, kiểm chứng nào đó có thể được thực hiện. Tuy nhiên, đặc tả hình thức của thiết kế các chương trình lớn hiện tại là rất khó.

Trong khi hiện nay không có kỹ thuật rộng lớn hoàn chỉnh nào để phản biện một thiết kế, quan trọng là phản biện thiết kế được thực hiện có hệ thống. Chúng được kiểm tra cả hai tính chất thiết kế cục bộ và tổng thể. Tính chất cục bộ có thể được kiểm tra bằng nghiên cứu đặc tả của các module riêng biệt, các tính chất tổng thể bằng cách nghiên cứu làm thế nào các module gắn kết với nhau.

Hai tính chất cục bộ quan trọng là tính kiên định và tính đầy đủ. Một tính chất cục bộ quan trọng khác là hiệu năng. Bước đầu tiên là đánh giá hiệu năng tổng thể của hệ thống là xây dựng cho mỗi trùu tượng một biểu thức liên quan đến thời gian chạy và giả thiết lưu trữ các đối số của nó. Chính xác như thế nào điều này có thể tính được phụ thuộc vào tính đầy đủ của thiết kế. Xem xét thủ tục *sort* sau:

```
void sort (Vector v) throws ClassCastException  
    // MODIFIES: v  
    // EFFECTS: Nếu v là khác null, sort nó vào danh sách tăng sử dụng phương  
    // thức CompareTo, nếu có một số phần tử của v là null hoặc so sánh  
    // được thì throws ClassCastException
```

Thiết kế trên không đặc tả bất cứ ràng buộc hiệu năng nào cho *sort*, ít liên quan đến hiệu năng cài đặt của nó hoặc về hiệu năng của các trùu tượng được cài đặt sử dụng *sort*. Vấn đề là cài đặt này của *sort* có phạm vi rộng về hiệu năng. Xem xét kỹ hơn có thể nói về hiệu năng cài đặt, nếu thiết kế bao gồm tiêu chí sau:

```
// worst case time = n*log (n) so sánh, trong đó n là kích thước của vector
```

Và hơn nữa, nếu thiết kế khẳng định

```
// worst case time = n*log (n) so sánh, trong đó n là kích thước của vector  
// Bộ nhớ chính tạm thời tối đa là một hằng số nhỏ
```

Một chức năng quan trọng của phản biện thiết kế là khám phá ra các chỗ ở đó thiết kế cần là chuyên biệt hơn về cái gì nó yêu cầu cho cài đặt.

Sau khi mỗi module được đánh giá riêng biệt, chúng ta sẽ đánh giá thiết kế một cách tổng thể. Một cách tốt để bắt đầu là bàn bạc về mối liên hệ của lược đồ phụ thuộc module với mô hình dữ liệu. Nó cần phải là có khả năng giải thích cài đặt kiểm soát mỗi tập hợp và bài trì mỗi ràng buộc như thế nào.

Bước tiếp theo là giám sát các cách qua thiết kế mà tương ứng với các thao tác khác nhau. Chúng ta chọn một số dữ liệu kiểm thử và sau đó giám sát làm thế nào cả kiểm soát và dữ liệu có thể đi qua một cài đặt dựa trên thiết kế. Quá trình giám sát này đôi khi được gọi là *dạo qua* (*walk-through*). Tuy nhiên, vì kiểm chứng một thiết kế là rất tốn công, chúng ta cần rất có chọn lọc khi chọn dữ liệu kiểm thử.

Vì vấn đề giám sát thiết kế là để thuyết phục chúng ta rằng mọi cài đặt trong thiết kế sẽ có tính năng mong muốn, thành công của phương pháp này là liên quan đến việc hoàn thành các trường hợp kiểm thử. Chúng ta sử dụng các trường hợp kiểm thử để tiến hành quá trình kiểm chứng không hình thức. Trong khi phản biện thiết kế, quan trọng là bàn luận tính hoàn thiện của quá trình đó – mà là, tranh luận mọi trường hợp xem xét. Cả hai các trường hợp thông thường và ngoại lệ đều cần được xem xét.

Lựa chọn các trường hợp kiểm thử cho phản biện thiết kế được đơn giản hóa bởi thực tế rằng dữ liệu đó có thể là tượng trưng. Chúng ta cần định danh các tính chất mà dữ liệu kiểm thử cần có; chúng ta không cần tạo ra dữ liệu với các tính chất đó.

Như một ví dụ, xét chương trình máy tìm kiếm được thiết kế trong chương trước. Chúng ta bắt đầu với lời gọi hàm tạo *Engine*. Nếu có lỗi xảy ra khi tạo “*WordTable*” (vì rằng file các từ không quan tâm được tạo lỗi), chương trình sẽ dừng. Ngược lại, một bảng sẽ được khởi tạo, và chúng ta có thể tiếp tục với các lệnh người sử dụng. Tại điểm này, không có tài liệu nào được gom; và do đó việc tử tìm tài liệu sử dụng tiêu đề của nó sẽ lỗi, và truy vấn cũng sẽ lỗi (nếu đầu vào là một từ không quan tâm hoặc không phải là từ) hoặc không cho bản sánh nào.

Bây giờ chúng ta giả thiết người sử dụng yêu cầu lấy xử lý một số tài liệu. Ở đây chúng ta có thể định nghĩa một số dữ liệu tượng trưng: việc lấy tạo ra ba tài liệu:

- d1 chứa w1 6 lần và w2 12 lần và có tiêu đề t1
- d2 chứa w1 10 lần và có tiêu đề t2
- d3 chứa w2 4 lần và có tiêu đề t3

Mọi từ này đều là các từ quan tâm; ngoài ra, các tài liệu chứa một số từ không quan tâm.

Chúng ta sử dụng dữ liệu này *dạo qua* thiết kế máy *Engine*. Việc *dạo qua* trên thực tế là mô phỏng bằng tay của thiết kế. Cái chính chúng ta mong muốn kiểm tra là luồng thông tin đi qua chương trình. Ở đây là làm thế nào chúng ta có thể bắt đầu *dạo qua* dựa trên dữ liệu trước đó:

- 1 Xử lý lệnh lấy tài liệu làm cho ba *Docs* được tạo ra. Sau đó *Docs* được bổ sung cho các bảng tiêu đề *TitleTable* và bảng từ *WordTable*. Khi một *Doc* được truyền cho phương thức *addDoc* của *WordTable*, các từ của nó được bổ sung cho bảng này nếu chúng là các từ quan tâm. Công việc này được thực thi hiệu quả, vì các bảng *TitleTable* và *WordTable* là các bảng băm, và các tài liệu được xử lý từng bước.
- 2 Giả sử người sử dụng thực hiện truy vấn cho *w1*. Từ này được chuẩn hóa, xem có là từ quan tâm không, và truyền cho *Query*. *Query* tìm kiếm từ này trong bảng *WordTable* và tìm được nó chứa trong *d1* và *d2*. Nó sẽ sắp xếp theo thứ *d2, d1*. Bây giờ người sử dụng có thể kiểm tra các tài liệu thông qua UI. Việc tìm kiếm từ trong bảng *WordTable* là nhanh, vì nó là bảng băm, và sắp xếp cũng là hiệu quả.
- 3 Tiếp theo người sử dụng bổ sung *w2* vào truy vấn. Nó được chuẩn hóa, xem có là từ quan tâm không, và truyền cho phương thức *addKey* của *Query*. Phương thức này tìm kiếm từ đó trong *WordTable* và tìm được nó sánh với *d1* và *d3*. Nó kết hợp thông tin này với kết quả truy vấn trước để xác định chỉ có *d1* có cả hai từ.
- 4 Tiếp theo người sử dụng tìm kiếm theo tiêu đề *t*. Nó được tìm kiếm trong bảng *TitleTable*, mà trả về bản sánh phù hợp hoặc tung ra một ngoại lệ, nếu *t* không là một tiêu đề của một tài liệu đang có.

Dạo qua tiếp tục theo cách như vậy cho đến khi hành vi của toàn bộ chương trình được khám phá. Trong quá trình này, chúng ta đánh giá hiệu năng của mỗi module, sao cho chúng ta có thể xây dựng đánh giá hiệu năng trong trường hợp xấu nhất và trung bình cho toàn bộ chương trình.

Dạo qua là quá trình tốn kém và không chính xác. Kinh nghiệm chỉ ra rằng người thiết kế hiếm khi có khả năng kiểm tra thiết kế của chính họ một cách đầy đủ. Quá trình này làm việc tốt nhất khi nó được thực hiện bởi một đội ngũ những người kể cả người thiết kế, nhưng số người thiết kế không quá bán.

Giám sát qua toàn bộ thiết kế với tập nhỏ đầu vào giúp chúng ta khám phá các lỗi thô theo cách các trùu tượng mà tạo nên thiết kế gắn kết với nhau. Một bước tiếp theo tốt là làm việc từ dưới lên thông qua lược đồ phụ thuộc module, cô lập các hệ thống con mà có thể đánh giá có ý nghĩa một cách độc lập với ngữ cảnh ở đó chúng được sử dụng. Vì các hệ thống con này thường như là nhỏ hơn đáng kể so với toàn bộ hệ thống, chúng ta có thể giám sát nhiều hơn các tập hợp của dữ liệu kiểm thử qua chúng. Chẳng hạn, nếu chuẩn hóa từ có thể bỏ ra ngoài thiết kế, lỗi có thể được phát hiện trong khi phản biện *WordTable*, vì tại điểm đó chúng ta có thể xem xét chi tiết hơn đến các từ trong tài liệu.

Tính chất thay đổi của một thiết kế cần được đề cập một cách tường minh trong khi phản biện. Việc tranh luận làm như thế nào một thiết kế cần được thay đổi để thích nghi mỗi

thay đổi mong muốn cần phải được xem xét. Một độ đo có về hợp lý về độ tốt của một thiết kế thích nghi với các thay đổi là có bao nhiêu trùu tượng cần phải cài đặt lại hoặc đặc tả lại trong mỗi trường hợp. Tình huống tốt nhất là cái mà ở đó chỉ có một trùu tượng cần phải cài đặt lại.

Chẳng hạn, giả sử máy tìm kiếm được sửa đổi không lưu trữ văn bản các tài liệu; thay vào đó khi một người sử dụng muốn kiểm tra một tài liệu, tài liệu này cần phải lấy lại từ trang của nó. Điều này yêu cầu thiết kế của chúng ta cần phải thay đổi theo một số cách. Trước hết, chúng ta có thể dường như tải lại tài liệu sử dụng URL của trang, nhưng điều này có nghĩa là chúng ta cần nhận được URL đó và các tài liệu liên quan đến nó (chẳng hạn bằng cách lưu nó trong *Doc*). Một cách để làm như vậy là thay thế bộ duyệt *getDocs* bằng bộ duyệt mà tạo ra các URL của các tài liệu và sau đó sử dụng một thủ tục để lấy một tài liệu cho trước URL; thủ tục này cũng có thể được sử dụng để lấy lại các tài liệu. Chúng ta cũng cần tách việc tạo ra các từ trong thân của một tài liệu ra khỏi các đối tượng được lưu trong các bảng *TitleTable* và *WordTable* sao cho việc lưu trữ này với thân có thể được xóa bỏ một lần khi các từ của một tài liệu đã được bổ sung vào *WordTable*.

Một cách để kiểm soát thay đổi này là đưa ra một trùu tượng *FullDoc*, với một bộ duyệt từ; và cũng là một phương thức mà trả về một *Doc*. *Engine* tạo ra một *FullDoc* và truyền nó cho phương thức *addDoc* của *TitleTable* và *WordTable*, nhưng cả hai bảng ánh xạ vào *Docs* chứ không phải *FullDocs*. Do đó, *FullDoc* có thể là nơi lưu giữ, một khi thông tin về nó đã được bổ sung vào trong hai bảng. *Doc* lưu giữ chỉ URL của tài liệu và có thể cái khác chứa thêm thông tin (chẳng hạn tiêu đề), nhưng không phải toàn bộ nội dung. Như vậy, thay đổi này yêu cầu rất ít thay đổi, và chúng là trực tiếp chấp nhận được.

Một lần *đao qua* buộc chúng ta xem xét thiết kế từ một góc nhìn khác với cái mà đã đặc trưng quá trình thiết kế. Trong khi thiết kế, chúng ta đã tập trung vào xác định các trùu tượng và đặc tả giao diện của chúng. Các trùu tượng này được诞生 từ việc xem xét bước nào được thực thi, nhưng chú ý của chúng ta đã được tập trung vào các phần của chương trình một cách riêng biệt. Nay giờ chúng ta quay trở lại các bước được thực hiện bởi toàn bộ chương trình mà nó sử dụng các trùu tượng, và bài tập này hướng chúng ta đến các câu hỏi các trùu tượng có được kết hợp với nhau để giải bài toán gốc không. Bảng 4.1 tổng kết phần này của phản biện thiết kế

- *Giải thích làm nhu nào một thiết kế bao quát các tập hợp và các ràng buộc của mô hình dữ liệu.*
- *Thực hiện một lần *đao qua* chương trình trên dữ liệu kiểm thử tượng trưng để chỉ ra các đối số của nó là đủ và hiệu năng của chúng có thể là hợp lý.*
- *Bàn luận về thiết kế có thích nghi với các thay đổi tiềm năng không.*

Bảng 4.1 Phản biện thiết kế: Đánh giá chức năng

4.1.2 Cấu trúc

Vấn đề cấu trúc quan trọng nhất được đề cập trong đánh giá một thiết kế là tính phù hợp của các danh giới giữa các module. Có hai câu hỏi chính thường được hỏi:

1. Chúng ta đã bị lỗi khi xác định một trùu tượng mà có thể dẫn đến việc phân tách thành phần tốt hơn chưa?
2. Chúng ta có đã nhóm các thứ mà thực tế không thuộc vào một cùng một module không?

Chúng ta có thể không cung cấp công thức nào cho các câu hỏi này. Cái mà chúng ta có thể cung cấp là một danh sách các dấu hiệu mà xảy ra khi một chương trình đã được phân tách module một cách không tốt. Chúng ta sẽ tập trung vào các dấu hiệu cục bộ - cái mà, các vấn đề có thể phát hiện bằng cách xem xét một module đơn hoặc tại giao diện giữa hai module.

Sự kết dính của các thủ tục

Mỗi thủ tục trong một thiết kế cần thể hiện một trùu tượng gắn kết duy nhất. Sự kết dính của một trùu tượng có thể được kiểm tra bằng cách xem xét đặc tả của nó. Một thủ tục cần thực hiện một thao tác trùu tượng duy nhất trên các đối số của nó. Bàn luận của chúng ta cũng được áp dụng cho các bộ duyệt. Một bộ duyệt ánh xạ đầu vào của nó vào dãy các đồ vật; ánh xạ này cần là một thao tác trùu tượng duy nhất.

Một số thủ tục không có độ kết dính bên ngoài rõ ràng. Chúng được giữ cùng nhau không bởi cái gì ngoài việc được đặt bởi một cơ chế đóng gói tùy ý. Trong thửa ban đầu của lập trình cấu trúc, nhiều người hiểu sai chân lý quan trọng: một cấu trúc chương trình tốt về cơ bản là một khái niệm ngữ nghĩa. Họ tìm kiếm một định nghĩa cú pháp đơn giản cho từ “được cấu trúc tốt”. Rất nhiều các định nghĩa đơn giản bao gồm một cận trên tùy ý, như một trang, về kích thước của thủ tục. Một ví dụ khác của hạn chế kích thước tùy ý xảy ra trong các chương trình mà cần quản trị bộ nhớ của riêng nó và được chia tách thành các module để làm giảm nhẹ bে ngoài. Các hạn chế tùy ý như vậy thường dẫn đến các thủ tục hoàn toàn mất tính kết dính.

Nguyên nhân thứ hai mất tính kết dính là tối ưu các chương trình bằng tay. Một lập trình viên dựa trên quan sát có thể nhận thấy rằng một nhóm bất kỳ nào đó các lệnh dường như xuất hiện một số lần. Và thử tiết kiệm bộ nhớ, lập trình viên đó có thể tách các lệnh này thành một thủ tục. Khi chạy lâu, tuy nhiên, các tối ưu như vậy nói chung là phản hiệu quả, vì chúng làm cho chương trình sửa đổi khó hơn.

Có hai chỉ thị tin cậy của việc mất hoàn toàn sự kết dính. Nếu nó có vẻ là cách tốt nhất để đặc tả một thủ tục bao gồm mô tả cấu trúc bên trong của nó (cái mà, nó làm việc như thế nào), thủ tục đó là có thể không gắn kết. Dấu hiệu tốt thứ hai là khó khăn trong việc tìm một cái tên phù hợp cho một thủ tục. Nếu tên tốt nhất mà chúng ta có thể có là “thủ tục 1”, có một cái gì đó có thể sai với thiết kế của chúng ta. Nếu không có sự kết dính bên ngoài cho một trùu tượng thủ tục, chúng ta cần nghĩ lại về thiết kế với mục đích loại bỏ thủ tục này.

Sự kết dính hội là một bước tiến từ không kết dính gì cả. Nó được chỉ ra bằng đặc tả dạng:

A && B && C && ...

Sự kết dính hội thường chỉ xuất hiện khi một dãy các hành động liên tiếp tạm thời được kết hợp vào một thủ tục duy nhất. Ví dụ điển hình là một trùu tượng mà công việc của nó là khởi tạo mọi cấu trúc dữ liệu. Đặc tả của một trùu tượng như vậy thường là một phép hội:

initialize A && initialize B && ...

Lưu ý rằng một cấu trúc như vậy có thể làm cho nó khó hơn để xác định trùu tượng dữ liệu vì phần của công việc của mỗi loại được lĩnh trách nhiệm bởi thủ tục đó.

Phương thức *isInteresting* của *WordTable* có thể xem là biểu hiện một kết dính hội, vì nó kiểm tra cả không là từ và từ không quan tâm. Tuy nhiên, có vẻ như là có cơ sở để xem một không từ là một từ không quan tâm, và khi đó kết hợp các kiểm tra này trong một phương thức đơn là chấp nhận được. Tuy nhiên, chúng ta có thể cần phải đi tiếp và có *isInteresting* được chuẩn hóa các đầu vào của nó và trả về dạng chuẩn nếu từ đó là quan tâm. Làm nó thay đổi có nghĩa là các phương thức *query* và *queryMore* của *Engine* trước cần thực hiện chỉ một lời gọi, còn bây giờ cần hai lời gọi (trước hết đến *canon* và sau đó cho *isInteresting*). Tuy nhiên, việc nhóm này trông có vẻ không mong muốn, vì chuẩn hóa một từ và kiểm tra từ đó có phải là quan tâm không có vẻ liên quan chặt chẽ với nhau.

Trong một môi trường, ở đó các lời gọi thủ tục là tốn kém quá mức, sự kết dính hội có thể là hữu ích, vì nó có thể loại bỏ một số lời gọi. Tuy nhiên, trừ khi các hành động có một kết nối logic mạnh, nói chung tốt hơn là không kết hợp các thủ tục. Càng nhiều thứ đưa vào một thủ tục, càng khó hơn bắt lỗi và bảo trì nó. Tiếp theo, như chúng ta bảo trì một chương trình chúng ta thường khám phá các trường hợp khi có thể là có ích thực hiện một tập con nào đó của hội. Nếu điều này xảy ra, việc thích hợp có thể làm là chia nhỏ thủ tục gốc. Cái mọi người thường làm thay vào đó, tuy nhiên, là bổ sung một trùu tượng thủ tục khác. Nó dẫn đến lập trình nhiều hơn để bắt lỗi và bảo trì, và dẫn đến một chương trình chiếm nhiều không gian hơn khi chạy. Một cách khác, lập trình viên có thể sửa trùu tượng gốc để có cò

kiểm soát tập con nào của công việc được thực thi. Điều này là ý tưởng tồi, vì nó dẫn sự gắn kết tuyển, mà sẽ bàn sau đây.

Sự gắn kết tuyển được chỉ ra bởi một đặc tả với một mệnh đề tác động có dạng:

$A \parallel B \parallel \dots$

thường ẩn trong dạng các mệnh đề if-then-else hoặc hội các phép suy. Một thủ tục thô như bao gồm một sự kết dính tuyển để tách trả về thông thường từ các ngoại lệ. Tuy nhiên, nếu đặc tả cái gì đã xảy ra khi thủ tục đó trả về thông thường gồm các tuyển, người ta cần phải lưu tâm. Xét một đặc tả ở Hình 4.2. Nếu *getEnd* trả về thông thường, nó có thể làm một trong hai việc khác nhau. Mỗi một trong hai cái đó có thể là một trùu tượng hoàn toàn hợp lý của riêng nó; điều này nói lên, chúng ta cần phải có hai trùu tượng trong Hình 4.3.

```
public static int getEnd (List a, int j)  
throws EmptyException, NullPointerException  
  
// REQUIRES:  $0 < j < 3$  và mọi phần tử của a là các số nguyên  
// EFFECTS: Nếu a là null throws NullPointerException  
// ngược lại nếu a là rỗng throws EmptyException  
// ngược lại nếu j = 1 trả về phần tử đầu tiên của a  
// ngược lại nếu j = 2 trả về phần tử thứ hai của a
```

Hình 4.2 Một ví dụ của sự kết dính tuyển

```
public static int getFirst (List a)  
throws EmptyException, NullPointerException  
  
// REQUIRES: Mọi phần tử của a là các số nguyên  
// EFFECTS: Nếu a là null throws NullPointerException  
// ngược lại nếu a là rỗng throws EmptyException  
// ngược lại trả về phần tử đầu tiên của a
```

```
public static int getLast (List a)  
throws EmptyException, NullPointerException
```

```
// REQUIRES: Mọi phần tử của a là các số nguyên  
// EFFECTS: Nếu a là null throws NullPointerException  
// ngược lại nếu a là rỗng throws EmptyException  
// ngược lại trả về phần tử cuối cùng của a
```

Hình 4.3 Hai thủ tục kết dính

Kết hợp hai thủ tục này vào một không có ưu điểm và có một số nhược điểm. Trước hết, một lời gọi dạng `getEnd (a, 1)` như là khó hiểu hơn đối với bạn đọc so với một lời gọi `getFirst (a)`. Thứ hai, một lớp mới của lỗi là có thể - chẳng hạn, một lời gọi dạng `getEnd (a, 3)`. Thứ ba, một chương trình sử dụng `getEnd` là ít hiệu quả hơn một người sử dụng hai trùu tượng trong Hình 4.3. Mỗi khi một lời gọi `getEnd` được thực hiện, người gọi biết thủ tục nào sẽ được mong muốn. Tuy nhiên, thông tin này cần được mã để đưa vào đối số thứ hai của lời gọi, và `getEnd` cần kiểm tra đối số này để quyết định cần làm gì. Công việc thêm này đòi hỏi cả thời gian và không gian. Cũng như vậy, chúng ta có thể cài đặt `getEnd` sử dụng các trùu tượng phụ như `getFirst` và `getLast`, như vậy làm tăng số lời gọi thủ tục để được thực hiện.

Sự gắn kết tuyển thường được nảy sinh từ nỗ lực thiếu chỉ dẫn để tổng quát hóa các trùu tượng. Khi một thiết kế chương trình chứa hai hoặc nhiều hơn các trùu tượng tương tự, thường là có ý nghĩa xem xét một trùu tượng duy nhất tổng quát hơn thay thế mọi hoặc một số trùu tượng tương tự. Nếu thành công, sự tổng quát này có thể tiết kiệm không gian và công sức của lập trình viên với giá ít hơn trong tốc độ thực hiện hoặc độ phức tạp trong cài đặt trùu tượng tổng quát. Tuy nhiên, nếu kết quả là một trùu tượng với kết dính tuyển, thì nó thường không tốt hơn là không thay thế.

Đôi khi, dường như sự kết dính tuyển thừa chỉ ra lỗi khi đưa ra một trùu tượng dữ liệu hợp lý vào trong thiết kế. Trong trường hợp như vậy, sự kết hợp một số chức năng khác nhau vào một thủ tục có thể là một dự định đóng gói thông tin thể hiện mà cần phải đóng gói trong một kiểu bị khuyết. Kết quả là kiểu này được cài đặt bởi một thủ tục duy nhất, và các đối số thừa là được sử dụng để xác định thao tác sẽ được gọi.

Kết dính tuyển không phải luôn là tồi (và cũng không là kết dính hội). Chẳng hạn, một chương trình dịch có thể có thể nhận một đối tượng môi trường làm đối số và tạo ra các đầu ra khác nhau phụ thuộc vào môi trường. Điều này thể hiện một kiểu kết dính tuyển. Tuy nhiên, nó cũng thể hiện một tổng quát của chương trình dịch (để kiểm soát một số dạng môi trường); ở đây lợi ích vượt qua nhược điểm. Nói chung, sự kết dính hội hoặc tuyển là cho phép trong thiết kế, nhưng chỉ nếu ở đây có giải thích tốt vì sao nó đáng giá.

Sự kết dính các kiểu

Mỗi phương thức của một kiểu cần là một thủ tục hay bộ duyệt chặt chẽ. Ngoài ra, một kiểu cần cung cấp một trùu tượng mà những người sử dụng nó có thể suy nghĩ thích hợp về tập các giá trị và tập các phương thức gắn kết về bản chất với các giá trị đó. Một cách đánh giá sự kết dính của kiểu là kiểm tra mỗi phương thức để xem nó có thực sự thuộc kiểu đó không. Một kiểu cần là hợp lý – tức là, cần cung cấp đủ phương thức sao cho sự sử dụng chúng là hiệu quả. Trong các kiểu được thiết kế kém, người ta thường tìm thấy các phương thức bổ sung mà không trông như liên quan cụ thể đến trùu tượng đó và cài đặt của nó có thể có ít hoặc không có ưu thế truy cập trực tiếp đến biểu diễn đó. Nói chung là tốt hơn chuyển các phương thức như vậy ra khỏi kiểu đó. Nếu có ít các thao tác có truy cập đến biểu diễn đó, sẽ dễ dàng thay đổi biểu diễn nếu nó trở nên mong muốn làm như vậy.

Chẳng hạn, ta xem xét kiểu ngăn xếp chứa phương thức *sqrtTop*:

```
float sqrtTop() throws EmptyException  
  
// EFFECTS: Nếu this.size = 0 throws EmptyException, ngược lại trả về căn  
// bậc hai của this.top.
```

sqrtTop có ít thứ được làm với ngăn xếp, và cài đặt của nó có thể chạy tốt mà không cần truy cập đến biểu diễn ngăn xếp. Do đó, phương thức này cần được đưa ra khỏi trùu tượng ngăn xếp.

Trao đổi giữa các modules

Một kiểm tra cẩn thận bao nhiêu và kiểu thông tin nào được trao đổi giữa các modules có thể khám phá các thiếu sót quan trọng trong một thiết kế. Chúng ta đã nhấn mạnh tính quan trọng của các giao diện hẹp: một module cần có truy cập đến chỉ vừa đủ thông tin mà nó cần để thực hiện công việc của nó. Phương pháp của chúng ta được thiết kế để ủng hộ giao diện hẹp – chẳng hạn, bằng việc không cho phép các thủ tục tham chiếu đến các biến tổng thể - nhưng nó vẫn có khả năng truyền nhiều thông tin cho một module.

Một module có thể được truyền quá nhiều thông tin, vì kiểu này không được xác định. Khi thiếu một kiểu, mọi modules mà cần có trao đổi ở dạng các đối tượng trùu tượng thay vì trao đổi theo thuật ngữ các biến thể hiện. Kết quả là các module mà có giao diện rộng hơn cần thiết; thay vì liên quan chỉ qua kiểu, chúng chia sẻ thông tin về kiểu đó được cài đặt như thế nào. Điều này bao gồm hàm trùu tượng và các bát biến biểu diễn bổ sung vào chính biểu diễn đó. Lưu ý rằng tất cả những cái mà sử dụng modules cần được xem xét trong suy luận rằng cài đặt của kiểu thiếu đó là đúng đắn. Tiếp theo, nếu cài đặt của một kiểu thay đổi, mỗi cái sử dụng module cũng cần thay đổi theo.

Ngay cả nếu mọi cái cần kiểu đã được xác định và được cài đặt bởi các lớp của riêng chúng, một số giao diện cũng có thể vẫn là rộng hơn cần thiết. Các chương trình được thiết kế tốt thông thường có các kiểu mà bao gồm có nhiều thông tin. Một số modules có thể không

cần truy cập đến mọi thông tin này. Đã có nhiều thiết kế gọi truyền toàn bộ một đối tượng trừu tượng khi một mẫu nhỏ của nó có thể đã là đủ. Chẳng hạn, chúng ta có thể có một kiểu *StudentRecord* mà bao gồm giữa những thứ khác, là tên sinh viên, số an ninh xã hội, nơi ở và bảng điểm. Một thủ tục, *printAddress*, mà in một nhãn địa chỉ có thể sẽ cần chỉ tên sinh viên và nơi ở. Một thủ tục như vậy không cần được truyền cả một bản ghi sinh viên; thay vào đó, thông tin mà cần chiết xuất ra từ bản ghi sinh viên bởi người gọi nó và truyền cho nó một cách tường minh. Có một số lý do tốt cho điều đó:

- Nếu *printAddress* được truyền một đối tượng kiểu *StudentRecord*, người cài đặt nó sẽ cần biết làm thế nào để chiết xuất thông tin cần thiết – tức là, phương thức nào cần gọi. nếu đặc tả *StudentRecord* thay đổi, cài đặt đó của *printAddress* có thể phải thay đổi. Không cái nào của *this* sẽ là cần thiết nếu thông tin cần thiết được truyền tường minh.
- Cài đặt của *printAddress* có thể có lỗi mà buộc nó thay đổi *StudentRecord*. Các lỗi như vậy là rất khó để tìm ra.

Giảm sự phụ thuộc

Một thiết kế với ít hơn sự phụ thuộc nói chung là tốt hơn so với cái nhiều hơn. Có ít hơn sự phụ thuộc có thể cho kết quả từ các giao diện hẹp hơn; chẳng hạn, truyền một phần tử thay vì cả một tập hợp *Set* có thể có nghĩa là trừu tượng được gọi không còn phụ thuộc vào tập hợp *Set* nữa. Nó cũng cho kết quả thay đổi từ phụ thuộc mạnh sang phụ thuộc yếu. Một thay đổi như vậy có thể là một cải tiến, vì một module với phụ thuộc yếu sẽ không bị tác động bởi thay đổi đến đặc tả của một trừu tượng dữ liệu mà nó phụ thuộc vào.

Chẳng hạn, trong thiết kế *Engine*, *WordTable* phụ thuộc vào *Doc* vì phương thức *addDoc* của nó gọi phương thức *words* của *Doc*; tình huống tương tự tồn tại cho *TitleTable*. Chúng ta có thể giảm sự phụ thuộc này thành phụ thuộc yếu bằng cách thay đổi thiết kế một chút. Trước hết, thay vì có phương thức *addDoc* của *WordTable* gọi bộ duyệt *words*, chúng ta có thể thay vào đó truyền cho nó bộ sinh được trả về bởi bộ duyệt *words*:

Hash table addDoc (Iterator e, Doc d)

// REQUIRES: e sinh ra các xâu

// MODIFIES: this

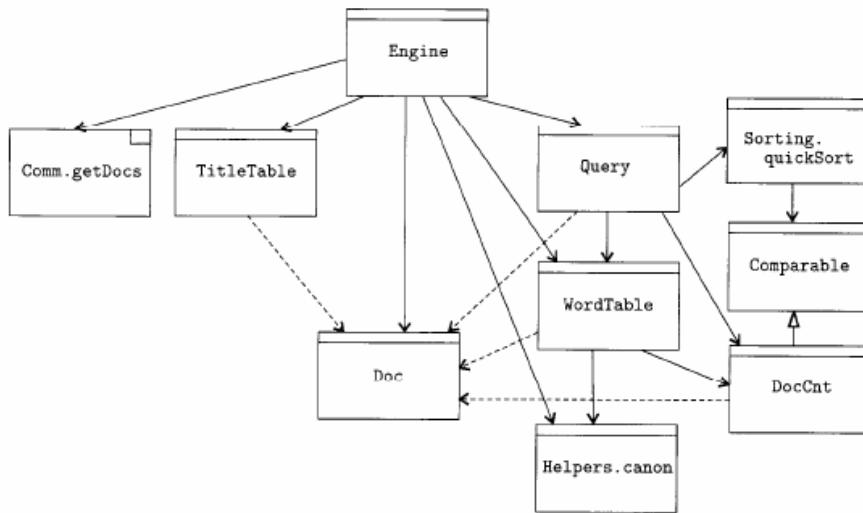
// *Bổ sung mỗi từ quan tâm w được sinh bởi e vào this, ánh xạ vào d và số lần*
// *xuất hiện của w trong e; cũng trả về một bảng ánh xạ mỗi từ quan tâm trong*
// *e vào số lần xuất hiện của nó trong e.*

Thứ hai, chúng ta có thể thay đổi đặc tả cho phương thức *addDoc* của *TitleTable* thành:

```

void addDoc (String t, Dic d) throws DuplicateException
    // REQUIRES: d và t là khác null
    // MODIFIES: this
    // EFFECTS: Nếu một tài liệu với tiêu đề t đã có trong this throws
    // DuplicateException, ngược lại bổ sung d vào this với tiêu đề t.

```



Hình 4.4 Lược đồ phụ thuộc module với thiết kế thay đổi

Lược đồ phụ thuộc module trên, là kết quả từ các thay đổi, được chỉ ra trên *Hình 4.4*. Một ưu điểm của cấu trúc này so với cái được nêu ra trong *Hình 3.20* là *TitleTable* không cần chuẩn hóa tiêu đề nữa, vì nó có thể được làm trong *Engine*, và do đó ở đây có ít sự phụ thuộc hơn trong thiết kế này. Một ưu điểm khác là cấu trúc này thích nghi dễ dàng hơn với các thay đổi tiềm năng được bàn trong phân tích bài toán, ở đó các thân tài liệu được bỏ qua và được lấy lại từ trang khi cần thiết. Với cấu trúc xem xét lại này, các thay đổi với các bảng *TitleTable* và *WordTable* là không cần thiết phải thích ứng với thay đổi đó.

Bảng 4.2 Tổng kết điều mong ước với cấu trúc chương trình:

- Mỗi trừu tượng cần là chặt chẽ: nó cần có mục đích rõ ràng, và nếu đặc tả của nó thể hiện sự kết dính hội hoặc tuyển, thì cần phải có lời giải thích thuyết phục. Tiếp theo, các kiểu dữ liệu cần không chứa các phương thức mà không liên quan đến mục đích của chúng.
- Các giao diện của các trừu tượng cần không rộng quá cần thiết.
- Một thiết kế với ít sự phụ thuộc thường là tốt hơn cái với nhiều hơn, và sự phụ thuộc yếu là tốt hơn sự phụ thuộc mạnh.

Bảng 4.2 Phản biện thiết kế: Đánh giá cấu trúc chương trình

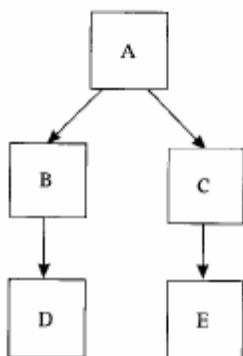
4.2 Thứ tự tiến trình phát triển chương trình

Chúng ta đã nhấn mạnh chiến lược phát triển chương trình mà cơ bản dựa vào trên-xuống. Trong khi chúng ta công nhận rằng phát triển chương trình là quá trình tương tác, chúng ta đã tranh luận rằng, trong mỗi bước lặp, đặc tả và thiết kế cần được thực hiện trước cài đặt, ít nhất cho phần cài đặt mà đã bắt đầu. Chúng ta chưa bàn làm thế nào sắp xếp tiến trình đi từ một thiết kế đến một cài đặt. Lựa chọn chính là kết hợp giữa một chiến lược trên-xuống và một chiến lược dưới lén.

- Trong phát triển, trên-xuống, mọi modules mà sử dụng module M được cài đặt và kiểm thử trước khi M được cài đặt.
- Trong phát triển dưới lén, mọi module được sử dụng bởi module M được cài đặt và kiểm thử trước khi M được cài đặt.

Bảng 4.3 Phát triển trên-xuống và dưới lén

Chế độ truyền thống phát triển là dưới lén. Trong phát triển dưới lén, chúng ta cài đặt và kiểm thử mọi modules được sử dụng bởi một module M trước khi chúng ta cài đặt và kiểm thử M. Chẳng hạn, xem xét cài đặt một thiết kế với lược đồ phụ thuộc module trên Hình 4.7. Chúng ta có thể bắt đầu cài đặt và kiểm thử đơn vị D và E. Sau đó chúng ta cài đặt và kiểm thử B và C; ở đây chúng ta có thể sử dụng D và E để tránh viết bẩn nháp tạm. Cuối cùng, chúng ta có thể cài đặt và kiểm thử A. Lưu ý rằng, bằng cách này, mà nó không chỉ là thứ tự dưới lén có thể. Chúng ta có thể sử dụng thứ tự cũng tốt như nhau là D, B, E, C, A.



Hình 4.7 Lược đồ phụ thuộc module đơn giản

Khi chúng ta sử dụng D và E trong kiểm thử B và C chúng ta đã không thực hiện kiểm thử đơn vị nữa, vì chúng ta không kiểm thử một module riêng biệt. Điều này là cả tốt và xấu. Nó xấu, vì B và C có thể phụ thuộc chi tiết cài đặt (và lỗi) trong mã thực tế của D và E. Tuy nhiên nó tốt, vì nó có thể khó viết tạm bản nháp cho một số trùu tượng, và một ưu điểm của cài đặt dưới lên là chúng ta tránh được việc viết các bản nháp.

Trong phát triển trên xuống, chúng ta cài đặt và kiểm thử mọi modules mà sử dụng module M trước khi cài đặt và kiểm thử M. Các thứ tự trên-xuống có thể đối với ví dụ trước bao gồm A, B, C, D, E và A, C, E, B, D. Chỉ như phát triển dưới lên giảm sự phụ thuộc vào bản nháp tạm, phát triển trên xuống giảm sự phụ thuộc của chúng ta vào cái dẫn dắt drivers. Nó là quan trọng, tuy nhiên, phát triển trên xuống được đồng hành bằng việc kiểm thử đơn vị cẩn thận của mọi modules. Nếu chúng ta kiểm thử chỉ B vì rằng nó được sử dụng bởi A, chúng ta có thể thấy chỉ phần của hành vi đặc tả của B. Sau đó nếu chúng ta thay đổi A như chỉ dẫn driver của A, các lỗi mới có thể được lộ ra trong B. Do đó, nếu chúng ta chọn A như chỉ dẫn driver của B, chúng ta cần tin tưởng rằng A kiểm thử B xuyên suốt mọi khả năng. Ngược lại chúng ta cần sử dụng một chỉ dẫn driver riêng biệt để kiểm thử B.

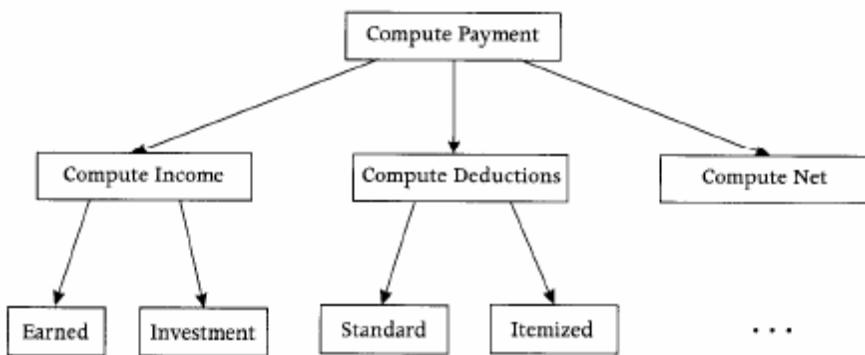
Không chiến lược phát triển nào trội hơn cái kia. Hầu hết thời gian, có vẻ tốt nhất là làm việc trên-xuống. Tuy nhiên, có thể có các nguyên nhân thúc dục theo đuổi tiếp cận dưới lên. Chúng ta ủng hộ một chiến lược pha trộn ở đó người ta làm việc trên-xuống trên một số phần của hệ thống và dưới-lên trên một số phần khác.

Phát triển trên-xuống có ưu điểm giúp chúng ta nắm bắt được các lỗi thiết kế sớm. Trong khi chúng ta kiểm thử một module, chúng ta sẽ kiểm thử không chỉ cài đặt của module đó, mà còn đặc tả của các modules sử dụng nó. Nếu chúng ta tuân theo chiến lược dưới-lên, chúng ta có thể dễ dàng nỗ lực cài đặt và kiểm thử các modules mà chúng không còn có ích, vì có vấn đề thiết kế của một trong những cái trước chúng trong lược đồ phụ thuộc module. Một vấn đề tương tự có thể xảy ra trong phát triển trên xuống, nếu chúng ta phát hiện được một module nào đó sau nó không cài đặt được hoặc không thể được cài đặt với hiệu năng chấp nhận được. Kinh nghiệm chỉ ra rằng, tuy nhiên, vấn đề này xảy ra ít thường xuyên hơn. Nó có thể vì các trùu tượng mức thấp hơn có xu thế là có đặc thù thấy trước hơn.

Trong phát triển trên-xuống, luôn là có thể tích hợp một module tại mỗi thời điểm. Chúng ta đơn giản thay một bản nháp tạm bằng một module mà nó được sử dụng để mô phỏng. Trong phát triển dưới-lên, mặt khác, chúng ta hướng tới tích hợp một số modules cùng một lúc; trong hầu hết các trường hợp, một module mức cao đơn tương ứng không chỉ một mà một số chỉ dẫn drivers. Chẳng hạn, khi A được tích hợp với chương trình trên Hình 4.8, nó sẽ được thay thế là driver cho cả B và C. Vì tích hợp hệ thống hướng đến êm ái hơn, nếu

chúng ta có thể bổ sung một giao diện tại mỗi thời điểm, phát triển trên-xuống có ưu điểm rất đáng kể liên quan đến việc này. Tiếp theo, tích hợp hệ thống có thể là nút thắt cổ chai thực tế khi hoàn thành phát triển: với phát triển dưới-lên, một hệ thống là “90% được cài đặt xong” có thể yêu cầu nhiều hơn 10% thời gian để hoàn thành, vì những gì còn lại là tích hợp hệ thống.

Khi một thiết kế chứa một phân cấp kiểu, phát triển trên xuông là được yêu cầu. Trong trường hợp này, chúng ta cần xem xét cài đặt của các kiểu cha trước các kiểu con, vì vậy chúng ta quyết định các phần nào của cài đặt trong toàn bộ gia đình đó là được kiểm soát bởi kiểu cha và cái gì cần được trì hoãn đến các kiểu con.



Hình 4.8 Lược đồ phụ thuộc module cho chương trình thuế thu nhập

Phát triển trên-xuống làm tăng khả năng đem đến các phiên bản bộ phận có ích của chương trình đang được phát triển. Giả sử một chương trình tính trả thuế thu nhập có lược đồ phụ thuộc module (bộ phận) trên Hình 4.8. Bằng cách là việc trên-xuống, có thể mang đến nhiều cài đặt bộ phận có ích của thiết kế này. Một hệ thống không có các thủ tục đầu tư *investment* và ghi thành từng khoản *itemized*, chẳng hạn, có thể là đủ hữu ích cho một số người.

Ngay cả nếu các phương án bộ phận của một hệ thống không thể được sử dụng hiệu quả bởi khách hàng, cũng mang đến các phiên bản sớm trong quá trình phát triển, có thể có một số lợi ích quan trọng. Nếu các cài đặt là có thể trình diễn một cái gì đó sớm, họ có thể nhận được phản hồi mà để lộ ra một số vấn đề với các yêu cầu cho chương trình. Tiếp theo, với các dự án dài hơi, khả năng tạo ra một chuỗi các hệ thống bộ phận làm việc trông có vẻ giúp ích về mặt tinh thần cho cả khách hàng và những người phát triển.

Trong khi phát triển dưới-lên trì hoãn thời điểm ở đó người ta có hệ thống bộ phận làm việc, nó dẫn đến hoàn thành sớm các hệ thống con có ích. Các hệ thống con nói chung có tính ứng dụng rộng hơn so với các hệ thống bộ phận có mà chúng ta nhận được khi làm việc trên-xuống. Điều này là đặc biệt đúng đối với các hệ thống con mức thấp (chẳng hạn,

các hệ thống con đầu vào/đầu ra). Trong một môi trường, ở đó một số hệ thống con liên quan đang được phát triển đồng thời, nó có thể là hữu ích đem đến các hệ thống con chia sẻ bằng cách làm việc dưới-lên. Hơn nữa, đôi khi dễ dàng xây dựng một hệ thống con mức thấp hơn là xây dựng hệ thống nháp tạm thời mô tả nó.

Một ưu điểm tiềm năng khác của phát triển dưới lên là nó có thể đặt ít tải trọng lên máy nguồn so với phát triển trên xuống. Phát triển trên xuống hướng tới sử dụng thời gian máy nhiều hơn, vì chúng ta chạy một phần lớn hơn của hệ thống theo nhiều pha của phát triển. Thông thường, kiểm thử đầy đủ của hệ thống sẽ được phát triển và sau đó chạy mỗi lần khi một module mới được bổ sung. Các kiểm thử như vậy có thể thực hành nhiều phần khác của hệ thống dựa trên module đang được bổ sung, và như vậy tiêu thụ nhiều thời gian và không gian mà thực sự không cần thiết cho bước tích hợp này. Mặt khác, một cách tiếp cận hệ thống như vậy để kiểm thử là được giới hạn để khám phá nhiều lỗi hơn và là xuyên suốt nhiều hơn một cách tiếp cận được sắp đặt hơn.

Một ưu điểm liên quan của phát triển dưới lên là có thể cho phép chúng ta tiếp tục khi thiếu vắng một số nguồn lực tính toán. Xem xét việc xây dựng một hệ thống mà là chạy trên một máy với dung lượng bộ nhớ rất lớn. Nếu máy đó chưa có, nhưng một máy tương tự với ít bộ nhớ hơn đã sẵn sàng, chúng ta có thể thực hiện hoàn toàn phát triển dưới lên một chút trước khi bộ nhớ trở thành vấn đề. Chúng ta có thể chạy bị thiếu bộ nhớ sớm hơn nếu làm việc trên xuống.

Nói chung, tốt nhất là sử dụng chiến lược hỗn hợp, ở đó phát triển trên xuống ưu tiên hơn, nhưng không theo suốt toàn bộ. Trong khi tạo ra chiến lược này, các yếu tố cả kỹ thuật và phi kỹ thuật đều cần được xem xét. Vấn đề quan trọng nhất là một chiến lược phát triển cần được xác định tường minh trước khi cài đặt bắt đầu.

Bảng 4.4 tổng kết các vấn đề nảy sinh khi chọn thứ tự phát triển.

- Một chiến lược phát triển cần được xác định tường minh trước khi cài đặt bắt đầu.
- Nó cần chủ yếu là trên xuống, với dưới lên được sử dụng chính đối với các modules mà là dễ dàng cài đặt hơn là mô phỏng, vì rằng các ưu điểm của phát triển trên xuống tác động hơn các ưu điểm của phát triển dưới lên:
 - Phát triển trên xuống làm đơn giản rất nhiều tích hợp hệ thống và kiểm thử.
 - Phát triển trên xuống làm cho có thể tạo ra các phiên bản bộ phận hữu ích của hệ thống.
 - Phát triển trên xuống cho phép các lỗi thiết kế mức cao rất quan trọng được phát hiện sớm.

Bảng 4.4 Chiến lược phát triển

Một sự thấu hiểu của thứ tự cài đặt và kiểm thử có thể thấy rõ bằng việc xét lại máy tìm kiếm:

1. *Doc* cần được cài đặt và kiểm thử đơn vị sao cho chúng ta có thể xử lý các tài liệu thực tế trong suốt quá trình kiểm thử.
2. Bây giờ chúng ta có thể cài đặt *Engine*. Nó sẽ sử dụng *Doc* và cũng như bộ duyệt *getDocs* thực tế (vì nó được cung cấp bởi một thư viện) trừ các bản nháp tạm thời cho *canon*, *Query*, và các bảng *TitleTable* và *WordTable*, *canon* chỉ trả về đầu vào của nó. Các phương thức *addDoc* của hai bảng sẽ đơn giản lưu lại tài liệu dạng một mảng. Ngoài ra, phương thức *isInteresting* của *WordTable* trả về *false* và phương thức *lookup* của *TitleTable* tung ra ngoại lệ. Chỉ *Query* rỗng sẽ được cài đặt vì mọi từ đang đều là không quan tâm. Bây giờ chúng ta kiểm thử một lượng hợp lý *logic* của *Engine*; chúng ta sẽ có thể nhìn thấy các lời gọi đúng được thực hiện trên hai bảng, và chúng ta có thể cũng kiểm thử một số trường hợp cơ bản (chẳng hạn, các từ tồi và không quan tâm trong truy vấn, tiêu đề mà không sánh).
3. Tiếp theo chúng ta cài đặt và kiểm thử đơn vị thủ tục *canon*. Sau đó chúng ta sẽ cài đặt *TitleTable* và kiểm thử đơn vị nó sử dụng *Doc* và *canon*. Điều này cho phép chúng ta kiểm thử tìm kiếm dựa trên tiêu đề.
4. Tiếp theo, chúng ta cài đặt và kiểm thử đơn vị với *WordTable* sử dụng *Doc* và *canon*.
5. Tiếp theo chúng ta mở rộng bản nháp tạm thời cho *Query* sao cho kích thước trả về là 0 và các thay đổi khác nhau ghi chép các đối số của chúng. Điều này cho phép chúng ta hoàn thành kiểm thử logic của *Engine*.
6. Cuối cùng chúng ta cài đặt *DocCnt* và *Query* và kiểm thử đơn vị chúng sử dụng *Doc* và *WordTable*. Sau đó chúng ta chạy toàn bộ *Engine*.

Chiến lược này cho phép chúng ta nhận được phiên bản sớm của *Engine*: chúng ta sẽ tìm kiếm theo tiêu đề sau bước 3.

Chiến lược này là tiêu biểu cho các chiến lược kiểm thử trong khi hoặc chúng ta đã cài đặt toàn bộ trừu tượng dữ liệu, hoặc chúng ta sử dụng các bản nháp tạm đơn giản: một số phương thức ghi lại đầu vào của chúng, hoặc trả về phản hồi chuẩn, và còn những cái khác chưa được cài đặt, vì chúng không được gọi ở giai đoạn này của phát triển. Chẳng hạn, trong bước 2, phương thức *addDoc* của *WordTable* ghi lại các đối số của nó, phương thức *isInteresting* có phản hồi chuẩn, và phương thức *lookup* không được gọi. Tuy nhiên, đôi khi chúng ta đã cài đặt một trừu tượng qua nhiều giai đoạn. Chẳng hạn, chúng ta có thể cung cấp một cài đặt hoàn chỉnh của một truy vấn rỗng (như một kiểu con của *Query*) trong bước 2; các truy vấn khác rỗng có thể được cài đặt về sau.

4.3 Tóm tắt

Trong chương này, chúng ta đã bàn một số việc mà cần phải được thực hiện giữa hoàn thiện thiết kế và bắt đầu cài đặt. Các vấn đề chính cần giải quyết là tầm quan trọng của việc chỉ dẫn đánh giá một cách có hệ thống thiết kế và phát triển kế hoạch đặc tả thứ tự cài đặt và kiểm thử các modules có trong thiết kế.

Trong phản biện thiết kế, người ta xem xét có hay không các cài đặt của thiết kế sẽ thể hiện hành vi và hiệu năng mong muốn, và có hay không cấu trúc chương trình được mô tả bởi thiết kế sẽ là dễ dàng hợp lý để xây dựng, kiểm thử và bảo trì. Chúng ta đề xuất chỉ dẫn phản biện thiết kế bằng việc giám sát đường đi của các dữ liệu kiểm thử tượng trưng xuyên suốt thiết kế. Chúng ta cũng đề xuất một số tiêu chuẩn mà cũng được sử dụng trong đánh giá các vấn đề cấu trúc. Mọi tiêu chuẩn đều liên quan đến tính phù hợp của các biến module.

Chúng ta không cố định các qui tắc chọn thứ tự cài đặt và kiểm thử. Chúng ta bàn về các định lượng tương đối của phát triển và kiểm thử trên xuống và dưới lên. Kết luận của chúng ta là hầu hết thời gian, cách tốt nhất là tuân theo chiến lược hỗn hợp nhưng với điểm nhấn là hướng tới trên xuống.

Trong bàn luận của chúng ta về phản biện thiết kế, chúng ta thể hiện phản biện cực ngắn gọn về máy tìm kiếm trong chương trước. Đối với các chương trình lớn, bắt buộc là chúng ta chỉ đạo phản biện cẩn thận trong pha thiết kế. Nếu chúng ta muốn bắt đầu cài đặt một hệ thống con nào đó trước khi thiết kế hoàn thành, thì điều này cần xảy ra sau khi thiết kế đến điểm đó đã được phản biện một cách chu đáo. Nhưng ngay cả nếu chúng ta không lập kế hoạch cài đặt sớm, chúng ta cũng cần phản biện trước khi thiết kế hoàn thành, vì nó cần được phê phán để nắm bắt lỗi thiết kế sớm.

Cũng là rất quan trọng bắt đầu xem xét các nỗ lực cài đặt được tổ chức sớm trong thiết kế. Sự cần thiết hoàn thành sớm các hệ thống con, chẳng hạn, có thể có tác động quan trọng đến các quyết định thiết kế và đến thứ tự mà các trùu tượng được nghiên cứu trong quá trình thiết kế.

CHƯƠNG 5: ĐẶC TẢ HÌNH THỨC

5.1 Quan hệ

Tập $\alpha \leftrightarrow \beta$ là tập mọi quan hệ hai ngôi giữa tập α và tập β , có nghĩa là $\alpha \leftrightarrow \beta$ là viết tắt của $P(\alpha \times \beta)$ là tập tất cả các tập con của tích đề các $\alpha \times \beta$, vì mỗi một quan hệ tương ứng với một tập con của tích đề các đó.

Biểu thức $a \rightarrow b$ nghĩa là a là liên quan đến b , mà $a \rightarrow b$ là viết tắt của cặp có thứ tự (a, b) .

Ví dụ 5.1:

$$\{a \mapsto x, b \mapsto y\} \in (\{a, b\} \leftrightarrow \{x, y\})$$

Quan hệ nhị phân $<$ trên tập các số tự nhiên là kiểu $N \leftrightarrow N$, tức là một phần tử của $N \leftrightarrow N$.

Chúng ta thường viết $x < y$ là viết tắt của ký hiệu $(x, y) \in <$.

Miền xác định dom và miền giá trị ran

$$\begin{aligned} \forall r. (r \in X \leftrightarrow Y \Rightarrow \\ \text{dom } r = \{x \mid x \in X \wedge \exists y. (y \in Y \wedge (x \mapsto y) \in r)\} \wedge \\ \text{ran } r = \{y \mid y \in Y \wedge \exists x. (x \in X \wedge (x \mapsto y) \in r)\}) \end{aligned}$$

Phép hợp quan hệ

$$\begin{aligned} \forall r. (r \in X \leftrightarrow Y \Rightarrow \forall s. (s \in Y \leftrightarrow Z \Rightarrow \\ r ; s = \{x, z \mid x \in X \wedge z \in Z \wedge \exists y. (y \in Y \wedge \\ (x \mapsto y) \in r \wedge (y \mapsto z) \in s)\})) \end{aligned}$$

Ví dụ 5.2:

$$\begin{aligned} \text{dom}\{a \mapsto x, b \mapsto y, a \mapsto y\} &= \{a, b\} \\ \{a \mapsto x, b \mapsto y, a \mapsto y\} ; \{x \mapsto 1, x \mapsto 3, y \mapsto 2\} \\ &= \{a \mapsto 1, a \mapsto 3, a \mapsto 2, b \mapsto 2\} \end{aligned}$$

\lhd là hạn chế miền xác định dom

\lhd là hạn chế đối miền xác định

\rhd là hạn chế miền giá trị ran

\rhd là hạn chế đối miền giá trị

Chúng được định nghĩa như sau:

$$\begin{aligned} \forall s. (s \in \mathbb{P} X \Rightarrow \forall r. (r \in X \leftrightarrow Y \Rightarrow \\ s \triangleleft r = \{x, y \mid x \in s \wedge (x \mapsto y) \in r\} \wedge \\ s \triangleleft r = \{x, y \mid x \notin s \wedge (x \mapsto y) \in r\})) \\ \forall s. (s \in \mathbb{P} Y \Rightarrow \forall r. (r \in Y \leftrightarrow X \Rightarrow \\ r \triangleright s = \{x, y \mid y \in s \wedge (x \mapsto y) \in r\} \wedge \\ r \triangleright s = \{x, y \mid y \notin s \wedge (x \mapsto y) \in r\})) \end{aligned}$$

Ví dụ 5.3:

$$\begin{aligned} \{a\} \triangleleft \{a \mapsto x, a \mapsto y, b \mapsto y\} &= \{a \mapsto x, a \mapsto y\} \\ \{a\} \triangleleft \{a \mapsto x, a \mapsto y, b \mapsto y\} &= \{b \mapsto y\} \end{aligned}$$

Nếu chúng ta sử dụng quan hệ để biểu diễn mỗi quan hệ giữa những cuốn sách được mượn và những người mượn chúng, tức là

$$onLoanTo \in READERID \leftrightarrow BOOKID$$

Thì các cuốn sách được mượn bởi một bạn đọc nào đó (rid) sẽ được biểu diễn dạng biểu thức sau

$$\text{ran}(\{rid\} \triangleleft onLoanTo)$$

Ảnh quan hệ

$$\forall s. (s \in \mathbb{P} X \Rightarrow \forall r. (r \in X \leftrightarrow Y \Rightarrow \\ r[s] = \{x \mid \exists y. (y \mapsto x \in r \wedge y \in s)\}))$$

Quan hệ ngược

$$\forall r. (r \in X \leftrightarrow Y \Rightarrow \\ r^{-1} = \{x, y \mid y \mapsto x \in r\})$$

Quan hệ đồng nhất

Cho trước tập S

$$id(S) = \{p \mid \exists x. (x \in S \wedge p = x \mapsto x)\}$$

Ví dụ 5.4:

$$\{a \mapsto x, a \mapsto y, b \mapsto z\}[\{a\}] = \{x, y\}$$

$$\{a \mapsto x, a \mapsto y, b \mapsto z\}^{-1} = \{x \mapsto a, y \mapsto a, z \mapsto b\}$$

$$id(\{a, b\}) = \{a \mapsto a, b \mapsto b\}$$

Quan hệ (và hàm số) nói chung được sử dụng để lưu giữ dữ liệu và thông thường các quan hệ này cần được cập nhật thông qua các phép toán. Hai phép cập nhật thường dùng là add – thêm một ánh xạ mới hoặc cập nhật một ánh xạ đã có. Để thêm một ánh xạ mới chúng ta có thể sử dụng phép hợp, nhưng để cập nhật chúng ta thường sử dụng phép ghi đè quan hệ.

Ghi đè quan hệ là thường được sử dụng cho các hàm số hơn quan hệ, vì nó giữ tính chất của hàm, trong khi hợp thì không.

Ý tưởng của ghi đè quan hệ là viết đè lên một quan hệ bằng quan hệ khác.

Nó được định nghĩa như sau

$$\begin{aligned} \forall r, s. (r \in X \leftrightarrow Y \wedge s \in X \leftrightarrow Y \Rightarrow \\ r \lhd s = ((\text{dom } s) \lhd r) \cup s) \end{aligned}$$

Ví dụ 5.5:

$$\begin{aligned} \{1 \mapsto a, 2 \mapsto b\} \lhd \{2 \mapsto c, 3 \mapsto d\} \\ = \{1 \mapsto a, 2 \mapsto c, 3 \mapsto d\} \end{aligned}$$

Bây giờ chúng ta sẽ sử dụng một số kiểu hàm số như: hàm bộ phận, hàm tổng thể, hàm đơn trị và toàn ánh.

Chúng ta bắt đầu xem xét **hàm bộ phận**, vì nó thường được sử dụng như một kiểu hàm, và vì các kiểu hàm khác là các trường hợp đặc biệt của hàm bộ phận.

Tập $\alpha \rightarrow \beta$ là tập tất cả các hàm số từ α vào β .

$$\begin{aligned} \alpha \rightarrow \beta = \\ \{f \mid f \in \alpha \leftrightarrow \beta \wedge \\ \forall u, v, w. (u \in \alpha \wedge v \in \beta \wedge w \in \beta \Rightarrow \\ (u \mapsto v) \in f \wedge (u \mapsto w) \in f \Rightarrow v = w)\} \end{aligned}$$

Ví dụ 5.6:

Trở lại bài toán hệ thống thư viện, chúng ta thấy đối với mỗi cuốn sách sẽ có nhiều nhất một người mượn và như vậy cách biểu diễn tốt hơn người mượn sẽ là sử dụng hàm số mà gắn kết người mượn với các cuốn sách được mượn.

$$onLoanTo \in BOOKID \rightarrow READERID$$

Cho trước bid là cuốn sách được mượn, khi đó người mượn cuốn sách bid đó là

$$onLoanTo(bid)$$

Và tập các cuốn sách được mượn bởi bạn đọc rid là

$$onLoanTo^{-1}[\{rid\}]$$

Chúng ta cũng có thể mô hình bạn đọc rid mượn cuốn sách bid bằng cách cập nhật quan hệ $onLoanTo$ như sau

$$onLoanTo \Leftarrow \{ bid \mapsto rid \}.$$

Hàm tổng thể là hàm số mà gán giá trị cho mỗi phần tử trên miền xác định

$$S \rightarrow T = \{ f \mid f \in S \rightarrow T \wedge \text{dom}(f) = S \}$$

Hàm đơn trị là hàm số mà quan hệ ngược của nó cũng là hàm số

$$S \leftrightarrow T = \{ f \mid f \in S \rightarrow T \wedge f^{-1} \in T \rightarrow S \}$$

Hàm toàn ánh là hàm mà ánh xạ từ một tập nào đó đến mọi phần tử của miền giá trị.

$$S \twoheadrightarrow T = \{ f \mid f \in S \rightarrow T \wedge \text{ran}(f) = S \}$$

Chúng cũng có thể được kết hợp lại như sau

$$S \rightarrowtail T = (S \rightarrow T) \cap (S \leftrightarrow T)$$

$$S \rightarrowtail T = (S \rightarrow T) \cap (S \twoheadrightarrow T)$$

$$S \rightarrowtail T = (S \twoheadrightarrow T) \cap (S \rightarrow T)$$

Các dãy thường được sử dụng để biểu diễn dữ liệu mà được sắp thứ tự, chẳng hạn như danh sách, ngăn xếp, hàng đợi.

$$\text{seq } X = \{ s \mid s \in \mathbb{N} \rightarrow X \wedge \text{dom } s = 1 \dots \text{card}(s) \}$$

Chúng ta sử dụng $n_1 \dots n_2$, trong đó n_1, n_2 là các số nguyên, để ký hiệu tập các số nguyên nằm giữa hai số nguyên đó, kể cả hai đầu mút.

$$\{ n \mid n \in \mathbb{N} \wedge n_1 \leq n \wedge n \leq n_2 \}$$

Như vậy, 1...4 là viết tắt của tập {1, 2, 3, 4}

Ví dụ 5.7:

$$\text{seq}\{a, b\} = \{\{\}, \{1 \mapsto a\}, \{1 \mapsto b\}, \\ \{1 \mapsto a, 2 \mapsto a\}, \{1 \mapsto a, 2 \mapsto b\}, \dots\}$$

Dãy $\{1 \rightarrow a, 2 \rightarrow b\}$ được viết tắt là [a, b].

Dãy rỗng được ký hiệu là [].

Phép nối hai dãy được biểu diễn bởi \cap

Ví dụ 5.8:

$$[a, b] \cap [b, c] = [a, b, b, c]$$

Nếu S là một danh sách, thì $\text{first}(S)$ là phần tử đầu của danh sách và $\text{tail}(S)$ là danh sách còn lại từ S nếu bỏ đi phần tử đầu.

Như vậy đối với mọi danh sách S khác rỗng

$$S = [\text{first}(S)] \cap \text{tail}(S)$$

5.2 Phương pháp hình thức thiết kế phần mềm

Các khía cạnh của phương pháp thiết kế B-method:

- Ký hiệu để đặc tả phần mềm
- Kỹ thuật để thiết kế phần mềm
- Công cụ hỗ trợ phát triển phần mềm

B method là kết quả của các công trình của Jean-Raymond Abrial, người phát triển nó với đội ngũ nghiên cứu ở Sunbury vào cuối những năm 1980. Ông ta cũng là một trong những người sáng tạo ra ngôn ngữ đặc tả Z.

Để đọc đầy đủ về B method và lý thuyết của nó, có thể sử dụng cuốn sách của Abrial: The B method (nhà xuất bản Cambridge University Press).

B method dùng để đặc tả, thiết kế và lập trình các hệ thống phần mềm.

Các đặc trưng chính gồm:

- Abstract Machine, dữ liệu và các thao tác
- Đặc tả dữ liệu, đặc tả các thao tác
- Tiếp tục làm mịn để cài đặt, các kỹ thuật làm mịn
- Thư viện, tái sử dụng, sinh code
- Chứng minh

- B-toolkit

Nội dung

- Đặc tả phần mềm dùng các ký hiệu và Abstract machine
 - Tin tưởng vào tính đúng đắn của máy
 - Xây dựng máy lớn từ các máy nhỏ
 - Tạo nên phần mềm từ các đặc tả của nó
 - Sử dụng khái niệm làm mịn để liên kết một máy với cài đặt của nó
- **Pro B:** hỗ trợ và kiểm tra mô hình cho B.

Khái niệm máy trùu tượng

Một Abstract Machine bao gồm:

1. Tên máy (có thể với các tham số) được viết bởi mệnh đề MACHINE
2. State – trạng thái
 - VARIABLE: danh sách các tên ký hiệu các thành phần của trạng thái
 - INVARIANT: mệnh đề logic tạo nên các qui tắc tĩnh của máy
 - INITIALISATION: tiền chương trình chỉ ra máy được khởi tạo như thế nào
3. Các thao tác
 - OPERATIONS: Danh sách các thao tác, mà đối với mỗi thao tác, định nghĩa đầu vào, đầu ra, tác động đến các biến trạng thái.

Ví dụ 5.9 Đặc tả và thiết kế ngăn xếp

```
MACHINE Stack ( max , CHAR )
CONSTRAINTS max ∈ N1
VARIABLES stack
INVARIANT
    stack ∈ seq ( CHAR ) ∧ size ( stack ) ≤ max
INITIALISATION stack := [ ]

OPERATIONS
push ( nn ) ≡
    PRE size ( stack ) < max ∧ nn ∈ CHAR
    THEN stack := nn → stack
    END ;
pop ≡
    PRE size ( stack ) > 0
    THEN stack := tail ( stack )
    END ;
    nn ← top ≡
        PRE size ( stack ) > 0
        THEN nn := first ( stack )
        END

END
```

5.3 Máy bán hàng

Sau đây chúng ta sẽ đặc tả một Hệ thống hóa đơn cho khách hàng trong môi trường thương mại.

Mô tả hệ thống hóa đơn

1. Một khách hàng (client) được lưu lại trong hệ thống cùng với loại khách (category) của anh ta và cũng với quyền được chi tối đa cho phép. Đối với mỗi loại khách ở đây tương ứng có giảm giá áp dụng cho các hóa đơn tương ứng.
2. Một sản phẩm *Product* được lưu lại cùng với giá của nó, trạng thái của nó (sẵn sàng hay đã bán hết) và các sản phẩm thay thế có thể, mà là sản phẩm khác mà đảm bảo bán hết.
3. Một hóa đơn liên quan đến khách hàng, mà xuất cho người đó. Hóa đơn cũng có giảm giá được áp dụng cho tổng giá trị của hóa đơn. Cuối cùng nó được đặc trưng bởi tổng tối đa số tiền mà được chi cho nó. Hai thuộc tính cuối được lấy từ các thuộc tính tương tự của khách hàng.
4. Mỗi dòng của hóa đơn tương ứng với một sản phẩm nào đó, số lượng và giá đơn vị của sản phẩm đó. Thuộc tính cuối được lấy từ thuộc tính giá của sản phẩm.

Các thao tác của Hệ thống hóa đơn

1. Tạo và sửa khách hàng - *client*
2. Tạo và sửa sản phẩm – *product*
3. Tạo và xóa hóa đơn – *Invoice*
4. Thêm dòng mới vào hóa đơn

Các yêu cầu không hình thức

- R1. Mọi sản phẩm đã bán hết không là một phần trong hóa đơn
- R2. Nếu có tồn tại sản phẩm thay thế cho sản phẩm đã bán hết, thì hệ thống cần thay thế nó trong hóa đơn sản phẩm đó bởi sản phẩm thay thế.
- R3. Không có hai dòng khác nhau trong cùng một hóa đơn có thể tương ứng với cùng một sản phẩm.
- R4. Không có hóa đơn nào được tạo cho khách hàng loại *dubious*.
- R5. Tổng tiền đã được giảm giá cho một hóa đơn không được lớn hơn số tiền tối đa cho phép chi trả cho hóa đơn đó
- R6. Loại khách hàng *Friend* được giảm 20% giá, trong khi các loại khách khác không được giảm giá.

Xử lý lỗi

Hệ thống cần tạo ra thông báo lỗi khi nhập một sản phẩm mới vào hóa đơn, mà:

1. Sản phẩm đó có thể đã được bán hết và có thể không có sản phẩm thay thế (R1, R2)
2. Sản phẩm hoặc thay thế của nó có thể đã có trong hóa đơn, và không thể có nhiều hơn một dòng ứng với một sản phẩm trong hóa đơn (R2, R3).
3. Số tiền tối đa cho phép trong hóa đơn có thể đạt được khi nhập sản phẩm mới (R2, R5, R6).

Máy client

Máy client đóng gói client và đưa vào hai tập với một hằng số sau:

1. CLIENT là tập dành riêng ký hiệu mọi khách hàng có thể
2. CATEGORY dành cho các loại khách hàng khác nhau.
3. *discount* là hàm liên kết mỗi loại khách hàng với số phần trăm tương ứng được giảm giá cho tổng giá trị trong mọi hóa đơn của khách hàng

State of the Client Machine

MACHINE

Client

SETS

CLIENT;

CATEGORY =

{friend, dubious, normal}

CONSTANTS

discount

PROPERTIES

discount ∈ *CATEGORY* → (0..100) ∧

discount = {friend ↦ 80,
dubious ↦ 100,
normal ↦ 100}

VARIABLES

client, category, allowance

INVARIANT

client ⊆ *CLIENT* ∧
category ∈ *client* → *CATEGORY* ∧
allowance ∈ *client* → *NAT*

INITIALISATION

client, category, allowance := ∅, ∅, ∅

The *create_client* Operation

create_client has an input parameter *a* : *NAT* representing the allowance given to the newly registered client, who is given *normal* as his category.

c ← *create_client(a)* ≡

PRE
a ∈ *NAT* ∧ *client* ≠ *CLIENT*

THEN

ANY

cc

WHERE

cc ∈ *CLIENT* – *client*

THEN

client := *client* ∪ {*cc*} || *category(cc)* := *normal* ||
allowance(cc) := *a* || *c* := *cc*

END

END

More Operations

```

 $c \leftarrow \text{read\_client} \triangleq$ 
PRE
 $\text{client} \neq \emptyset$ 
THEN
 $c : \in \text{client}$ 
END

```

```

 $\text{modify\_category}(c, k) \triangleq$ 
PRE
 $c \in \text{client} \wedge$ 
 $k \in \text{CATEGORY}$ 
THEN
 $\text{category}(c) := k$ 
END

 $\text{modify\_allowance}(c, a) \triangleq \dots$ 

```

The Product Machine

The *Product* machine is used to encapsulate the product, and introduce its attributes.

MACHINE

Product

SETS

PRODUCT;

STATUS = {available,
sold_out}

VARIABLES

product, *price*,
status, *substitute*

INVARIANT

$\text{product} \subseteq \text{PRODUCT} \wedge$
 $\text{price} \in \text{product} \rightarrow \text{NAT} \wedge$
 $\text{status} \in \text{product} \rightarrow \text{STATUS} \wedge$
 $\text{substitute} \in$
 $\text{product} \leftrightarrow \text{status}^{-1}[\{\text{available}\}]$

INITIALISATION

$\text{product}, \text{price}, \text{status}, \text{substitute} :=$
 $\emptyset, \emptyset, \emptyset, \emptyset$

The *create_product* Operation

Function: to create a product. Parameter c is supposed to denote the price of the product. The status of the new product is supposed to be *available*.

```
p ←— create_product(c) ≡  
PRE  
    c ∈ NAT ∧ product ≠ PRODUCT  
THEN  
    ANY  
        pp  
    WHERE  
        pp ∈ PRODUCT – product  
    THEN  
        price(pp) := c || status(pp) := available ||  
        product := product ∪ {pp} || p := pp  
    END  
END
```

More Operations

```
assign_substitute(p, q) ≡  
PRE  
    p ∈ product ∧  
    q ∈ product ∧  
    status(q) = available  
THEN  
    substitute(p) := q  
END  
  
make_unavailable(p) ≡  
PRE  
    p ∈ product  
THEN  
    status(p) := sold_out ||  
    substitute := substitute ▷ {p}  
END  
  
modify_price(p, c) ≡ ...  
p ←— read_product ≡
```

The *Invoice* Machine

The *Invoice* machine

- uses *Client* and *Product* (because it needs to access some of the variables of these machines).
- needs two sets *INVOICE* and *LINE* for denoting all the possible invoices and lines.

The *Invoice* Machine (con'd)

MACHINE

Invoice

USES

Client, Product

SETS

*INVOICE ;
LINE*

VARIABLES

*invoice, customer, percentage,
allowed, total, line, origin,
article, quantity, unit_cost*

INVARIANT

$invoice \subseteq INVOICE \wedge$
 $customer \in invoice \rightarrow client \wedge$
 $percentage \in invoice \rightarrow (0..100) \wedge$
 $allowed \in invoice \rightarrow NAT \wedge$
 $total \in invoice \rightarrow NAT \wedge$
 $\text{ran}(total \otimes allowed) \subseteq \text{leq} \wedge$
 $line \subseteq LINE \wedge origin \in line \rightarrow invoice \wedge$
 $article \in line \rightarrow product \wedge$
 $quantity \in line \rightarrow NAT \wedge$
 $unit_cost \in line \rightarrow NAT \wedge$
 $origin \otimes article \in line \nrightarrow invoice \times product$

where $(f \otimes g)(x) \triangleq (f(x), g(x))$, all the variables are set to \emptyset initially,
and \nrightarrow denotes partial injective mapping

Creating an Invoice

This operation is designed to create an invoice for a given non-dubious client ((R4)).

```
inv ← create_invoice_header(c) ≡  
PRE  
    c ∈ client ∧ category(c) ≠ dubious ∧ invoice ≠ INVOICE  
THEN  
    ANY j  
    WHERE  
        j ∈ INVOICE – invoice  
    THEN  
        invoice := invoice ∪ {j} || customer(j) := c ||  
        percentage(j) := discount(category(c)) ||  
        allowed(j) := allowance(c) || inv := j  
    END  
END
```

Adding a Line

It adds a line to an invoice when the product has not appeared yet.

```
l ← new_line(i, p) ≡  
PRE  
    i ∈ invoice ∧ p ∈ product ∧ status(p) = available ∧  
    (i, p) ∉ ran(origin ⊗ article) ∧  
    line ≠ LINE  
THEN  
    ANY m  
    WHERE m ∈ LINE – line  
    THEN  
        l := m || line := line ∪ {m} ||  
        origin(m) := i || article(m) := p ||  
        quantity(m) := 0 || unit_cost(m) := price(p)  
    END  
END
```

Obtaining a Line

The operation is to obtain the line of an available product, which is supposed to appear already in a certain line of the given invoice.

```
 $l \leftarrow \text{the\_line}(i, p) \triangleq$ 
PRE
 $i \in \text{invoice} \wedge$ 
 $p \in \text{product} \wedge$ 
 $\text{status}(p) = \text{available} \wedge$ 
 $(i, p) \in \text{ran}(\text{origin} \otimes \text{article})$ 
THEN
 $l := (\text{origin} \otimes \text{article})^{-1}(i, p)$ 
END
```

Incrementing a Line

The operation is for incrementing a line, corresponding to an available product, with an extra quantity.

```
 $\text{increment}(l, q) \triangleq$ 
PRE
 $l \in \text{line} \wedge$ 
 $q \in \text{NAT} \wedge$ 
 $\text{status}(\text{article}(l)) = \text{available} \wedge$ 
 $\text{quantity}(l) + q \in \text{NAT} \wedge$ 
 $\text{total}(\text{origin}(l)) +$ 
 $(q \times \text{unit\_cost}(l) \times \text{percentage}(\text{origin}(l))/100)$ 
 $\leq \text{allowed}(\text{origin}(l))$ 
THEN
 $\text{quantity}(l) := \text{quantity}(l) + q \parallel$ 
 $\text{total}(\text{origin}(l)) := \text{total}(\text{origin}(l)) +$ 
 $(q \times \text{unit\_cost}(l) \times \text{percentage}(\text{origin}(l))/100)$ 
```

Removing All Lines

$\text{remove_all_lines}(i) \triangleq$

PRE

$i \in \text{invoice}$

THEN

$\text{line} := \text{line} - \text{origin}^{-1}[\{i\}] \parallel$
 $\text{origin} := (\text{origin}^{-1}[\{i\}] \triangleleft \text{origin}) \parallel$
 $\text{article} := (\text{origin}^{-1}[\{i\}] \triangleleft \text{article}) \parallel$
 $\text{quantity} := (\text{origin}^{-1}[\{i\}] \triangleleft \text{quantity}) \parallel$
 $\text{unit_cost} := (\text{origin}^{-1}[\{i\}] \triangleleft \text{unit_cost})$

where for a function f and a set S we define

$$\begin{aligned}\text{dom}(S \triangleleft f) &\triangleq \text{dom}(f) - S \\ (S \triangleleft f)(x) &\triangleq f(x)\end{aligned}$$

Removing an Invoice

The operation $\text{remove_invoice_header}$ is used to remove an invoice header.

$\text{remove_invoice_header}(i) \triangleq$

PRE

$i \in \text{invoice} \wedge$
 $i \notin \text{ran}(\text{origin})$

THEN

$\text{invoice} := \text{invoice} - \{i\} \parallel$
 $\text{customer} := \{i\} \triangleleft \text{customer} \parallel$
 $\text{percentage} := \{i\} \triangleleft \text{percentage} \parallel$
 $\text{allowed} := \{i\} \triangleleft \text{allowed} \parallel$
 $\text{total} := \{i\} \triangleleft \text{total}$

END

An invoice can be deleted after removal of all its lines using the remove_all_lines operation, i.e. $i \notin \text{ran}(\text{origin})$.

The *Invoice-System* Machine

The *Invoice-System* machine is just the joining together of all the previous machines. We add a number of operations to help in defining good error reporting.

MACHINE
 Invoice-System
EXTENDS
 Client, Product, Invoice

OPERATIONS

```
b ← some_client_exists ≡ ...;  
b ← clients_not_saturated ≡ ... ;  
b ← client_not_dubious(c) ≡ ... ;  
b ← some_product_exists ≡ ... ;  
b ← products_not_saturated ≡ ... ;  
b ← product_avaliable(p) ≡ ... ;  
b ← product_has_substitute(p) ≡ ... ;  
b ← invoices_not_saturated ≡ ... ;  
b ← new_product_in_invoice(i, p) ≡ ...
```

END

5.4 Hệ thống đa thang máy

Một hệ thống thang máy được khởi tạo trong một tòa nhà m tầng. Thiết kế logic để di chuyển các thang máy giữa các tầng tương ứng với các qui tắc sau:

R1. Mỗi thang có một tập các nút nhấn, mỗi nút nhấn cho một tầng. Các nút này sáng khi nhấn và buộc thang máy chờ ở tầng tương ứng. Nút này sẽ tắt sáng khi thang đến tầng đó.

R2. Mỗi tầng (trừ tầng thấp nhất và tầng trên cùng) có hai nút nhấn, một nút để yêu cầu đi lên và một nút để yêu cầu đi xuống. Chúng có thể sáng, khi nhấn. Các nút này sẽ tắt khi có một thang máy đi đến tầng đó theo đúng hướng mong muốn hoặc có thang máy đến tầng đó nhưng không có yêu cầu tiếp theo nào bên trong. Trong trường hợp sau, nếu cả hai nút đều sáng, thì chỉ có một nút tắt đi.

R3. Khi một thang máy không có yêu cầu dịch vụ, nó cần ở lại bên cuối cùng của nó với cửa được đóng và chờ yêu cầu tiếp theo.

R4. Mọi yêu cầu thang máy từ tầng cần phải cuối cùng được phục vụ

R5. Mọi yêu cầu thang máy từ bên trong thang máy cần phải cuối cùng được phục vụ

R6. Mỗi thang máy có nút khẩn cấp, mà khi nhấn, buộc gửi cảnh báo đến quản trị thang. Thang máy đó sau đó được cho rằng sẽ không cung cấp dịch vụ nữa. Mỗi thang máy có cơ chế tắt trạng thái không phục vụ.

The Lift Machine

We introduce two sets *LIFT*, which is a deferred set, and the enumerated set *DIRECTION* in to the *Lift* machine. We also define two constants, *top* and *ground*, yielding the top and ground floors.

```
MACHINE
  Lift
SETS
  LIFT ;
  DIRECTION = {up, dn}
CONSTANTS
  ground, top
PROPERTIES
  ground ∈ NAT ∧ top ∈ NAT ∧ top > ground
DEFINITIONS
  FLOOR ≡ ground..top
```

Các biến

1. Biến *moving* được sử dụng để chỉ ra tập các thang máy đang chuyển động.
2. Với mỗi thang máy l , ta ký hiệu tầng tương ứng là: $floor(l)$, là tầng mà được giả thiết ở đó thang máy đang dừng, nếu l không chuyển động, hoặc tầng gần nhất mà nó sắp đến.
3. Với mỗi thang máy l , ta ký hiệu hướng $dir(l)$ được giả thiết là hướng theo đó l đang chuyển động, nếu nó là moving hoặc theo hướng của tầng mà nó định đến tiếp theo, nếu nó không chuyển động.
4. Biến *in* là quan hệ hai ngôi từ *FLOOR* vào *DIRECTION*. Khi $(f, d) \in in$, nó có nghĩa là có ai đó muốn di chuyển đến tầng f theo hướng d .
5. Biến *out* là quan hệ hai ngôi từ *LIFT* vào *FLOOR*. Khi $(l, f) \in out$, nó có nghĩa là có ai đó đang ở trong thang máy l muốn ra khỏi thang máy l ở tầng f .

Variables (Cont'd)

VARIABLES

moving, floor, dir, in, out

INVARIANT

$$\begin{aligned} & moving \subseteq LIFT \wedge \\ & floor \in LIFT \rightarrow FLOOR \wedge \\ & dir \in LIFT \rightarrow DIRECTION \wedge \\ & in \in FLOOR \leftrightarrow DIRECTION \wedge \\ & out \in LIFT \leftrightarrow FLOOR \wedge \\ & (ground \leftrightarrow dn) \notin in \wedge \\ & (top \leftrightarrow up) \notin in \wedge \\ & moving \triangleleft (out \cap floor) = \emptyset \wedge \\ & in \cap ran(moving \triangleleft (floor \otimes dir)) = \emptyset \end{aligned}$$

INITIALISATION

$$\begin{aligned} & in, out, moving := \emptyset, \emptyset, \emptyset \parallel \\ & floor, dir := \\ & LIFT \times \{ground\}, LIFT \times \{up\} \end{aligned}$$

Request a Floor

This operation is used to correspond to the event of pressing buttons to request a floor inside a lift.

Request_Floor(l, f) $\hat{=}$
PRE
 $l \in LIFT \wedge$
 $f \in FLOOR \wedge$
 $(l \notin moving \Rightarrow (floor(l) \neq f))$
THEN
 $out := out \cup \{l \mapsto f\}$
END

Request a Lift

This operation corresponds to actions of pressing a button to request a lift on a floor.

Request_Lift(f, d) ≡

PRE

$f \in FLOOR \wedge$
 $d \in DIRECTION \wedge$
 $(f, d) \neq (ground, dn) \wedge$
 $(f, d) \neq (top, up) \wedge$
 $(f, d) \notin \text{ran}(\text{moving} \Leftarrow (floor \otimes dir))$

THEN

$in := in \cup \{f \mapsto d\}$

END

Các sự kiện điều khiển

Bây giờ chúng ta định nghĩa các sự kiện mà ở đó hệ thống quyết định cho thang máy đang chuyển động, mà đến gần một tầng nào đó, sẽ tiếp tục chuyển động hay là dừng ở tầng đó. Mệnh đề *attracted_up(l)* là đúng, khi mà thang máy l đang ở hoặc gần đến *floor(l)*, và ít nhất một trong các điều kiện sau được thỏa mãn

- Một ai đó bên trong thang máy l đã thể hiện ý của họ là ra khỏi thang ở một tầng trên tầng *floor(l)* theo hướng lên.

$$\text{out}[\{l\}] \cap ((\text{floor}(l) + 1) \dots \text{top}) \neq \emptyset$$

- Có một ai đó đang chờ thang máy ở một tầng trên *floor(l)* theo hướng lên của l.

$$\text{dom}(in) \cap ((\text{floor}(l) + 1) \dots \text{top}) \neq \emptyset$$

The DEFINITIONS Clause

Predicate *attracted_dn(l)* is defined in a similar way.

DEFINITIONS

$$\begin{aligned} \text{attracted_up}(l) &\equiv \\ &(\text{dom}(\text{in}) \cup \text{out}[\{l\}]) \cap ((\text{floor}(l) + 1) \dots \text{top}) \neq \emptyset; \\ \text{attracted_dn}(l) &\equiv \\ &(\text{dom}(\text{in}) \cup \text{out}[\{l\}]) \cap (\text{ground} \dots (\text{floor}(l) - 1)) \neq \emptyset \end{aligned}$$

Điều khiển chuyển động

can_continue_up(l) và *can_continue_down(l)* được giả thiết là đúng khi một thang máy đang chuyển động l không có lý do nào dừng ở tầng *floor(l)* mà nó sắp đến. Rõ ràng điều đó xảy ra khi ba điều kiện sau đồng thời thỏa mãn:

1. Không ai muốn ra từ thang máy l ở tầng *floor(l)*.

$$(l \mapsto \text{floor}(l)) \notin \text{out}$$

2. Không ai muốn vào thang máy từ tầng *floor(l)* theo hướng *dir(l)*.

$$(\text{floor}(l) \mapsto \text{dir}(l)) \notin \text{in}$$

3. Thang máy l là vẫn được lôi kéo đi theo hướng *dir(l)*

Control of Movement

can_continue_up(l) and *can_continue_dn(l)* are supposed to hold when a moving lift *l* has no reason to stop at *floor(l)*, where it is about to arrive. Obviously, this is when the following three conditions hold *simultaneously*:

(1) nobody wants to get out from *l* at *floor(l)*

$$(l \mapsto floor(l)) \notin out$$

(2) nobody wants to get in from *floor(l)* to travel in *dir(l)*

$$(floor(l) \mapsto dir(l)) \notin in$$

(3) lift *l* is still attracted in the *dir(l)* direction.

Continue to Move Up

$$\begin{aligned} can_continue_up(l) \triangleq \\ (l \mapsto floor(l)) \notin out \wedge \\ (floor(l) \mapsto dir(l)) \notin in \wedge \\ attracted_up(l); \end{aligned}$$

$$\begin{aligned} can_continue_dn(l) \triangleq \\ (l \mapsto floor(l)) \notin out \wedge \\ (floor(l) \mapsto dir(l)) \notin in \wedge \\ attracted_dn(l); \end{aligned}$$

Lift Control Operations

Continue_up(l) $\hat{=}$
PRE
 $l \in moving \wedge$
 $dir(l) = up \wedge$
 $can_continue_up(l)$
THEN
 $floor(l) := floor(l) + 1$
END ;

Stop_up(l) $\hat{=}$
PRE
 $l \in moving \wedge$
 $dir(l) = up \wedge$
 $\neg can_continue_up(l)$
THEN
 $moving := moving - \{l\} \parallel$
 $out := out - \{l \leftrightarrow floor(l)\} \parallel$
 $in := in - \{floor(l) \leftrightarrow dir(l)\}$
END

Departure Operations

The decision for the departure of a lift from a floor in a certain direction or for the change of direction of a lift: The key idea is that a lift gives the priority to continuing its travel in the direction it was travelling when it stopped at the floor. If the lift has no reason to travel in the same direction then it is free to change to the opposite direction.

Depart_up(l) $\hat{=}$
PRE
 $l \in LIFT - moving \wedge$
 $dir(l) = up \wedge$
 $attracted_up(l)$
THEN
 $moving := moving \cup \{l\} \parallel$
 $floor(l) := floor(l) + 1$
END

Change_up_to_dn(l) $\hat{=}$
PRE
 $l \in LIFT - moving \wedge$
 $dir(l) = up \wedge$
 $\neg attracted_up(l) \wedge$
 $attracted_dn(l)$
THEN
 $in := in - \{floor(l) \leftrightarrow dn\} \parallel$
 $dir(l) := dn$
END

CHƯƠNG 6: MẪU THIẾT KẾ

6.1 Mở đầu

6.1.1 Mẫu thiết kế là gì

Như người phát triển phần mềm chúng ta có thể nghĩ rằng code của chúng ta chứa mọi lợi ích được cung cấp bởi ngôn ngữ lập trình hướng đối tượng. Code mà chúng ta viết là mềm dẻo đến mức mà chúng ta có thể thay đổi tùy ý mà không làm tổn hại nhiều. Code của chúng ta là có thể tái sử dụng, như vậy chúng ta có thể tái sử dụng nó ở bất cứ nơi nào mà không phải lo lắng. Chúng ta có thể bảo trì code của chúng ta dễ dàng và mọi thay đổi đến một phần của code sẽ không làm ảnh hưởng đến bất cứ phần nào khác của code đó.

Không may là những ưu điểm này không tự nó đến. Như người phát triển, đó là trách nhiệm của chúng ta thiết kế code theo cách sao cho nó cho phép code của chúng ta là mềm dẻo, dễ bảo trì và tái sử dụng được.

Thiết kế là nghệ thuật và nó đến với kinh nghiệm. Nhưng có một tập các giải pháp đã được viết bởi một số nhà phát triển tiên tiến có kinh nghiệm, mà đã đổi mới và giải quyết các vấn đề thiết kế tương tự. Các vấn đề này được biết đến như Mẫu thiết kế.

Mẫu thiết kế là kinh nghiệm trong thiết kế mã hướng đối tượng.

Mẫu thiết kế là các giải pháp sử dụng tổng quan cho các vấn đề xuất hiện trong hoàn cảnh giống nhau. Đây là các kinh nghiệm thực tế tốt nhất được sử dụng bởi các nhà phát triển tiên tiến. Mẫu không phải là các code hoàn chỉnh, nhưng có thể được sử dụng như các bản nháp mà được áp dụng giải quyết vấn đề. Mẫu là tái sử dụng, chúng có thể được áp dụng cho bài toán thiết kế tương tự không liên quan đến lĩnh vực nào. Nói cách khác, chúng ta có thể nghĩ mẫu như tài liệu hình thức mà chứa bài toán thiết kế lặp lại và giải pháp của nó. Một mẫu được sử dụng trong một bối cảnh thực tế có thể được tái sử dụng trong bối cảnh khác.

Nói chung một mẫu có bốn thành phần cốt yếu sau:

- **Tên mẫu:** được sử dụng để cung cấp tên duy nhất và có ý nghĩa cho mẫu mà định nghĩa bài toán thiết kế và giải pháp cho nó. Đặt tên cho mẫu thiết kế giúp cho nó được tham chiếu đến các mẫu khác được dễ dàng. Làm cho việc cung cấp tài liệu được dễ dàng và từ vựng đúng làm cho dễ dàng nghĩ về mẫu thiết kế đó.
- **Bài toán mô tả khi nào áp dụng mẫu.** Nó giải thích vấn đề và bối cảnh của nó. Nó có thể mô tả bài toán thiết kế chuyên biệt như làm sao biểu diễn thuật toán như đối tượng. Nó có thể mô tả một lớp hoặc cấu trúc đối tượng mà có dấu hiệu của thiết kế không mềm dẻo. Đôi khi bài toán sẽ bao gồm một danh sách các điều kiện mà cần thỏa mãn trước khi áp dụng mẫu.
- **Lời giải** mô tả các thành phần mà tạo nên thiết kế, quan hệ, nhiệm vụ và tương tác của chúng. Giải pháp không là code đầy đủ, nhưng làm việc như bản nháp mà có thể được thực hiện với code đó. Thay vào đó, mẫu cung cấp mô tả tổng quát về bài toán thiết kế

và làm sao sắp xếp các thành phần (các lớp và các đối tượng trong trường hợp của chúng ta) để giải quyết vấn đề.

- **Kết quả và hệ quả của việc áp dụng mẫu.** Hệ quả đối với phần mềm thường liên quan đến việc cân bằng thời gian và không gian. Chúng có thể đụng chạm đến ngôn ngữ và chí phí cài đặt. Vì việc tái sử dụng thường là yếu tố trong thiết kế hướng đối tượng, nên các hệ quả của một mẫu bao gồm tác động của nó đến tính mềm dẻo, mở rộng hoặc chuyển mang của hệ thống. Liệt kê các hệ quả này giúp cho chúng ta hiểu và đánh giá tầm quan trọng của chúng.

6.1.2 Tại sao phải sử dụng Mẫu thiết kế

- **Tính mềm dẻo:** sử dụng mẫu thiết kế code của bạn trở nên mềm dẻo. Nó giúp cung cấp mức độ khái quát đúng theo đó các đối tượng trở nên gắn kết lỏng lẻo với nhau làm cho code của bạn dễ thay đổi.
- **Tính tái sử dụng:** Các đối tượng và các lớp gắn kết lỏng lẻo làm cho code của bạn dễ tái sử dụng hơn. Kiểu code này trở nên dễ dàng kiểm thử so sánh với kiểu code gắn kết chặt chẽ.
- **Từ vựng chia sẻ:** Từ vựng chia sẻ làm cho dễ dàng chia sẻ code và suy nghĩ của bạn với các thành viên khác của đội. Nó làm tăng hiểu biết lẫn nhau giữa các thành viên liên quan đến code.
- **Nắm bắt thực tiễn tốt nhất:** Thiết kế mẫu nắm bắt giải pháp mà đã thành công áp dụng cho bài toán. Bằng cách học các mẫu này và vấn đề liên quan, người phát triển chưa có kinh nghiệm học được nhiều điều về thiết kế phần mềm.

Mẫu thiết kế làm cho dễ dàng tái sử dụng thiết kế và kiến trúc thành công.

Các kỹ thuật được chứng minh thực tiễn như mẫu thiết kế làm cho chúng được truy cập dễ hơn cho những người phát triển các hệ thống mới. Mẫu thiết kế giúp cho bạn chọn các lựa chọn thiết kế mà làm cho hệ thống tái sử dụng và tránh các lựa chọn mà tái sử dụng thỏa hiệp. Mẫu thiết kế có thể giúp soạn tài liệu và bảo trì hệ thống đang tồn tại bằng việc gắn kết với đặc tả tường minh tương tác lớp, các đối tượng và chủ đích ở bên dưới. Mẫu thiết kế giúp người thiết kế có được thiết kế đúng nhanh hơn.

6.1.3 Chọn và sử dụng một mẫu

Có một số mẫu thiết kế để chọn; để chọn một cái, bạn cần có kiến thức tốt về từng cái trong số đó. Có rất nhiều mẫu thiết kế mà mới nhìn qua, thấy chúng giống nhau. Chúng giải quyết gần giống nhau kiểu bài toán thiết kế và cũng có phần cài đặt tương tự. Bạn cần phải hiểu rất sâu về chúng để cài đặt mẫu thiết kế đúng cho bài toán thiết kế cụ thể.

Trước hết bạn cần nhận diện kiểu bài toán thiết kế mà bạn đang đối mặt. Một bài toán thiết kế cần được phân loại thành nhóm khởi tạo, nhóm cấu trúc hay nhóm hành vi. Dựa trên sự phân loại này bạn lọc các mẫu và lựa chọn cái phù hợp. Chẳng hạn

- Ở đây có nhiều khởi tạo như nhau của một lớp mà biểu diễn cùng một vật, một giá trị về các tính chất của các đối tượng, và chúng chỉ được sử dụng để đọc: bạn cần lựa chọn mẫu Singleton cho bài toán thiết kế đó, mà đảm bảo rằng chỉ có một khởi tạo duy nhất cho toàn bộ ứng dụng. Nó cũng giúp giảm kích thước bộ nhớ.
- Các lớp phụ thuộc quá nhiều vào nhau. Một thay đổi ở một lớp tác động đến mọi lớp phụ thuộc khác, bạn cần phải sử dụng Bridge, Mediator hoặc Command để giải quyết bài toán thiết kế này.
- Có hai giao diện không tương thích khác nhau trong hai phần khác nhau của code, và bạn cần chuyển đổi một giao diện sang giao diện khác mà được sử dụng bởi code của client để làm cho toàn bộ code vẫn làm việc: mẫu thiết kế Adapter đáp ứng yêu cầu bài toán thiết kế này.

Một mẫu thiết kế có thể được sử dụng để giải nhiều hơn một bài toán thiết kế, và một bài toán thiết kế có thể được giải quyết bởi nhiều hơn một mẫu thiết kế. Có thể có rất nhiều bài toán thiết kế và lời giải cho chúng, nhưng chọn mẫu thiết kế mà vừa khít là phụ thuộc vào kiến thức và hiểu biết của bạn về mẫu thiết kế. Nó cũng phụ thuộc vào code mà bạn đã có trong tay.

6.1.4 Phân loại các mẫu thiết kế

Các mẫu thiết kế được chia thành ba loại sau:

- Các mẫu khởi tạo
- Các mẫu cấu trúc
- Các mẫu hành vi

Các mẫu khởi tạo

Các mẫu thiết kế khởi tạo được sử dụng để thiết kế các tiến trình tạo các đối tượng. Một mẫu khởi tạo sử dụng để thừa để đa dạng việc tạo đối tượng.

Có hai chủ đề tái diễn trong các mẫu này. Trước hết, tất cả chúng đóng gói kiến thức về việc các lớp cụ thể nào hệ thống sử dụng. Thứ hai, chúng che giấu việc khởi tạo các lớp này được thực hiện và gộp với nhau như thế nào. Mọi việc mà hệ thống biết về các đối tượng là các giao diện của chúng được định nghĩa bởi các lớp trừu tượng. Kéo theo, các mẫu khởi tạo cho bạn nhiều sự mềm dẻo trong việc cái gì được tạo, ai tạo chúng, được tạo như thế nào và khi nào.

Có thể có một số trường hợp khi hai hay nhiều hơn, các mẫu trông như phù hợp cho lời giải của một bài toán thiết kế. Đôi khi, hai mẫu bù trừ nhau, chẳng hạn: mẫu Builder có thể được dùng với mẫu khác để cài đặt thành phần nào được xây dựng.

Các mẫu cấu trúc

Các mẫu cấu trúc liên quan đến việc các lớp và các đối tượng được kết hợp để tạo cấu trúc lớn hơn như thế nào. Các mẫu lớp cấu trúc được kế thừa để kết hợp tạo các giao diện hoặc cài đặt.

Một ví dụ đơn giản, xét đa kế thừa trộn hai hoặc nhiều lớp vào một. Kết quả là một lớp kết hợp các tính chất của các lớp cha của nó. Mẫu này đặc biệt hữu ích để tạo cho các thư viện lớp phát triển độc lập làm việc với nhau.

Ngoài việc kết hợp giao diện hoặc cài đặt, các mẫu cấu trúc mô tả cách kết hợp các đối tượng để hiện thực chức năng mới. Tính mềm dẻo bổ sung này của việc tích hợp đối tượng đến từ khả năng thay đổi việc tích hợp trong thời gian thực mà là không thể đổi với việc tích hợp lớp tĩnh.

Các mẫu hành vi

Các mẫu hành vi liên quan đến các thuật toán và việc gán trách nhiệm giữa các đối tượng. Các mẫu hành vi không chỉ mô tả các mẫu đối tượng hoặc các lớp mà còn các mẫu truyền thông giữa chúng. Các mẫu này khắc họa dòng điều khiển phức tạp mà là khó tuân thủ trong thời gian thực. Chúng làm cho bạn khỏi bận tâm vào dòng điều khiển, cho phép bạn tập trung chỉ vào cách các đối tượng kết nối với nhau.

Các mẫu hành vi sử dụng kết hợp đối tượng hơn là giao diện. Một số mô tả nhóm các đối tượng phối hợp để thực hiện nhiệm vụ mà không đối tượng riêng rẽ nào thực hiện được. Vấn đề quan trọng ở đây là làm sao các đối tượng ngang hàng đó biết về nhau. Các nút hàng ngang có thể duy trì tham chiếu tường minh đến nhau, nhưng điều đó có thể làm tăng sự gắn kết của chúng. Mẫu Mediator tránh làm điều này bằng việc đưa vào đối tượng trung gian mediator giữa các nút hàng ngang. Mediator cung cấp điều hướng cần thiết cho các đối tượng gắn kết lỏng lẻo.

Dưới đây là bảng chỉ rõ danh sách các mẫu thuộc các nhóm phân loại nào:

Creational Patterns	Structural Patterns	Behavioral Patterns
Abstract Factory	Adapter	Chain of Responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	FaÇade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template Method
		Visitor

6.2 Mẫu thiết kế ADAPTER

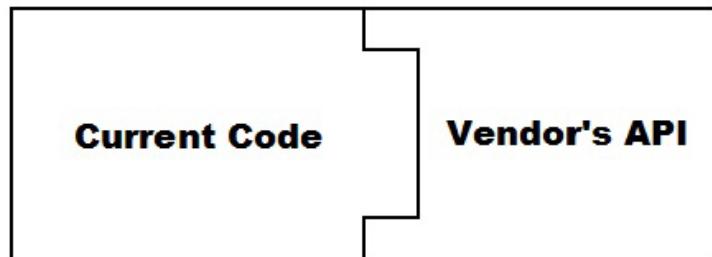
6.2.1 Mẫu thiết kế Adapter

Người phát triển phần mềm Max, làm việc xây dựng website thương mại điện tử. Website cho phép người sử dụng mua và trả tiền trực tuyến. Trang này được tích hợp với cổng trả tiền bên thứ ba, mà theo đó người sử dụng cần trả hóa đơn sử dụng thẻ credit card của họ. Mọi việc tiến triển tốt, cho đến khi quản lý dự án kêu anh ta thay đổi một chút.

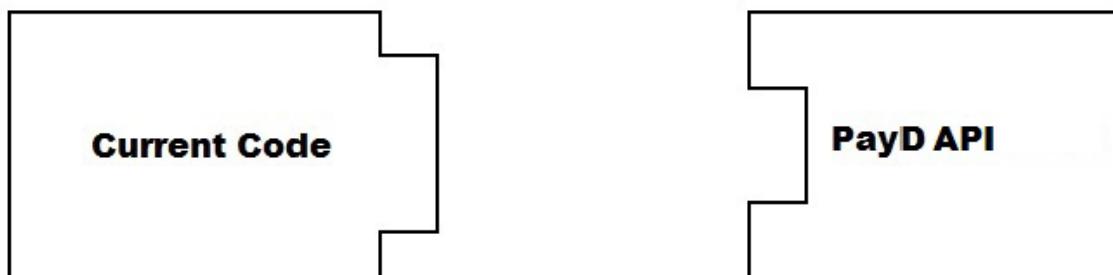
Quản lý dự án nói với anh ta là họ phải thay đổi nhà cung cấp cổng trả tiền, và họ cần phải cài đặt lại điều đó.

Vấn đề này sinh ở đây là trang đã được đính vào cổng trả tiền Xpay mà sử dụng đối tượng kiểu *Xpay*. Nhà cung cấp mới PayD, chỉ cho phép dùng các đối tượng kiểu *PayD* để xử lý. Max không muốn thay đổi toàn bộ tập khoảng 100 lớp đã tham chiếu đến các đối tượng kiểu *Xpay*. Nó cũng tiềm ẩn rủi ro cho Dự án, mà đã gần đưa vào sử dụng. Anh ta không thể thay đổi công cụ của cổng trả tiền bên thứ ba. Vấn đề xảy ra vì các giao diện không tương thích giữa hai phần code. Để tiến trình vẫn tiếp tục làm việc, Max cần phải tìm cách làm cho code tương thích với các API được cung cấp bởi cổng trả tiền mới

Code hiện tại với API của *Xpay*



Bây giờ giao diện của code hiện tại không tương thích với API của cổng trả tiền mới



6.2.2 Mẫu Adapter giải cứu

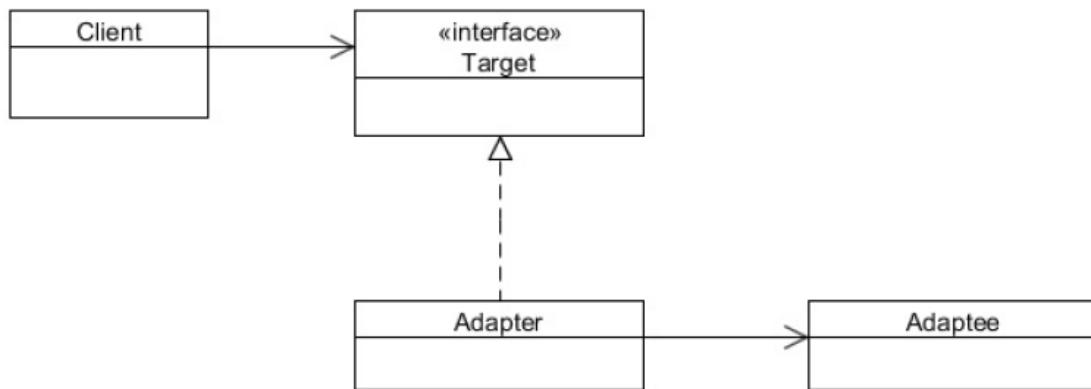
Cái mà Max cần ở đây là mẫu Adapter, mà có thể nằm giữa code và API của nhà cung cấp mới, và có thể cho phép tiến trình tiếp tục làm việc. Nhưng trước khi đưa ra giải pháp, chúng ta sẽ xem Adapter là gì và nó làm việc như thế nào.

Đôi khi, có kịch bản hai đối tượng không khớp nhau, nhưng chúng cần lắp ghép để hoàn thành công việc. Tình huống này có thể phát sinh khi chúng ta tìm cách tích hợp code đã có với code mới, hoặc khi thay đổi API của bên thứ ba. Nó xảy ra vì các giao diện không tương thích của hai đối tượng mà không khớp với nhau.

Mẫu Adapter là mẫu cấu trúc, nó cho phép bạn lắp một đối tượng hay một lớp đặt vào một đối tượng hay một lớp khác. Nó chuyển đổi giao diện của một lớp sang giao diện khác mà client mong đợi. Nó cho phép các lớp làm việc với nhau, mặc dù có các giao diện không tương thích. Nó cho phép xác lập giao diện giữa các đối tượng và các lớp mà không thay đổi trực tiếp các đối tượng và các lớp.

Bạn có thể nghĩ là Adapter giống như bộ chuyển ô trên thực tế mà được sử dụng để kết nối giữa hai thiết bị mà không nối trực tiếp được với nhau. Một Adapter nằm giữa các thiết bị này để cho phép luồng vào từ một thiết bị chảy sang thiết bị khác ở dạng mà nó muốn, nếu không sẽ không nối hai thiết bị đó với nhau được.

Một Adapter sử dụng kết hợp để lưu đối tượng mà nó hỗ trợ thích nghi, và khi phương thức của Adapter được triệu gọi, nó chuyển các lời gọi này thành những điều mà đối tượng được thích nghi có thể hiểu và truyền các lời gọi này đến các đối tượng được thích nghi. Code mà gọi Adapter không cần biết là nó không tương tác với kiểu đối tượng mà nó nghĩ, mà thay vào đó là các đối tượng được thích nghi.



Giải pháp cho bài toán

Giao diện Xpay được trông như sau:

```
package com.javacodegeeks.patterns.adapterpattern.xpay;

public interface Xpay {

    public String getCreditCardNo();
    public String getCustomerName();
    public String getCardExpMonth();
    public String getCardExpYear();
    public Short getCardCVVNo();
    public Double getAmount();

    public void setCreditCardNo(String creditCardNo);
    public void setCustomerName(String customerName);
    public void setCardExpMonth(String cardExpMonth);
    public void setCardExpYear(String cardExpYear);
    public void setCardCVVNo(Short cardCVVNo);
    public void setAmount(Double amount);

}
```

Nó chứa tập các phương thức *set* và *get* các thông tin về thẻ và tên chủ tài khoản. Giao diện Xpay này đã được cài đặt trong code mà được sử dụng để tạo đối tượng kiểu này và đưa đối tượng cho các API của công ty trả tiền.

Lớp sau đây xác định cài đặt cho giao diện Xpay:

```
package com.javacodegeeks.patterns.adapterpattern.site;

import com.javacodegeeks.patterns.adapterpattern.xpay.Xpay;

public class XpayImpl implements Xpay {

    private String creditCardNo;
    private String customerName;
    private String cardExpMonth;
    private String cardExpYear;
    private Short cardCVVNo;
    private Double amount;

    @Override
    public String getCreditCardNo() {
        return creditCardNo;
    }

    @Override
    public String getCustomerName() {
        return customerName;
    }

    @Override
    public String getCardExpMonth() {
        return cardExpMonth;
    }

    @Override
    public String getCardExpYear() {
        return cardExpYear;
    }
}
```

Nhưng Xpay đã được tạo bởi hầu hết các phần khác của code, thật sự là khó và rủi ro khi thay đổi toàn bộ tập các lớp này.

Chúng ta cần một cách khác, mà có thể thực hiện yêu cầu của nhà cung cấp mới để xử lý trả tiền mà cũng làm cho code hiện thời không hoặc thay đổi ít. Cách này được cung cấp bởi mẫu Adapter.

Chúng ta sẽ tạo ra Adapter mà sẽ là kiểu *PayD* và bao bọc đối tượng kiểu *Xpay*.

```
@Override  
public short getCardCVVNo() {  
    return cardCVVNo;  
}  
  
@Override  
public Double getAmount() {  
    return amount;  
}  
  
@Override  
public void setCreditCardNo(String creditCardNo) {  
    this.creditCardNo = creditCardNo;  
}  
  
@Override  
public void setCustomerName(String customerName) {  
    this.customerName = customerName;  
}  
  
@Override  
public void setCardExpMonth(String cardExpMonth) {  
    this.cardExpMonth = cardExpMonth;  
}  
  
@Override  
public void setCardExpYear(String cardExpYear) {  
    this.cardExpYear = cardExpYear;  
}  
  
@Override  
public void setCardCVVNo(Short cardCVVNo) {  
    this.cardCVVNo = cardCVVNo;  
}  
  
@Override  
public void setAmount(Double amount) {  
    this.amount = amount;  
}  
}
```

Giao diện của nhà cung cấp mới trông như sau:

```
package com.javacodegeeks.patterns.adapterpattern.payd;

public interface PayD {

    public String getCustCardNo();
    public String getCardOwnerName();
    public String getCardExpMonthDate();
    public Integer getCVVNo();
    public Double getTotalAmount();

    public void setCustCardNo(String custCardNo);
    public void setCardOwnerName(String cardOwnerName);
    public void setCardExpMonthDate(String cardExpMonthDate);
    public void setCVVNo(Integer CVVNo);
    public void setTotalAmount(Double totalAmount);
}
```

Như bạn đã thấy, giao diện này có tập phương thức khác mà cần được cài đặt trong code. Nhưng Xpay đã được tạo bởi hầu hết các phần khác của code, thật sự là khó và rủi ro khi thay đổi toàn bộ tập các lớp này.

Chúng ta cần một cách khác, mà có thể thực hiện yêu cầu của nhà cung cấp mới để xử lý trả tiền mà cũng làm cho code hiện thời không hoặc thay đổi ít. Cách này được cung cấp bởi mẫu Adapter.

Chúng ta sẽ tạo ra Adapter mà sẽ là kiểu *PayD* và bao bọc đối tượng kiểu *Xpay*.

```
package com.javacodegeeks.patterns.adapterpattern.site;

import com.javacodegeeks.patterns.adapterpattern.payd.PayD;
import com.javacodegeeks.patterns.adapterpattern.xpay.Xpay;

public class XpayToPayDAdapter implements PayD{

    private String custCardNo;
    private String cardOwnerName;
    private String cardExpMonthDate;
    private Integer cVVNo;
    private Double totalAmount;

    private final Xpay xpay;

    public XpayToPayDAdapter(Xpay xpay) {
        this.xpay = xpay;
        setProp();
    }

    @Override
    public String getCustCardNo() {
        return custCardNo;
    }

    @Override
    public String getCardOwnerName() {
        return cardOwnerName;
    }
}
```

```
@Override
public String getCardExpMonthDate() {
    return cardExpMonthDate;
}

@Override
public Integer getCVVNo() {
    return cVVNo;
}

@Override
public Double getTotalAmount() {
    return totalAmount;
}

@Override
public void setCustCardNo(String custCardNo) {
    this.custCardNo = custCardNo;
}

@Override
public void setCardOwnerName(String cardOwnerName) {
    this.cardOwnerName = cardOwnerName;
}

@Override
public void setCardExpMonthDate(String cardExpMonthDate) {
    this.cardExpMonthDate = cardExpMonthDate;
}

@Override
public void setCVVNo(Integer cVVNo) {
    this.cVVNo = cVVNo;
}

@Override
public void setTotalAmount(Double totalAmount) {
    this.totalAmount = totalAmount;
}

private void setProp(){
    setCardOwnerName(this.xpay.getCustomerName());
    setCustCardNo(this.xpay.getCreditCardNo());
    setCardExpMonthDate(this.xpay.getCardExpMonth() + "/" + this.xpay. ←
        getCardExpYear());
    setCVVNo(this.xpay.getCardCVVNo().intValue());
    setTotalAmount(this.xpay.getAmount());
}

}
```

Trong đoạn code trên đây, chúng ta tạo Adapter (*XpayToPayDAdapter*). Adapter cài đặt giao diện PayD, nó được yêu cầu mô phỏng đối tượng kiểu PayD. Adapter sử dụng kết hợp đối tượng để giữ đối tượng mà nó hỗ trợ thích nghi là đối tượng Xpay. Đối tượng này được truyền cho Adapter thông qua hàm tạo.

Bây giờ, lưu ý rằng ta có hai kiểu giao diện không tương thích, mà ta cần khớp với nhau sử dụng Adapter để làm cho code hoạt động. Các giao diện này có tập phương thức khác nhau.

Nhưng mục tiêu chính của các giao diện này rất giống nhau, tức là cung cấp thông tin khách hàng và thẻ tín dụng để chuyển cho cổng trả tiền.

Phương thức `setProp()` của lớp trên đây được sử dụng để thiết lập các tính chất của Xpay vào đối tượng PayD. Chúng ta thiết lập các phương thức mà làm việc tương tự nhau trong hai giao diện. Tuy nhiên, chỉ có phương thức duy nhất trong giao diện PayD để thiết lập tháng và năm của thẻ tín dụng, trái ngược với hai phương thức trong Xpay. Chúng ta hợp kết quả của hai phương thức của đối tượng Xpay

```
(this.xpay.getCardExpMonth() + "/" + this.xpay.getCardExpYear())
```

và thiết lập nó vào phương thức

```
setCardExpMonthDate()
```

Bây giờ chúng ta kiểm tra đoạn code trên giải quyết bài toán của Max như thế nào

```
package com.javacodegeeks.patterns.adapterpattern.site;

import com.javacodegeeks.patterns.adapterpattern.payd.PayD;
import com.javacodegeeks.patterns.adapterpattern.xpay.Xpay;

public class RunAdapterExample {

    public static void main(String[] args) {

        // Object for Xpay
        Xpay xpay = new XpayImpl();
        xpay.setCreditCardNo("4789565874102365");
        xpay.setCustomerName("Max Warner");
        xpay.setCardExpMonth("09");
        xpay.setCardExpYear("25");
        xpay.setCardCVVNo((short)235);
        xpay.setAmount(2565.23);

        PayD payD = new XpayToPayDAdapter(xpay);
        testPayD(payD);

    }

    private static void testPayD(PayD payD) {

        System.out.println(payD.getCardOwnerName());
        System.out.println(payD.getCustCardNo());
        System.out.println(payD.getCardExpMonthDate());
        System.out.println(payD.getCVVNo());
        System.out.println(payD.getTotalAmount());
    }
}
```

Trong lớp trên, trước hết chúng ta tạo đối tượng Xpay và thiết lập các tính chất của nó. Sau đó, ta tạo đối tượng adapter và truyền Xpay cho nó trong hàm tạo, và gán nó cho giao diện PayD. Phương thức tĩnh `TestPayD()` dùng đối tượng PayD như đối số mà chạy và in ra các phương thức để kiểm tra chạy. Kiểu được truyền cho phương thức `TestPayD()` là kiểu `PayD`, phương thức sẽ thực hiện không có bất cứ vấn đề gì. Ở bên trên, ta truyền Adapter cho nó, mà sẽ thấy như kiểu `PayD`, nhưng bên trong bao bọc đối tượng kiểu `Xpay`.

Như vậy trong Dự án của Max tất cả những gì chúng ta cần làm là cài đặt API của cổng trả tiền mới vào code và truyền Adapter đến phương thức của nhà cung cấp này để thanh toán tiền. Chúng ta không cần thay đổi bất cứ cái gì trong code đã có.



6.2.1 Khi nào sử dụng mẫu Adapter

Mẫu Adapter cần được sử dụng khi:

- Có lớp đã tồn tại, và giao diện của nó không khớp với cái ta cần
- Bạn muốn tạo một lớp tái sử dụng mà hợp tác với các lớp không liên quan và không biết trước và rằng các lớp này không cần thiết phải có giao diện tương thích.
- Có một số lớp con đã tồn tại để sử dụng, nhưng nó không thể thích nghi giao diện của chúng bằng cách tạo lớp con cho mỗi lớp. Đối tượng Adapter có thể thích nghi giao diện cho các lớp cha.

6.3 Mẫu thiết kế FACADE

6.3.1 Mở đầu

Trong bài này, chúng ta sẽ bàn luận về một mẫu cấu trúc khác, tức là mẫu Façade (về bên ngoài). Nhưng trước hết ta sẽ bàn luận về bài toán mà sẽ được giải quyết bởi mẫu này.

Công ty của bạn là công ty chế tạo sản phẩm và sẽ đưa sản phẩm ra thị trường, tên là Máy chủ lập lịch (Schedule Server). Nó là một kiểu máy chủ được dùng để quản lý các công việc. Các công việc có thể có kiểu bất kỳ như danh sách thư điện tử email, tin nhắn sms, đọc hoặc viết các file từ một đích nào đó, hoặc chỉ đơn giản là truyền file từ nguồn đến đích. Sản phẩm được sử dụng bởi người phát triển để quản lý các kiểu công việc như vậy và có khả năng tập trung hướng tới mục tiêu kinh doanh của họ. Máy chủ thực hiện mỗi công việc tại mỗi thời điểm yêu cầu của họ và cũng quản lý mọi vấn đề bên dưới khác như vấn đề song song và an ninh bởi chính nó. Như một nhà phát triển, người ta chỉ cần code các yêu cầu kinh doanh liên quan và một số các lời gọi API sẽ được cung cấp để xếp lịch cho công việc theo yêu cầu của họ.

Mọi việc tiến triển tốt, cho đến khi khách hàng bắt đầu than phiền về các tiến trình bắt đầu start, kết thúc stop của máy chủ. Họ nói mặc dù máy chủ làm việc tốt, các tiến trình khởi động và dừng còn rất phức tạp và họ muốn có cách nào khác để làm điều đó. Máy chủ đưa ra giao diện phức tạp cho khách hàng mà trông hơi đáng sợ đối với họ.

Chúng ta cần cung cấp cách dễ dàng hơn để khởi động và tắt máy chủ.

Một giao diện phức tạp đối với khách hàng được xem xét như một lỗi thiết kế của hệ thống hiện tại. Nhưng may mắn hoặc không may mắn, chúng ta không thể bắt đầu thiết kế lại và coding từ bản nháp. Chúng ta cần một cách để giải quyết vấn đề này và làm cho giao diện dễ dàng truy cập hơn.

Mẫu Façade có thể giúp chúng ta giải quyết bài toán thiết kế này. Nhưng trước hết, chúng ta tìm hiểu về mẫu Façade.

6.3.2 Mẫu Façade là gì

Mẫu Façade làm cho giao diện phức tạp dễ dàng sử dụng hơn. Mẫu Façade cung cấp giao diện thống nhất cho tập các giao diện trong một hệ thống con. Façade định nghĩa một giao diện mức độ cao mà làm cho hệ thống con đó dễ sử dụng hơn.

Façade thống nhất các giao diện mức thấp phức tạp của hệ thống con để cung cấp cách đơn giản truy cập đến giao diện đó. Nó chỉ cung cấp một mức cho các giao diện phức tạp của hệ thống con mà làm cho nó dễ sử dụng hơn.

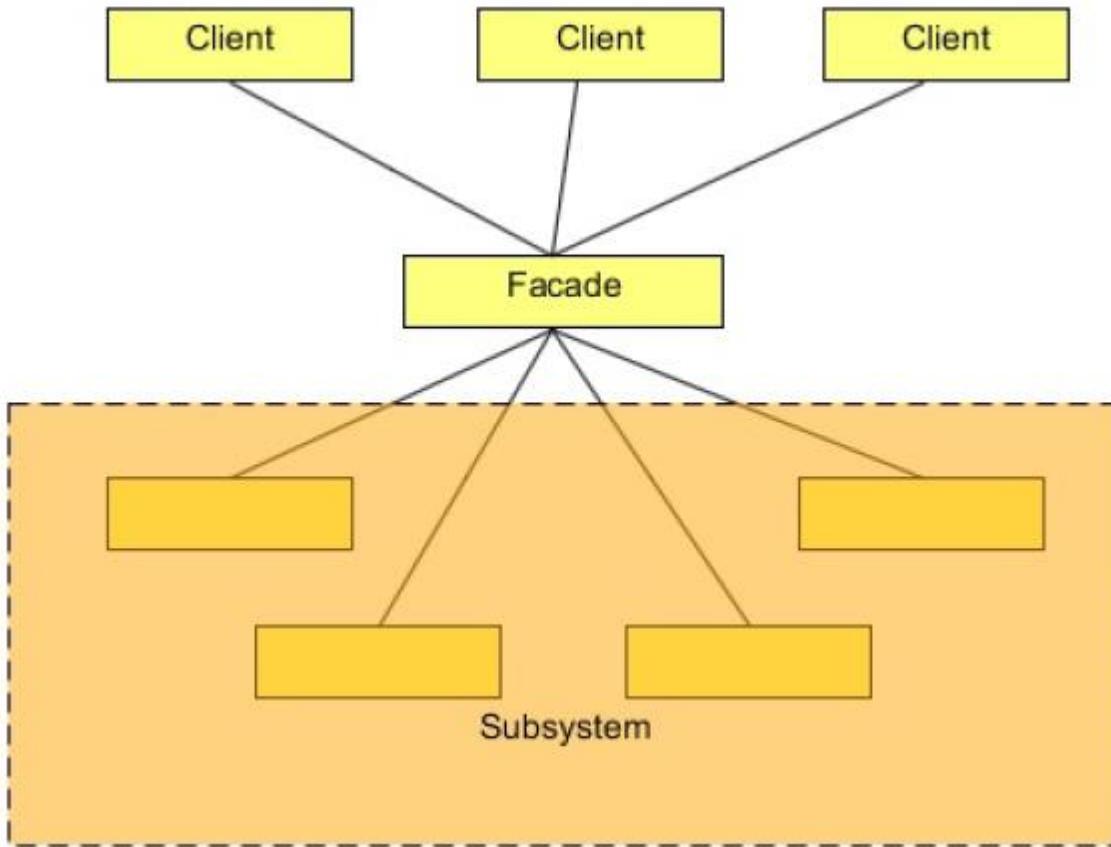
Façade không đóng gói các lớp hoặc giao diện của hệ thống con, nó chỉ cung cấp giao diện đơn giản hóa cho chức năng của chúng. Khách hàng có thể truy cập trực tiếp các lớp này. Nó vẫn phơi bày ra chức năng đầy đủ của hệ thống cho các khách hàng, những người có thể cần nó.

Façade không chỉ có thể đơn giản hóa giao diện, nhưng nó cũng tách gắn kết giữa khách hàng và hệ thống. Nó bám chặt vào “nguyên lý biết ít nhất” mà tránh gắn chặt giữa khách hàng và

hệ thống con. Nó cung cấp sự linh hoạt: giả sử trong hệ thống trên, một công ty muốn bổ sung một số bước để khởi động hoặc tắt máy chủ lập lịch, mà có giao diện khác nhau của riêng mình. Nếu bạn đã lập trình code khách hàng của bạn cho Façade thay vì cho hệ thống con, code client của bạn không cần phải thay đổi, chỉ Façade được yêu cầu thay đổi, mà có thể phân bổ như phiên bản mới cho client.

Các khách hàng trao đổi với hệ thống con bằng cách gửi yêu cầu đến Façade, mà nó sẽ truyền tiếp chúng cho các đối tượng hệ thống con phù hợp. Mặc dù các đối tượng hệ thống con thực hiện các công việc thực tiễn, façade có thể làm công việc của nó là truyền giao diện của nó cho giao diện của hệ thống con. Client mà sử dụng Façade không cần truy cập hệ thống con trực tiếp.

Cần lưu ý rằng, Façade cũng như Adapter có thể bao bọc nhiều lớp, nhưng Façade được sử dụng cho một giao diện để đơn giản hóa việc sử dụng các giao diện phức tạp, trong khi Adapter được dùng để chuyển đổi một giao diện sang giao diện mà client mong muốn.



6.3.3 Giải pháp cho bài toán

Vấn đề đối mặt với khách hàng trong việc sử dụng Máy chủ Lập lịch là tính phức tạp của máy chủ khi khởi động và tắt các dịch vụ. Khách hàng muốn cách đơn giản để làm điều đó. Sau đây là đoạn code mà client được yêu cầu để khởi động và tắt máy chủ:

```
ScheduleServer scheduleServer = new ScheduleServer();
```

Để khởi động máy chủ, client cần tạo đối tượng của ScheduleServer và sau đó cần gọi các phương thức sau theo trình tự đó để bắt đầu khởi động và khởi tạo server.

```
scheduleServer.startBooting();
scheduleServer.readSystemConfigFile();
scheduleServer.init();
scheduleServer.initializeContext();
scheduleServer.initializeListeners();
scheduleServer.createSystemObjects();

System.out.println("Start working.....");
System.out.println("After work done.....");
```

Để tắt máy chủ, client cần gọi các phương thức sau theo cùng một thứ tự

```
scheduleServer.releaseProcesses();
scheduleServer.destory();

scheduleServer.destroySystemObjects();
scheduleServer.destoryListeners();
scheduleServer.destoryContext();
scheduleServer.shutdown();
```

Điều này trông nồng nàn với họ, họ không quan tâm đến việc thực hiện các thủ tục này và tại sao chúng cần. Ngay cả khi điều này làm cho một số khách hàng quan tâm họ phải quan tâm giao diện mức thấp của hệ thống, nói chung hầu hết khách hàng không thích điều đó.

Để giải quyết điều này, chúng ta cần tạo ra lớp Façade mà sẽ bao bọc đối tượng máy chủ. Lớp này sẽ cung cấp giao diện (phương thức) đơn giản cho khách hàng. Các giao diện này bên trong sẽ gọi các phương thức trên đối tượng máy chủ. Trước hết ta xem code và sau đó bàn luận về nó.

```
package com.javacodegeeks.patterns.facadepattern;

public class ScheduleServerFacade {

    private final ScheduleServer scheduleServer;

    public ScheduleServerFacade(ScheduleServer scheduleServer) {
        this.scheduleServer = scheduleServer;
    }

    public void startServer() {

        scheduleServer.startBooting();
        scheduleServer.readSystemConfigFile();
        scheduleServer.init();
        scheduleServer.initializeContext();
        scheduleServer.initializeListeners();
        scheduleServer.createSystemObjects();
    }

    public void stopServer() {

        scheduleServer.releaseProcesses();
        scheduleServer.destory();
        scheduleServer.destroySystemObjects();
        scheduleServer.destoryListeners();
        scheduleServer.destoryContext();
        scheduleServer.shutdown();
    }
}
```

Lớp ScheduleServerFacade là lớp Façade, mà bao bọc đối tượng ScheduleServer, nó khởi tạo đối tượng server qua hàm tạo của nó, và có hai phương thức đơn giản, *startServer ()*, và *stopServer ()*. Các phương thức này bên trong thực hiện khởi động và tắt máy chủ. Client chỉ cần gọi các phương thức đơn giản này. Bây giờ, không cần gọi mọi phương thức vòng đời và hủy, chỉ cần phương thức đơn giản và phần còn lại của tiến trình sẽ do Façade thực hiện.

Đoạn code sau chỉ ra cách Façade biến giao diện phức tạp thành đơn giản để sử dụng.

```
package com.javacodegeeks.patterns.facadepattern;

public class TestFacade {

    public static void main(String[] args) {

        ScheduleServer scheduleServer = new ScheduleServer();
        ScheduleServerFacade facadeServer = new ScheduleServerFacade(scheduleServer ←
            );
        facadeServer.startServer();
```

```
        System.out.println("Start working.....");
        System.out.println("After work done.....");

        facadeServer.stopServer();
    }

}
```

Cũng phải lưu ý rằng, mặc dù lớp Façade cung cấp giao diện đơn giản cho hệ thống con phức tạp, nó không đóng gói hệ thống con. Client vẫn còn có thể truy cập giao diện mức thấp của hệ thống con. Vì vậy, Façade cung cấp một mức bổ sung, giao diện đơn giản cho hệ thống con phức tạp, nhưng nó không hoàn toàn che giấu truy cập trực tiếp đến giao diện mức thấp của hệ thống con phức tạp.

6.3.4 Sử dụng mẫu Façade

Sử dụng mẫu Façade khi

- Bạn muốn cung cấp giao diện đơn giản cho hệ thống con phức tạp. Nó làm cho hệ thống con dễ tái sử dụng hơn và dễ tùy biến, nhưng nó có thể khó sử dụng hơn đối với khách hàng cần phải tùy biến theo yêu cầu. Façade cung cấp cái nhìn mặc định đơn giản về hệ thống con mà đủ tốt với hầu hết khách hàng. Chỉ khách hàng cần tùy biến sâu hơn sẽ cần xem xét kỹ hơn về Façade.
- Có nhiều phụ thuộc giữa client và các lớp cài đặt của trùm tượng. Đưa ra Façade để tách gắn kết hệ thống con khỏi khách hàng và các hệ thống khác, khi đó tăng sự độc lập và chuyển mang của hệ thống con.
- Bạn cần tạo mức cho các hệ thống con của bạn. Sử dụng Façade để xác định điểm vào mức từng hệ thống con. Nếu các hệ thống con là phụ thuộc, thì bạn có thể đơn giản sự phụ thuộc giữa chúng bằng cách làm cho chúng trao đổi với nhau duy nhất qua các Façade của chúng.

6.4 Mẫu thiết kế COMPOSITE

6.4.1 Mở đầu

Trong bài này, chúng ta sẽ nói về một mẫu thiết kế cấu trúc rất thú vị, mẫu Composite (hợp lại, ghép lại). Nghĩa tiếng Anh của từ Composite là một cái gì đó được làm từ những phần phức tạp và liên quan. Composite nghĩa là ghép với nhau và đó cũng là cái mà mẫu thiết kế này nói về điều đó.

Có những lúc bạn cần cấu trúc dữ liệu cây trong code của bạn. Ở đây có nhiều biến thể của cấu trúc dữ liệu cây, nhưng đôi khi cần cây mà ở đó cả hai cây con ở hai nhánh cũng như các lá cần được xử lý như nhau.

Mẫu Composite cho phép bạn ghép các đối tượng vào cấu trúc cây để biểu diễn phân cấp một phần - tất cả, mà có nghĩa là bạn có thể tạo một cây các đối tượng mà được làm từ các phần khác nhau, nhưng nó có thể được xử lý như một vật lớn trọn vẹn. Composite cho phép clients xử lý các đối tượng riêng rẽ và hợp lại của các đối tượng một cách đồng nhất, đó chính là chủ đích của mẫu thiết kế Composite.

Có thể có rất nhiều ví dụ thực tế về mẫu Composite. Hệ thống thư mục file, biểu diễn html trong Java, phân tích cú pháp XML, tất cả đều là composite và tất cả có thể dễ dàng được biểu diễn sử dụng mẫu Composite. Nhưng trước khi đi vào ví dụ cụ thể, chúng ta sẽ xét mẫu Composite một cách chi tiết.

6.4.2 Mẫu Composite là gì

Định nghĩa hình thức mẫu thiết kế nói rằng nó cho phép bạn hợp ghép các đối tượng thành cấu trúc cây biểu diễn phân cấp phần – toàn bộ. Composite cho phép client xử lý từng đối tượng và cả hợp ghép đối tượng đồng nhất.

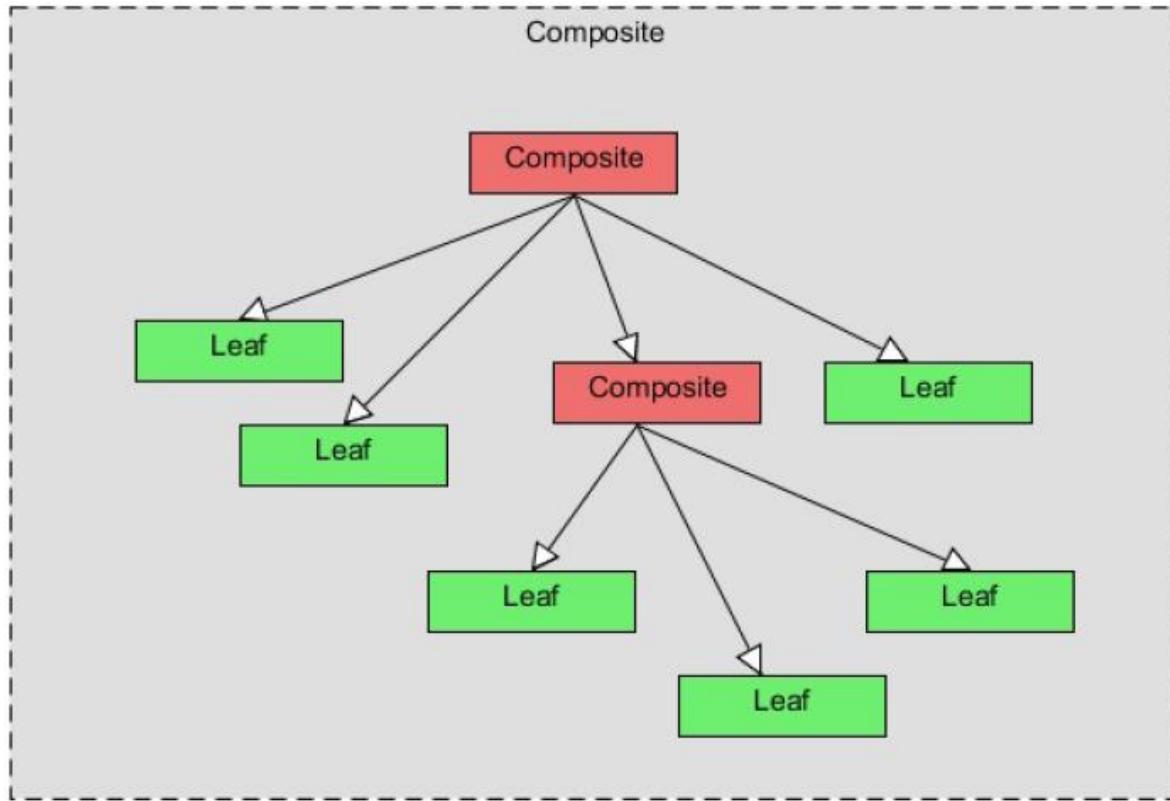
Nếu bạn đã quen với cấu trúc dữ liệu cây, bạn sẽ biết cây có các cha và các con của chúng. Có thể có nhiều con của một cha, nhưng chỉ có một cha cho một con. Trong mẫu Composite, các thành phần có con gọi là node và các thành phần không có con gọi là lá.

Mẫu Composite cho phép chúng ta xây dựng cấu trúc của các đối tượng ở dạng cây mà chưa cả hợp ghép của các đối tượng và các đối tượng riêng rẽ như các node. Sử dụng cấu trúc Composite, chúng ta có thể áp dụng cùng một phép toán trên cả các đối tượng hợp ghép và riêng lẻ.

Mẫu Composite có bốn thành viên:

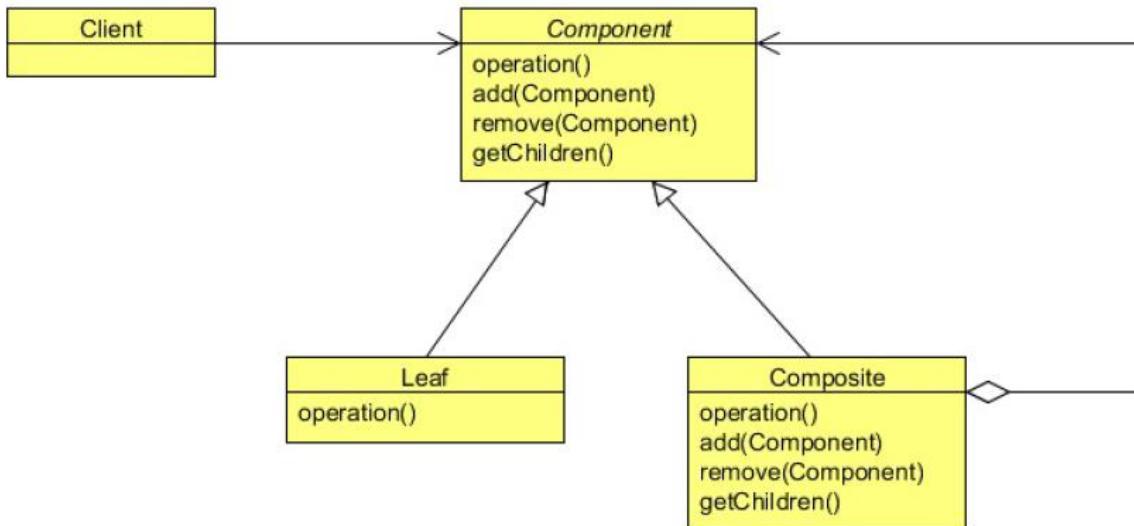
- Component
- Leaf
- Composite
- Client

Hình vẽ sau chỉ ra cấu trúc đối tượng Composite tiêu biểu. Như bạn thấy, có thể có nhiều con của một cha, tức là Composite, nhưng chỉ có một Composite cho một con.



Component trong lược đồ lớp dưới đây định nghĩa giao diện cho mọi đối tượng ở cả các node composite và lá. Component có thể cài đặt hành vi mặc định cho các phương thức tổng quát.

Vai trò của Composite là định nghĩa hành vi của các component có con và lưu giữ các component con. Composite cũng cài đặt các thao tác liên quan đến lá. Các thao tác này có thể có hoặc không có ý nghĩa gì, nó phụ thuộc vào chức năng cài đặt sử dụng mẫu.



Lá Leaf định nghĩa hành vi cho các phần tử trong hợp ghép. Nó làm điều này bằng cách cài đặt các thao tác mà Component hỗ trợ. Leaf cũng kế thừa các phương thức mà không có ý nghĩa lầm đối với lá.

Client thao tác các đối tượng trong hợp ghép thông qua giao diện Component.

6.4.3 Ví dụ mẫu Composite

Mẫu Composite có thể được cài đặt ở bất cứ đâu mà bạn có môi trường phân cấp của một hệ thống hoặc hệ thống con mà bạn muốn xử lý các đối tượng riêng rẽ và hợp ghép các đối tượng một cách đồng nhất. Hệ thống File, XML, Html, hoặc phân cấp công sở (bắt đầu từ chủ tịch đến nhân viên) có thể được cài đặt nhờ mẫu Composite.

Xét ví dụ đơn giản sau, ở đó chúng ta cài đặt biểu diễn html trong Java sử dụng mẫu Composite. Html là phân cấp tự nhiên, nó bắt đầu từ tag <html>, mà là cha hoặc tag gốc, và nó chứa các tag khác mà có thể là tag cha hoặc tag con.

Mẫu Composite trong Java có thể được cài đặt sử dụng lớp Component như lớp trùu tượng hoặc giao diện. Trong ví dụ này, chúng ta sẽ sử dụng đó là một lớp abstract mà chứa mọi phương thức quan trọng sử dụng trong lớp Composite và lớp Leaf.

```
package com.javacodegeeks.patterns.compositepattern;

import java.util.List;

public abstract class HtmlTag {

    public abstract String getTagName();
    public abstract void setStartTag(String tag);
    public abstract void setEndTag(String tag);
    public void setTagBody(String tagBody) {
        throw new UnsupportedOperationException("Current operation is not support for this object");
    }
    public void addChildTag(HtmlTag htmlTag) {
        throw new UnsupportedOperationException("Current operation is not support for this object");
    }
    public void removeChildTag(HtmlTag htmlTag) {
        throw new UnsupportedOperationException("Current operation is not support for this object");
    }
    public List<HtmlTag>getChildren() {
        throw new UnsupportedOperationException("Current operation is not support for this object");
    }
    public abstract void generateHtml();
}
```

Lớp `HtmlTag` là lớp component mà định nghĩa mọi phương thức được sử dụng bởi lớp composite và lớp leaf. Có một số phương thức mà có thể là chung cho cả hai lớp mở rộng, do đó các phương thức này được giữ là `abstract` trong lớp trên, buộc phải cài đặt chúng trong các lớp con.

Phương thức `getTagName()` mà trả về tên tag và cần được sử dụng bởi cả hai lớp con, tức là lớp composite và lớp leaf.

Mỗi phần tử html cần có start tag và end tag, các phương thức `setStartTag` và `setEndTag` được sử dụng để đặt start và end tag của phần tử html đó và cần được cài đặt bởi cả hai lớp con, như vậy chúng được giữ `abstract` trong lớp trên.

Có các phương thức mà là có ích chỉ cho lớp composite và không được dùng cho lớp leaf. Chỉ cung cấp cài đặt mặc định cho các phương thức này, đưa ra ngoại lệ là cài đặt tốt cho các phương thức này để tránh bắt cứ lời gọi sự cố nào đến các phương thức này bởi các đối tượng mà không hỗ trợ chúng.

Phương thức `generateHtml()` là thao tác mà cần phải hỗ trợ bởi cả hai lớp mở rộng. Để đơn giản, nó chỉ in tag ra màn hình.

Bây giờ ta xem lớp Composite

```
package com.javacodegeeks.patterns.compositepattern;

import java.util.ArrayList;
import java.util.List;

public class HtmlParentElement extends HtmlTag {

    private String tagName;
    private String startTag;
    private String endTag;
    private List<HtmlTag> childrenTag;

    public HtmlParentElement(String tagName) {
        this.tagName = tagName;
        this.startTag = "";
        this.endTag = "";
        this.childrenTag = new ArrayList<>();
    }

    @Override
    public String getTagName() {
        return tagName;
    }

    @Override
    public void setStartTag(String tag) {
        this.startTag = tag;
    }

    @Override

    public void setEndTag(String tag) {
        this.endTag = tag;
    }

    @Override
    public void addChildTag(HtmlTag htmlTag) {
        childrenTag.add(htmlTag);
    }

    @Override
    public void removeChildTag(HtmlTag htmlTag) {
        childrenTag.remove(htmlTag);
    }

    @Override
    public List<HtmlTag> getChildren() {
        return childrenTag;
    }

    @Override
    public void generateHtml() {
        System.out.println(startTag);
        for(HtmlTag tag : childrenTag) {
            tag.generateHtml();
        }
        System.out.println(endTag);
    }
}
```

Lớp HtmlParentElement là lớp composite mà cài đặt phương thức như *addChildTag*, *removeChildTag*, *getChildren*, mà cần phải được cài đặt bởi một lớp để trở thành composite của cấu trúc này. Phương thức operation ở đây là *generateHtml*, mà in ra tag của lớp hiện tại, và cũng lặp qua các con của nó và cũng gọi phương thức *generateHtml* của chúng.

```
package com.javacodegeeks.patterns.compositepattern;

public class HtmlElement extends HtmlTag{

    private String tagName;
    private String startTag;
    private String endTag;
    private String tagBody;

    public HtmlElement(String tagName) {
        this.tagName = tagName;
        this.tagBody = "";
        this.startTag = "";
        this.endTag = "";
    }

    @Override
    public String getTagName() {
        return tagName;
    }

    @Override
    public void setStartTag(String tag) {
        this.startTag = tag;
    }

    @Override
    public void setEndTag(String tag) {
        this.endTag = tag;
    }

    @Override
    public void setTagBody(String tagBody) {
        this.tagBody = tagBody;
    }

    @Override
    public void generateHtml() {
        System.out.println(startTag+" "+tagBody+" "+endTag);
    }
}
```

Lớp HtmlElement là lớp Leaf, và công việc chính của nó là cài đặt phương thức operation, mà trong ví dụ này là phương thức *generateHtml*. Nó in StartTag, tùy chọn có thể có tagBody, và endTag của phân tử con.

Hãy kiểm tra ví dụ này.

```
package com.javacodegeeks.patterns.compositepattern;

public class TestCompositePattern {

    public static void main(String[] args) {
        HtmlTag parentTag = new HtmlParentElement("<html>");
        parentTag.setStartTag("<html>");
        parentTag.setEndTag("</html>");

        HtmlTag p1 = new HtmlParentElement("<body>");
        p1.setStartTag("<body>");
        p1.setEndTag("</body>");

        parentTag.addChildTag(p1);

        HtmlTag child1 = new HtmlElement("<P>");
        child1.setStartTag("<P>");
        child1.setEndTag("</P>");
        child1.setTagBody("Testing html tag library");
        p1.addChildTag(child1);

        child1 = new HtmlElement("<P>");
        child1.setStartTag("<P>");
        child1.setEndTag("</P>");
        child1.setTagBody("Paragraph 2");
        p1.addChildTag(child1);

        parentTag.generateHtml();
    }
}
```

Code trên cho ra kết quả như sau:

```
<html>
<body>
<P>Testing html tag library</P>
<P>Paragraph 2</P>
</body>

</html>
```

Trong ví dụ trên, trước hết ta tạo parent tag (<html>), sau đó bổ sung con cho nó, mà là cái khác kiểu composite (<body>) và đối tượng này chứa hai con (<P>).

Lưu ý rằng, cấu trúc bên trên biểu diễn phân cấp bộ phận – toàn bộ và lời gọi phương thức *generateHtml ()* trên parent tag cho phép client xử lý hợp ghép các đối tượng một cách nhất quán. Như vậy nó sinh ra html của đối tượng đó và của toàn bộ con của nó.

6.4.4 Khi nào sử dụng mẫu Composite

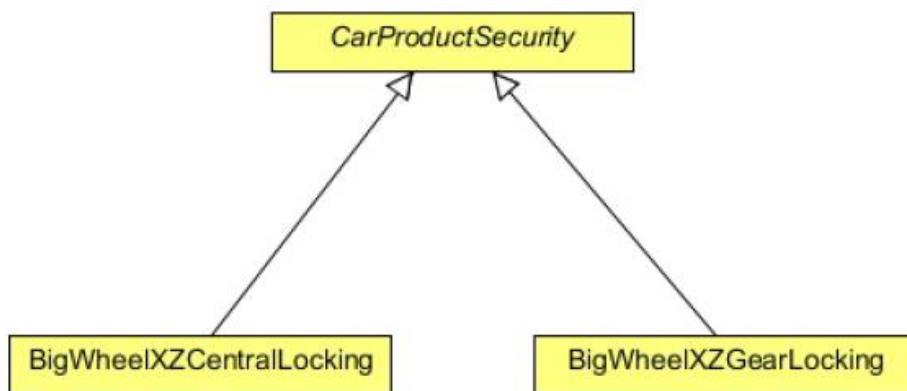
- Khi bạn muốn biểu diễn phân cấp bộ phận – toàn bộ của các đối tượng
- Khi bạn muốn client có khả năng bỏ qua sự khác biệt giữa hợp ghép đối tượng và các đối tượng riêng lẻ. Client sẽ xử lý mọi đối tượng trên cấu trúc composite này một cách nhất quán.

6.5 Mẫu thiết kế BRIDGE

6.5.1 Mở đầu

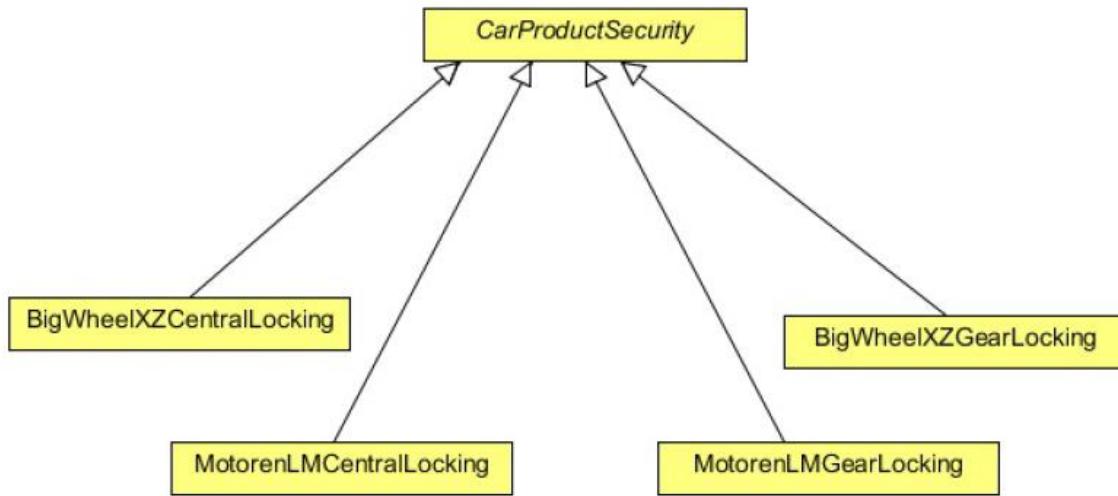
Hệ thống an ninh Sec là sản phẩm của công ty điện tử và an ninh, mà sản xuất và lắp đặt các sản phẩm cho ôtô. Nó phân phối bất cứ hệ thống điện tử và an ninh cho ôtô mà bạn muốn, từ túi khí cho đến hệ thống giám sát GPS, hệ thống đỗ lùi. Các công ty ôtô lớn sử dụng các sản phẩm của nó trong ôtô của họ. Công ty sử dụng cách tiếp cận hướng đối tượng được xác định tốt để theo dõi các sản phẩm của họ sử dụng phần mềm mà được phát triển và bảo trì chỉ bởi họ. Họ nhận ôtô và tạo ra các sản phẩm cho nó và lắp đặt chúng vào ôtô.

Hiện tại, họ nhận được các hợp đồng mới từ BigWheel (một công ty ôtô) để tạo ra hệ thống khóa bánh răng và khóa trung tâm cho mẫu ôtô mới xz. Để bảo trì nó, họ tạo ra hệ thống phần mềm mới. Họ bắt đầu từ việc tạo lớp abstract mới CarProductSecurity, mà ở đó họ giữ một số phương thức đặc thù cho ôtô và một số đặc trưng mà họ nghĩ là dùng chung cho mọi sản phẩm an ninh. Sau đó họ kế thừa lớp này để tạo ra hai lớp con khác nhau đặt tên là BigWheelXZCentral Locking và BigWheelXZGear Locking. Lược đồ lớp trông như sau:



Thời gian sau, một công ty ôtô khác Motoren yêu cầu họ sản xuất hệ thống khóa trung tâm và hệ thống khóa bánh răng mới cho mẫu ôtô mới lm. Do đó, một hệ thống an ninh cùng loại không thể được sử dụng cho cả hai mô hình ôtô khác nhau, Hệ thống an ninh Sec buộc phải sản xuất hệ thống mới cho họ, và buộc phải tạo các lớp mới MotorenLMCentralLocking và MotorenMLGearLocking mà cũng mở rộng lớp CarProductSecurity.

Bây giờ lược đồ lớp mới trông như sau:



Như vậy cũng được, nhưng điều gì xảy ra, nếu một công ty ôtô nữa lại yêu cầu một hệ thống mới khác khóa trung tâm và khóa bánh răng? Người ta cần phải tạo hai lớp mới cho nó. Thiết kế này sẽ tạo mỗi lớp cho một hệ thống, và sẽ tồi tệ hơn, nếu hệ thống đó lùi lại được sản xuất cho mỗi trong số hai công ty ôtô, hai hoặc nhiều hơn lớp mới sẽ được tạo cho mỗi một trong số đó.

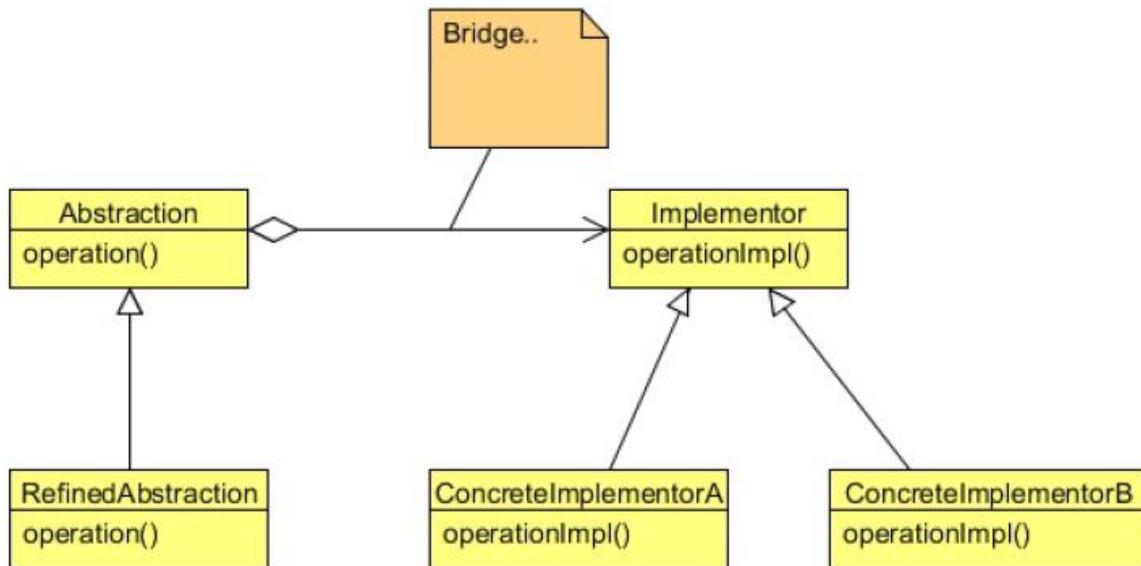
Một thiết kế với quá nhiều lớp con sẽ không linh hoạt và khó bảo trì. Kế thừa sẽ trói cài đặt với lớp trừu tượng thường xuyên, mà làm cho khó thay đổi, mở rộng và tái sử dụng lớp trừu tượng và cài đặt một cách độc lập.

Lưu ý rằng, ôtô và sản phẩm cần phải được đa dạng độc lập để tạo nên phần mềm dễ dàng mở rộng và tái sử dụng.

Mẫu thiết kế Bridge có thể giải quyết bài toán này, nhưng trước hết chúng ta xem xét chi tiết mẫu Bridge.

6.5.2 Mẫu thiết kế Bridge là gì

Mẫu thiết kế Bridge hướng tới tách gán kết trừu tượng ra khỏi cài đặt sao cho hai cái đó có thể đa dạng một cách độc lập. Nó đặt trừu tượng và cài đặt thành hai phân cấp khác nhau sao cho cả hai có thể mở rộng độc lập.



Các thành phần của mẫu Bridge được tạo từ trừu tượng, trừu tượng làm mịn, cài đặt và cài đặt cụ thể.

Trừu tượng xác định giao diện trừu tượng và cũng bảo trì tham chiếu đến đối tượng kiểu cài đặt và kết nối giữa trừu tượng và cài đặt được gọi là cái cầu Bridge.

Trừu tượng làm mịn mở rộng giao diện trừu tượng.

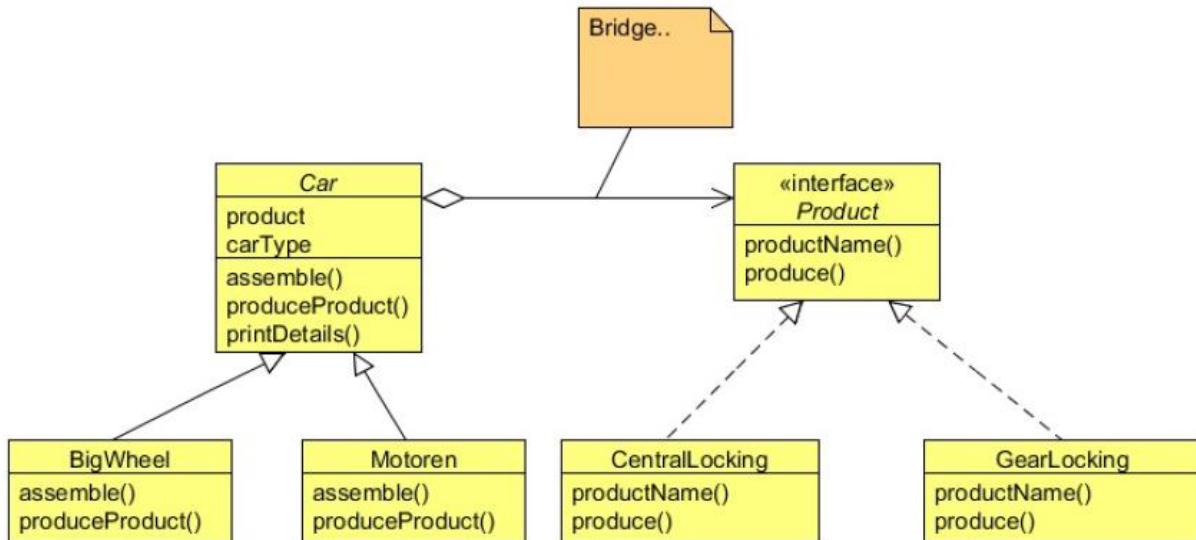
Cài đặt cung cấp giao diện cho lớp cài đặt, tức là cài đặt cụ thể.

Và cài đặt cụ thể triển khai giao diện Implementer và xác định cài đặt cụ thể.

Mẫu Bridge tách giao diện khỏi cài đặt. Kết quả là, cài đặt không gắn thường xuyên với giao diện. Cài đặt của trừu tượng có thể được cấu hình trong thời gian chạy. Nó cũng loại bỏ sự phụ thuộc thời gian dịch vào cài đặt. Thay đổi lớp cài đặt không đòi hỏi dịch lại lớp trừu tượng và client của nó. Client chỉ cần biết về trừu tượng và bạn có thể che giấu cài đặt khỏi chúng.

6.5.3 Giải pháp cho bài toán

Thay vì tạo các lớp con cho mỗi sản phẩm cho mỗi mẫu xe trong bài toán đang bàn luận ở trên, chúng ta có thể tách thiết kế thành hai phân cấp khác nhau. Một giao diện cho sản phẩm mà sẽ được dùng như một cài đặt và một cái khác sẽ là một trừu tượng của một kiểu ôtô.



```

package com.javacodegeeks.patterns.bridgepattern;

public interface Product {
    public String productName();
    public void produce();
}
  
```

Bộ cài Implementer có phương thức `produce()`, mà sẽ được sử dụng bởi bộ cài cụ thể concrete implementer để cung cấp chức năng cụ thể cho nó. Phương thức này sẽ tạo ra mẫu cơ bản của sản phẩm mà sẽ được sử dụng với bất cứ mẫu ôtô nào sau một số điều chỉnh chuyên biệt cho mẫu ôtô đó.

```

package com.javacodegeeks.patterns.bridgepattern;

public class CentralLocking implements Product {
    private final String productName;

    public CentralLocking(String productName) {
        this.productName = productName;
    }

    @Override
    public String productName() {
        return productName;
    }

    @Override
    public void produce() {
        System.out.println("Producing Central Locking System");
    }
}

package com.javacodegeeks.patterns.bridgepattern;
  
```

```
public class GearLocking implements Product{  
    private final String productName;  
    public GearLocking(String productName){  
        this.productName = productName;  
    }  
    @Override  
    public String productName(){  
        return productName;  
    }  
    @Override  
    public void produce(){  
        System.out.println("Producing Gear Locking System");  
    }  
}
```

Hai bộ cài cụ thể cung cấp cài đặt cho bộ cài Product.

Bây giờ nói đến trùu tượng là lớp Car, mà giữ tham chiếu đến kiểu sản phẩm và cung cấp hai phương thức *produceProduct()* và *assemble()*.

```
package com.javacodegeeks.patterns.bridgepattern;  
  
public abstract class Car {  
    private final Product product;  
    private final String carType;  
    public Car(Product product, String carType){  
        this.product = product;  
        this.carType = carType;  
    }  
    public abstract void assemble();  
    public abstract void produceProduct();  
    public void printDetails(){  
        System.out.println("Car: "+carType+", Product:"+product.productName());  
    }  
}
```

Các lớp con của Car sẽ cung cấp các cài đặt cụ thể và chuyên biệt cho các phương thức *assemble* và *produceProduct()*.

```
package com.javacodegeeks.patterns.bridgepattern;

public class BigWheel extends Car{

    private final Product product;
    private final String carType;

    public BigWheel(Product product, String carType) {
        super(product, carType);
        this.product = product;
        this.carType = carType;
    }

    @Override

    public void assemble() {
        System.out.println("Assembling "+product.productName()+" for "+carType);
    }

    @Override
    public void produceProduct() {
        product.produce();
        System.out.println("Modifying product "+product.productName()+" according to ←
                           "+carType);
    }

}

package com.javacodegeeks.patterns.bridgepattern;

public class Motoren extends Car{

    private final Product product;
    private final String carType;

    public Motoren(Product product, String carType) {
        super(product, carType);
        this.product = product;
        this.carType = carType;
    }

    @Override
    public void assemble() {
        System.out.println("Assembling "+product.productName()+" for "+carType);
    }

    @Override
    public void produceProduct() {
        product.produce();
        System.out.println("Modifying product "+product.productName()+" according to ←
                           "+carType);
    }

}
```

Bây giờ ta sẽ kiểm tra ví dụ này

```
package com.javacodegeeks.patterns.bridgepattern;

public class TestBridgePattern {

    public static void main(String[] args) {
        Product product = new CentralLocking("Central Locking System");
        Product product2 = new GearLocking("Gear Locking System");
        Car car = new BigWheel(product , "BigWheel xz model");
        car.produceProduct();
        car.assemble();
        car.printDetails();

        System.out.println();

        car = new BigWheel(product2 , "BigWheel xz model");
        car.produceProduct();
        car.assemble();
        car.printDetails();

        car = new Motoren(product, "Motoren lm model");

        car.produceProduct();
        car.assemble();
        car.printDetails();

        System.out.println();

        car = new Motoren(product2, "Motoren lm model");
        car.produceProduct();
        car.assemble();
        car.printDetails();
    }
}
```

Ví dụ trên sẽ in ra kết quả sau:

```
Producing Central Locking System
Modifing product Central Locking System according to BigWheel xz model
Assembling Central Locking System for BigWheel xz model
Car: BigWheel xz model, Product:Central Locking System

Producing Gear Locking System
Modifing product Gear Locking System according to BigWheel xz model
Assembling Gear Locking System for BigWheel xz model
Car: BigWheel xz model, Product:Gear Locking System

Producing Central Locking System
Modifing product Central Locking System according to Motoren lm model
Assembling Central Locking System for Motoren lm model
Car: Motoren lm model, Product:Central Locking System

Producing Gear Locking System
Modifing product Gear Locking System according to Motoren lm model
Assembling Gear Locking System for Motoren lm model
Car: Motoren lm model, Product:Gear Locking System
```

6.5.4 Sử dụng mẫu Bridge

Bạn sẽ sử dụng mẫu Bridge khi:

- Bạn muốn tránh trói buộc thường xuyên giữa trùu tượng và các cài đặt của nó. Điều này có thể là trường hợp, chẳng hạn cài đặt cần được lựa chọn hoặc chuyển đổi trong thời gian chạy.
- Cả các trùu tượng và các cài đặt cần được mở rộng bằng các lớp con. Trong trường hợp này, mẫu Bridge cho phép kết hợp các trùu tượng và cài đặt khác nhau và mở rộng chúng một cách độc lập.
- Các thay đổi trong cài đặt của trùu tượng không có tác động đến client; như vậy code của họ sẽ không phải dịch lại.
- Bạn muốn chia sẻ một cài đặt giữa nhiều đối tượng (chẳng hạn, sử dụng đếm tham chiếu) và sự việc này cần được che giấu khỏi client.

6.6 Mẫu thiết kế SINGLETON

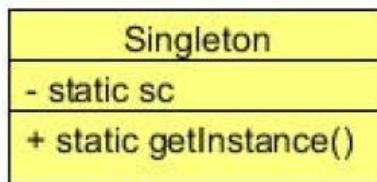
6.6.1 Mở đầu

Đôi khi rất quan trọng đối với một số lớp là chỉ có chính xác một khởi tạo. Có rất nhiều đối tượng mà chúng ta chỉ cần một khởi tạo của chúng và nếu chúng ta khởi tạo nhiều lần, chúng ta sẽ gặp nhiều vấn đề như hành vi chương trình không đúng, lạm dụng tài nguyên hoặc các kết quả không tương thích.

Bạn có thể yêu cầu chỉ có một đối tượng cho một lớp, chẳng hạn, khi bạn tạo một ngữ cảnh xác định của một ứng dụng hoặc một bể luồng quản trị được, thiết lập đăng ký, một driver kết nối với một thiết bị đầu vào và đầu ra. Nhiều hơn một đối tượng kiểu này rõ ràng sẽ sinh ra sự không tương thích cho hệ thống của bạn.

Mẫu Singleton đảm bảo rằng, một lớp chỉ có một khởi tạo, và cung cấp một điểm truy cập tổng thể đến nó. Tuy nhiên, mặc dù mẫu Singleton là đơn giản nhất theo sơ đồ lớp, vì ở đây chỉ có một lớp duy nhất, cài đặt của nó là hơi rắc rối.

Trong bài này chúng ta sẽ thử các cách khác nhau để tạo nên một đối tượng duy nhất cho một lớp và sẽ nhìn thấy cách nào tốt hơn cách kia như thế nào.



6.6.2 Tạo một lớp sử dụng mẫu Singleton như thế nào.

Có nhiều cách để tạo kiểu lớp như vậy, nhưng chúng ta sẽ thấy một cách sẽ tốt hơn cách khác như thế nào.

Bắt đầu bởi cách đơn giản nhất.

Cái gì sẽ xảy ra, nếu ta cung cấp một biến tổng thể mà tạo nên một đối tượng truy cập được.
Chẳng hạn:

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonEager {
    public static SingletonEager sc = new SingletonEager();
}
```

Như chúng ta biết, ở đây chỉ có một bản sao của biến tĩnh của một lớp, chúng ta có thể áp dụng nó. Code của client sử dụng biến `sc` tĩnh này có vẻ là tốt. Nhưng, nếu client sử dụng thao tác `new SingletonEager()`, ở đây có thể có khởi tạo mới của lớp này.

Để không cho phép lớp có thể được khởi tạo ngoài lớp, ta sẽ làm cho hàm tạo là *private*.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonEager {
    public static SingletonEager sc = new SingletonEager();
    private SingletonEager(){}
}
```

Bằng cách giữ hàm tạo là *private*, không lớp nào có thể khởi tạo được lớp này. Chỉ có một cách lấy được đối tượng của lớp là sử dụng biến tĩnh *sc* mà đảm bảo chỉ có một đối tượng ở đây.

Nhưng như chúng ta biết, cung cấp truy cập trực tiếp đến thành viên của lớp không phải là ý tưởng hay. Chúng ta sẽ cung cấp một phương thức mà biến *sc* sẽ được nhận không trực tiếp.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonEager {
    private static SingletonEager sc = new SingletonEager();
    private SingletonEager(){}
    public static SingletonEager getInstance(){
        return sc;
    }
}
```

Như vậy, đây là lớp singleton mà đảm bảo chỉ một đối tượng của một lớp được tạo và ngay cả nếu có một số yêu cầu, thì trả về cùng một đối tượng đã được khởi tạo.

Có một vấn đề với cách tiếp cận này là, đối tượng sẽ được tạo ngay khi lớp được tải vào máy ảo JVM. Nếu đối tượng không được yêu cầu, thì có thể đối tượng là không có ích bên trong bộ nhớ.

Luôn luôn là cách tiếp cận tốt nếu đối tượng được tạo khi nó được yêu cầu. Vì vậy chúng ta sẽ tạo đối tượng ở lời gọi đầu tiên và sẽ trả về cùng đối tượng đó trong các lời gọi tiếp theo.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonLazy {

    private static SingletonLazy sc = null;
    private SingletonLazy(){}
    public static SingletonLazy getInstance(){
        if(sc==null){
            sc = new SingletonLazy();
        }
        return sc;
    }
}
```

Trong phương thức *getInstance()*, ta kiểm tra nếu biến *sc* là null, thì ta sẽ khởi tạo đối tượng và trả về nó. Như vậy trong lần gọi đầu khi *sc* có thể là null, đối tượng được tạo và trong các lời gọi tiếp theo sẽ trả về cùng đối tượng đó.

Điều này trông có vẻ tốt? Tuy nhiên code này sẽ lỗi trong môi trường đa luồng. Tưởng tượng rằng, có hai luồng song song truy cập đến lớp, một luồng t1 đưa ra lời gọi đầu tiên đến phương thức `getInstance ()`, nó kiểm tra nếu biến tĩnh sc là null và sau đó bị ngắt vì một lý do nào đó. Luồng thứ hai t2 gọi phương thức `getInstance ()`, qua lệnh kiểm tra `if` thành công và khởi tạo đối tượng. Sau đó luồng t1 quay trở lại và nó cũng tạo đối tượng mới. Trong thời điểm này, có thể có hai đối tượng của lớp đó.

Để tránh điều này, chúng ta sẽ sử dụng từ khóa `synchronized` cho phương thức `getInstance ()`. Bằng cách này, chúng ta buộc mỗi luồng phải chờ đến lượt nó trước khi sử dụng phương thức. Vì vậy không có hai luồng sẽ sử dụng phương thức cùng một thời điểm. Phương thức đồng bộ được trả giá, là nó giảm hiệu năng, nhưng nếu lời gọi `getInstance ()` sẽ không phát sinh thêm gánh nặng cho ứng dụng, thì không sao. Phương án khác là chuyển đến cách tiếp cận khởi tạo sớm như đã nêu trong ví dụ trước.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonLazyMultithreaded {

    private static SingletonLazyMultithreaded sc = null;
    private SingletonLazyMultithreaded(){}
    public static synchronized SingletonLazyMultithreaded getInstance() {
        if(sc==null){
            sc = new SingletonLazyMultithreaded();
        }
        return sc;
    }
}
```

Nhưng nếu bạn muốn sử dụng đồng bộ, có một kỹ thuật khác là “khóa kiểm tra hai lần” để giảm việc sử dụng đồng bộ. Với “khóa kiểm tra hai lần”, trước hết ta kiểm tra xem một khởi tạo đã được tạo chưa, nếu chưa, thì ta đồng bộ. Cách này, ta chỉ sử dụng đồng bộ lần đầu.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonLazyDoubleCheck {

    private volatile static SingletonLazyDoubleCheck sc = null;
    private SingletonLazyDoubleCheck(){}
    public static SingletonLazyDoubleCheck getInstance(){
        if(sc==null){
            synchronized(SingletonLazyDoubleCheck.class){
                if(sc==null){
                    sc = new SingletonLazyDoubleCheck();
                }
            }
        }
        return sc;
    }
}
```

Ngoài ra còn một số cách khác mà bẻ mẫu singleton.

- Nếu lớp là `Serializable`
- Nếu nó là sử dụng được `clone`

- Và cũng, nếu lớp đó được tải bằng bộ tải đa lớp
- Ví dụ sau chỉ ra bạn có thể bảo vệ lớp của bạn khỏi việc khởi tạo hơn một lần như thế nào

```
package com.javacodegeeks.patterns.singletonpattern;

import java.io.ObjectStreamException;
import java.io.Serializable;

public class Singleton implements Serializable{

    private static final long serialVersionUID = -1093810940935189395L;
    private static Singleton sc = new Singleton();
    private Singleton() {

        if(sc!=null){
            throw new IllegalStateException("Already created.");
        }
    }

    public static Singleton getInstance () {
        return sc;
    }

    private Object readResolve() throws ObjectStreamException{
        return sc;
    }

    private Object writeReplace() throws ObjectStreamException{
        return sc;
    }

    public Object clone() throws CloneNotSupportedException{
        throw new CloneNotSupportedException("Singleton, cannot be cloned");
    }

    private static Class getClass(String classname) throws ClassNotFoundException {
        ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
        if(classLoader == null)
            classLoader = Singleton.class.getClassLoader();
        return (classLoader.loadClass(classname));
    }

}
```

- Cài đặt các *readResolve ()* và *writeReplace ()* trong lớp *singleton* của bạn và trả về cùng một đối tượng qua chúng.
- Bạn cần cài đặt phương thức *clone ()* và đưa ra ngoại lệ sao cho *singleton* không thể được *clone*.
- Để ngăn cản *singleton* được khởi tạo từ các bộ tải lớp khác nhau, bạn có thể cài đặt phương thức *getClass ()*. Phương thức *getClass ()* trên đây liên kết với bộ tải với luồng hiện tại, nếu bộ tải là *null*, phương thức sẽ sử dụng cùng một bộ tải mà tải lớp *singleton*.

Mặc dù chúng ta có thể sử dụng mọi kỹ thuật trên, có một cách đơn giản và dễ dàng để tạo lớp singleton. Như JDK 1.5, bạn có thể tạo lớp singleton sử dụng enums. Hằng số Enum là tĩnh ẩn và là không thay đổi (*final*) và bạn không thể thay đổi giá trị của chúng khi được tạo ra.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonEnum {
    public enum SingleEnum{
        SINGLETON_ENUM;
    }
}
```

Bạn sẽ có lỗi trong thời gian dịch khi bạn thử khởi tạo một đối tượng *Enum*. *Enum* sẽ được tải tĩnh, như vậy nó là an toàn luồng. Phương thức *clone* trong *Enum* là *final* nên đảm bảo rằng các hằng số *Enum* không bao giờ được *clone*. *Enum* là kế thừa từ *serializable*, cơ chế tuần tự đảm bảo khởi tạo kép sẽ không bao giờ được tạo.

6.6.3 Khi nào sử dụng Singleton

- Khi cần chính xác một khởi tạo của một lớp và nó cần được truy cập từ client từ một điểm truy cập được biết đến.
- Khi khởi tạo gốc có thể được mở rộng bởi các lớp con, và client có thể sử dụng khởi tạo mở rộng mà không cần thay đổi code của họ.

6.7 Mẫu thiết kế OBSERVER

6.7.1 Mẫu Observer

Sport Lobby là một trang thể thao tuyệt vời. Nó bao phủ hầu hết mọi kiểu thể thao và cung cấp các tin tức mới nhất, thông tin, lịch các trận đấu, thông tin về từng cầu thủ và đội riêng biệt. Bây giờ họ lập kế hoạch cung cấp bình luận trực tiếp hoặc tỷ số các trận đấu như dịch vụ tin nhắn, nhưng chỉ cho các người sử dụng đặc biệt. Mục đích là tin nhắn SMS tỷ số trực tiếp, tình huống trận đấu và các sự kiện quan trọng sau khoảng thời gian ngắn. Như người sử dụng, bạn cần đăng ký cho gói này và khi có trận đấu diễn ra bạn sẽ nhận được SMS bình luận trực tiếp. Trang này cũng cung cấp lựa chọn hủy đăng ký khỏi gói khi bạn muốn.

Như người phát triển, Sport Lobby yêu cầu bạn cung cấp đặc tính mới này cho họ. Các bình luận viên sẽ ngồi ở buồng bình luận trong trận đấu và họ sẽ cập nhật bình luận trực tiếp cho các đối tượng bình luận. Như người phát triển, công việc của bạn là cung cấp bình luận cho những người sử dụng đã đăng ký bằng cách lấy chúng từ đối tượng bình luận khi nó đã sẵn sàng. Khi có cập nhật, hệ thống cần cập nhật cho các đối tượng đăng ký bằng cách gửi tin nhắn SMS cho họ.

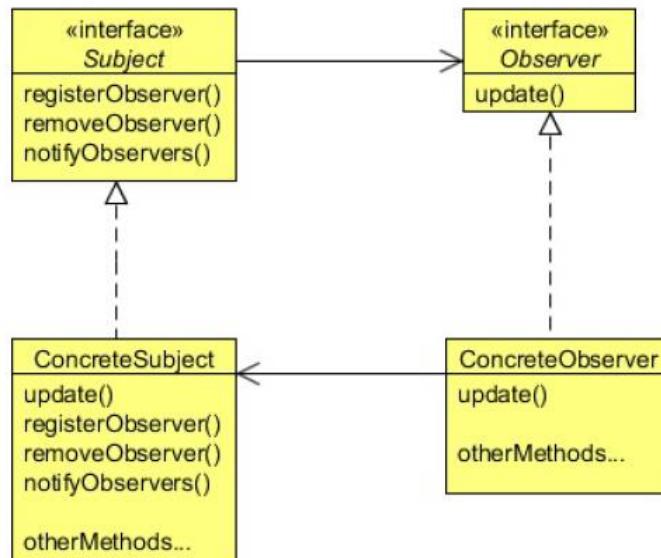
Tình huống này chỉ rõ ánh xạ một nhiều giữa trận đấu và người sử dụng như là có nhiều người sử dụng đăng ký đến một trận đấu. Mẫu thiết kế Observer là rất phù hợp cho tình huống này, ta sẽ xem xét mẫu này và sau đó tạo đặc tính đó cho Sport Lobby.

6.7.2 Mẫu Observer là gì

Mẫu Observer là kiểu mẫu hành vi mà liên quan đến việc giao trách nhiệm giữa các đối tượng. Các mẫu hành vi mô tả luồng điều khiển phức tạp mà khó theo dõi trong thời gian chạy. Chúng tách bạn khỏi luồng điều khiển và cho phép bạn chỉ tập trung vào cách đối tượng kết nối với nhau.

Mẫu Observer định nghĩa phụ thuộc một nhiều giữa các đối tượng sao cho khi một đối tượng thay đổi trạng thái, mọi cái phụ thuộc được thông báo và cập nhật tự động. Mẫu Observer mô tả các phụ thuộc này. Các đối tượng chính trong mẫu này là chủ đề *subject* và người quan sát *observer*, Chủ đề có thể có nhiều *observer* phụ thuộc. Mọi *observer* được thông báo khi chủ đề có thay đổi trong trạng thái của nó. Ngược lại, mỗi *observer* sẽ truy vấn chủ đề để đồng bộ trạng thái của nó với trạng thái của chủ đề.

Một cách khác để hiểu mẫu *Observer* là cách quan hệ *Publisher – Subscriber* (đăng lên và đăng ký) làm việc. Giả sử bạn đăng ký tạp chí về môn thể thao mà bạn ưu thích hay tạp chí mẫu. Khi có số mới được in, nó sẽ được phân phối cho bạn. Nếu bạn hủy đăng ký nó, khi bạn không muốn có tạp chí nữa, nó sẽ không được phân phối đến bạn. Nhưng nhà xuất bản vẫn tiếp tục làm việc như trước kia, vì còn nhiều người khác cũng đã đăng ký cho tạp chí đó.

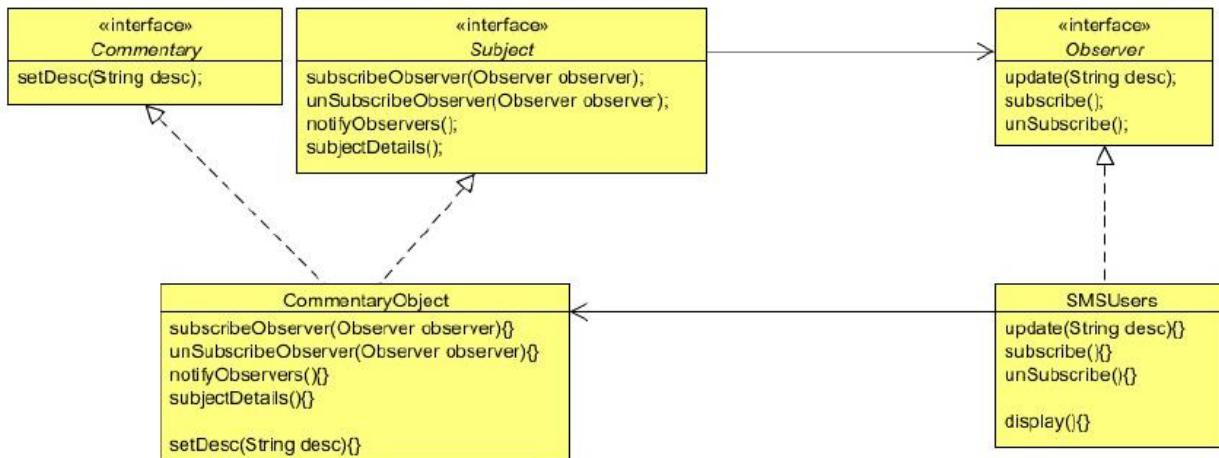


Có bốn thành phần trong mẫu *Observer*:

- Chủ đề, mà được sử dụng để đăng ký *observer*. Các đối tượng sử dụng giao diện này để đăng ký trở thành *observer* và cũng như tự hủy đăng ký khỏi *observer*.
- *Observer* định nghĩa giao diện *update* cho các đối tượng, mà muốn được thông báo các thay đổi của chủ đề. Mọi *observer* cần cài đặt giao diện *Observer*. Giao diện này có phương thức *update ()* mà sẽ được gọi khi trạng thái của chủ đề thay đổi.
- *ConcreteSubject* lưu giữ trạng thái quan tâm cho các đối tượng *ConcreteObserver*. Nó gửi thông báo cho các *observer* của nó khi trạng thái của nó thay đổi. Một chủ đề *Subject* cụ thể luôn cài đặt giao diện *Subject*. Phương thức *notifyObservers ()* được sử dụng để cập nhật mọi *observer* hiện tại khi trạng thái chủ đề thay đổi.
- *ConcreteObserver* duy trì tham chiếu đến đối tượng *ConcreteSubject* và cài đặt giao diện *Observer*. Mỗi *observer* đăng ký với chủ đề cụ thể để nhận các cập nhật.

6.7.3 Cài đặt mẫu *Observer*

Chúng ta xem sử dụng mẫu *Observer* như thế nào trong việc phát triển đặc tính yêu cầu của Sport Lobby. Ai đó sẽ cập nhật đối tượng chủ đề và công việc của bạn là cập nhật trạng thái đối tượng đã đăng ký với đối tượng chủ đề cụ thể đó. Do đó, khi có thay đổi trạng thái của đối tượng chủ đề cụ thể, mọi đối tượng phụ thuộc nó sẽ nhận được thông báo và sẽ cập nhật.



Dựa trên điều đó, trước hết ta tạo giao diện Subject. Trong giao diện Subject có ba phương thức chính và tùy chọn, nếu yêu cầu bạn có thể bổ sung một vài phương thức theo yêu cầu.

```

package com.javacodegeeks.patterns.observerpattern;

public interface Subject {
    public void subscribeObserver(Observer observer);
    public void unSubscribeObserver(Observer observer);
    public void notifyObservers();
    public String subjectDetails();
}
  
```

Ba phương thức chính trong giao diện Subject là:

- *subscribeObserver*, mà được sử dụng để đăng ký observer hoặc ta có thể nói đăng ký observer sao cho nếu có thay đổi trong trạng thái của chủ đề đó, mọi observer này sẽ nhận được thông báo.
- *unsubscriberObserver*, mà được sử dụng để hủy đăng ký observer sao cho nếu có thay đổi trong trạng thái của chủ đề đó, observer hủy đăng ký này sẽ không được nhận thông báo nữa.
- *notifyObservers*, phương thức này thông báo các observer đã đăng ký khi có thay đổi trong trạng thái của chủ đề đó.

Và tùy chọn, ở đây có thêm một phương thức *subjectDetails ()*, nó là phương thức hiển nhiên và cần thiết. Ở đây công việc là trả về chi tiết của chủ đề đó.

Bây giờ chúng ta sẽ xét giao diện Observer.

```

package com.javacodegeeks.patterns.observerpattern;

public interface Observer {
    public void update(String desc);
    public void subscribe();
    public void unSubscribe();
}
  
```

- *update (String desc)* là phương thức được gọi bởi chủ đề trên observer để thông báo mô tả *desc* đó, khi có thay đổi trong trạng thái của chủ đề.
- *subscriber ()* là phương thức được sử dụng để đăng ký bản thân nó với chủ đề đó.
- *unsubscribe ()* là phương thức được sử dụng để hủy đăng ký bản thân nó với chủ đề đó.

```
package com.javacodegeeks.patterns.observerpattern;

public interface Commentary {
    public void setDesc(String desc);
}
```

Giao diện trên được sử dụng bởi các bình luận viên để cập nhật các bình luận trực tiếp về đối tượng bình luận. Đây là giao diện tùy chọn chỉ để code tuân thủ theo nguyên tắc giao diện, chứ không liên quan gì đến mẫu Observer. Bạn cần áp dụng các nguyên lý hướng đối tượng với mẫu thiết kế mỗi khi áp dụng. Giao diện chỉ chứa một phương thức mà được sử dụng để thay đổi trạng thái của đối tượng chủ đề cụ thể.

```
package com.javacodegeeks.patterns.observerpattern;

import java.util.List;

public class CommentaryObject implements Subject,Commentary{

    private final List<Observer>observers;
    private String desc;
    private final String subjectDetails;

    public CommentaryObject(List<Observer>observers, String subjectDetails) {
        this.observers = observers;
        this.subjectDetails = subjectDetails;
    }
    @Override
    public void subscribeObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void unSubscribeObserver(Observer observer) {
        int index = observers.indexOf(observer);
        observers.remove(index);
    }

    @Override
    public void notifyObservers() {
        System.out.println();
        for(Observer observer : observers){
            observer.update(desc);
        }
    }

    @Override
    public void setDesc(String desc) {
        this.desc = desc;
        notifyObservers();
    }
    @Override
    public String subjectDetails() {
        return subjectDetails;
    }
}
```

Lớp trên làm việc như chủ đề cụ thể mà cài đặt giao diện Subject và cung cấp cài đặt của nó. Nó cũng lưu giữ tham chiếu đến các *observer* đã đăng ký chủ đề đó.

```

package com.javacodegeeks.patterns.observerpattern;

public class SMSUsers implements Observer{

    private final Subject subject;
    private String desc;
    private String userInfo;

    public SMSUsers(Subject subject, String userInfo) {
        if(subject==null){
            throw new IllegalArgumentException("No Publisher found.");
        }
        this.subject = subject;
        this.userInfo = userInfo;
    }

    @Override
    public void update(String desc) {
        this.desc = desc;
        display();
    }

    private void display(){
        System.out.println("[ "+userInfo+": "+desc);
    }

    @Override
    public void subscribe() {
        System.out.println("Subscribing "+userInfo+ " to "+subject.subjectDetails()+" ←
                           "...");
        this.subject.subscribeObserver(this);
        System.out.println("Subscribed successfully.");
    }

    @Override
    public void unSubscribe() {
        System.out.println("Unsubscribing "+userInfo+ " to "+subject.subjectDetails()+" ←
                           ()+" ...");
        this.subject.unSubscribeObserver(this);
        System.out.println("Unsubscribed successfully.");
    }
}

```

Lớp trên là *observer* cụ thể mà cài đặt giao diện *Observer*. Nó cũng lưu giữ tham chiếu đến chủ đề mà nó đăng ký vào và biến *userInfo* tùy chọn mà được sử dụng để hiện thông tin người sử dụng.

```

package com.javacodegeeks.patterns.observerpattern;

import java.util.ArrayList;

public class TestObserver {

    public static void main(String[] args) {
        Subject subject = new CommentaryObject(new ArrayList<Observer>(), "Soccer ←
                           Match [2014AUG24]");
        Observer observer = new SMSUsers(subject, "Adam Warner [New York]");
        observer.subscribe();
    }
}

```

```
        System.out.println();

        Observer observer2 = new SMSUsers(subject, "Tim Ronney [London]");
        observer2.subscribe();

        Commentary cObject = ((Commentary)subject);
        cObject.setDesc("Welcome to live Soccer match");
        cObject.setDesc("Current score 0-0");

        System.out.println();

        observer2.unSubscribe();

        System.out.println();

        cObject.setDesc("It's a goal!!!");
        cObject.setDesc("Current score 1-0");

        System.out.println();

        Observer observer3 = new SMSUsers(subject, "Marrie [Paris]");
        observer3.subscribe();

        System.out.println();

        cObject.setDesc("It's another goal!!!");
        cObject.setDesc("Half-time score 2-0");

    }

}
```

Ví dụ trên sẽ in ra:

```
Subscribing Adam Warner [New York] to Soccer Match [2014AUG24] ...
Subscribed successfully.

Subscribing Tim Ronney [London] to Soccer Match [2014AUG24] ...
Subscribed successfully.

[Adam Warner [New York]]: Welcome to live Soccer match
[Tim Ronney [London]]: Welcome to live Soccer match

[Adam Warner [New York]]: Current score 0-0
[Tim Ronney [London]]: Current score 0-0

Unsubscribing Tim Ronney [London] to Soccer Match [2014AUG24] ...
Unsubscribed successfully.

[Adam Warner [New York]]: It's a goal!!

[Adam Warner [New York]]: Current score 1-0

Subscribing Marrie [Paris] to Soccer Match [2014AUG24] ...
Subscribed successfully.

[Adam Warner [New York]]: It's another goal!!
[Marrie [Paris]]: It's another goal!!

[Adam Warner [New York]]: Half-time score 2-0
[Marrie [Paris]]: Half-time score 2-0
```

Như bạn thấy, đầu tiên hai người sử dụng đăng ký họ cho trận bóng và bắt đầu nhận được bình luận. Nhưng sau đó một người hủy đăng ký nó, nên người này không nhận được bình luận nữa. Sau đó người sử dụng khác đăng ký và bắt đầu nhận được bình luận.

Điều này xảy ra động không thay đổi code đã có và không chỉ vậy, giả sử công ty muốn truyền thông bình luận trên thư điện tử hoặc hãng khác muốn hợp tác với công ty này để truyền thông bình luận. Mọi việc bạn cần làm là tạo ra hai lớp mới `UserEmail` và `ColCompany` và bạn tạo họ là observer của chủ đề bằng cách cài đặt giao diện `Observe`. Cứ như vậy `Subject` đã biết observer của nó, nên sẽ cung cấp cập nhật.

6.7.4 Mẫu Observer có sẵn trong Java

Java có hỗ trợ xây sẵn cho mẫu Observer. Tổng quát nhất là mẫu giao diện `Observer` và lớp quan sát được `Observable` class trong gói `Java.util`. Nó hoàn toàn tương tự như giao diện `Subject` và `Observer` của chúng ta, nhưng cho bạn nhiều chức năng vượt qua giới hạn.

Giả sử thử cài đặt bài toán trên sử dụng mẫu Observer có sẵn trong Java:

```
package com.javacodegeeks.patterns.observerpattern;

import java.util.Observable;

public class CommentaryObjectObservable extends Observable implements Commentary {
    private String desc;
    private final String subjectDetails;

    public CommentaryObjectObservable(String subjectDetails) {
        this.subjectDetails = subjectDetails;
    }

    @Override
    public void setDesc(String desc) {
        this.desc = desc;
        setChanged();
        notifyObservers(desc);
    }

    public String subjectDetails() {
        return subjectDetails;
    }
}
```

Trên đây chúng ta mở rộng lớp `Observable` để làm cho lớp của chúng ta như chủ đề và lưu ý rằng lớp trên đây không giữ tham chiếu đến `observer`, nó được điều khiển bởi lớp cha, chính là lớp `Observable`. Tuy nhiên, chúng ta khai báo phương thức `setDesc` để thay đổi trạng thái của đối tượng như đã làm trong Ví dụ trước. Phương thức `setChanged` là phương thức từ lớp cha mà được sử dụng để đặt cờ thay đổi là true. Phương thức `notifyObservers` thông báo cho mọi obsever và sau đó gọi phương thức `clearChanged` để chỉ ra rằng đối tượng này sẽ không thay đổi lâu hơn nữa. Mỗi `observer` có phương thức `update` của nó với hai đối số: đối tượng kiểu `observable` và đối tượng `agr`.

```
package com.javacodegeeks.patterns.observerpattern;

import java.util.Observable;

public class SMSUsersObserver implements java.util.Observer{

    private String desc;
    private final String userInfo;
    private final Observable observable;

    public SMSUsersObserver(Observable observable, String userInfo){
        this.observable = observable;

        this.userInfo = userInfo;
    }

    public void subscribe() {
        System.out.println("Subscribing "+userInfo+ " to "+(( CommentatyObjectObservable) (observable)).subjectDetails()+" ...");
        this.observable.addObserver(this);
        System.out.println("Subscribed successfully.");
    }

    public void unSubscribe() {
        System.out.println("Unsubscribing "+userInfo+ " to "+(( CommentatyObjectObservable) (observable)).subjectDetails()+" ...");
        this.observable.deleteObserver(this);
        System.out.println("Unsubscribed successfully.");
    }

    @Override
    public void update(Observable o, Object arg) {
        desc = (String)arg;
        display();
    }

    private void display(){
        System.out.println("[ "+userInfo+" ]: "+desc);
    }
}
```

Ta bàn luận về một số phương thức chính.

Lớp trên đây cài đặt giao diện *Observer* mà có một phương thức chính *update*, mà được gọi khi chủ đề gọi phương thức *notifyObservers*. Phương thức *update* dùng đối tượng kiểu *Observable* và đối tượng *arg* như các tham số.

Phương thức *addObservable* được dùng để đăng ký *observer* cho chủ đề và phương thức *deleteObserver* được dùng để xóa *observer* khỏi danh sách của chủ đề.

Thử kiểm tra ví dụ này.

```
package com.javacodegeeks.patterns.observerpattern;

public class Test {

    public static void main(String[] args) {
        CommentaryObjectObservable obj = new CommentaryObjectObservable("Soccer ←
            Match [2014AUG24]");
        SMSUsersObserver observer = new SMSUsersObserver(obj, "Adam Warner [New ←
            York]");
        SMSUsersObserver observer2 = new SMSUsersObserver(obj, "Tim Ronney [London]" ←
            );
        observer.subscribe();
        observer2.subscribe();
        obj.setDesc("Welcome to live Soccer match");
        obj.setDesc("Current score 0-0");

        observer.unSubscribe();

        obj.setDesc("It's a goal!!!");
        obj.setDesc("Current score 1-0");
    }
}
```

Nó in ra kết quả sau:

```
Subscribing Adam Warner [New York] to Soccer Match [2014AUG24] ...
Subscribed successfully.
Subscribing Tim Ronney [London] to Soccer Match [2014AUG24] ...
Subscribed successfully.

[Tim Ronney [London]]: Welcome to live Soccer match
[Adam Warner [New York]]: Welcome to live Soccer match
[Tim Ronney [London]]: Current score 0-0
[Adam Warner [New York]]: Current score 0-0
Unsubscribing Adam Warner [New York] to Soccer Match [2014AUG24] ...
Unsubscribed successfully.
[Tim Ronney [London]]: It's a goal!!
[Tim Ronney [London]]: Current score 1-0
```

Lớp trên tạo chủ đề và hai *observer*. Phương thức *subscribe* của *observer* bổ sung nó vào danh sách *obsever* của chủ đề. Sau đó *setDesc* thay đổi trạng thái của chủ đề mà gọi phương thức *setChanged* để đặt cờ thay đổi là *true* và thông báo cho các *observer*. Kết quả thu được, phương thức *update* của *observer* được gọi mà phương thức trong lớp *display* sẽ hiển thị kết quả. Sau này, một *observer* rút khỏi danh sách *observer*. Theo đó các bình luận sau này không được cập nhật cho người đó.

Java cung cấp tiện ích sẵn cho mẫu *Observer*, nhưng nó có nhược điểm của nó. *Observable* là một lớp mà bạn kế thừa nó. Điều này nghĩa là bạn không thể bổ sung thêm hành vi *Observable* vào lớp có sẵn vì nó đã mở rộng một lớp cha khác. Điều này hạn chế khả năng tái sử dụng. Bạn ngay cả không thể tạo cài đặt của riêng bạn mà vẫn làm việc tốt với API *Observe* xây sẵn trong Java

Một số phương thức trong *API Observable* là *Protected*. Điều đó có nghĩa là bạn không thể gọi các phương thức như *setChange* trừ khi bạn có lớp con *Observable*. Và bạn không thể tạo khởi tạo của lớp *Observable* và kết hợp nó với các đối tượng của riêng bạn, mà bạn có lớp con. Thiết kế này vi phạm nguyên tắc thiết kế “kết hợp ưa thích trên kế thừa”.

6.7.5 Khi nào sử dụng mẫu Observer

Sử dụng mẫu Observer trong một số tình huống sau:

- Khi trùu tượng có hai khía cạnh, một phụ thuộc vào cái kia. Đóng gói các khía cạnh này trong các đối tượng tách bạch nhau cho phép bạn đa dạng và tái sử dụng chúng độc lập.
- Khi một thay đổi của đối tượng này yêu cầu thay đổi các đối tượng khác và bạn không biết có bao nhiêu đối tượng cần phải được thay đổi.
- Khi một đối tượng có thể thông báo cho các đối tượng khác mà không cần giả thiết về ai là các đối tượng này. Nói cách khác, bạn không muốn các đối tượng này quá gắn chặt nhau.

6.8 Mẫu thiết kế MEDIATOR

6.8.1 Mở đầu

Thế giới ngày nay vận hành trên phần mềm. Phần mềm chạy trong hầu hết mọi việc, nó không chỉ được sử dụng trong máy tính. Tivi thông minh, điện thoại di động, máy giặt đều có phần mềm nhúng mà vận hành máy đó.

Một công ty điện tử lớn có yêu cầu bạn phát triển đoạn phần mềm để vận hành máy giặt hoàn toàn tự động mới của họ. Công ty này cung cấp cho bạn đặc tả phần cứng và hiểu biết về cách làm việc của máy đó. Trong đặc tả này, họ đã cung cấp cho bạn các chương trình giặt khác nhau mà máy hỗ trợ. Họ muốn tạo ra một máy giặt hoàn toàn tự động, mà sẽ yêu cầu hầu như 0% sự tương tác của con người. Người sử dụng chỉ cần kết nối máy với điện và ống nước, bỏ quần áo vào, đặt kiểu quần áo trong máy như vải bông, lụa hay bông chéo, và đồ bột giặt, nước xả vải vào các khay tương ứng và ấn nút Start.

Máy này sẽ thông minh đến mức tự đổ nước vào trống, đủ theo yêu cầu. Nó cần điều chỉnh nhiệt độ nước bằng cách tự động bật máy làm nóng tùy theo kiểu vải trong đó. Nó sẽ khởi động động cơ và quay trống đủ dùng như đòi hỏi, vắt theo yêu cầu của vải, sử dụng nước xà phòng và xả vải theo trọng lượng và kiểu vải.

Như người phát triển hướng đối tượng, bạn bắt đầu phân tích và phân loại các đối tượng, các lớp và các quan hệ. Giả sử kiểm tra một số lớp và phần quan trọng của hệ thống. Trước hết lớp *Machine* mà có trống. Rồi lớp trống *Drum*, và cũng có lớp lò đun nóng *Heater*, và cảm ứng *Sensor* để đo nhiệt độ và lớp động cơ *Motor*.Thêm vào đó, máy có vòi *valve* để kiểm soát lượng nước và xà phòng, nước xả vải.

Các lớp này có quan hệ rất phức tạp với nhau và các quan hệ cũng rất phong phú. Lưu ý rằng hiện tại chúng ta đang xét mức độ trừu tượng cao của máy. Nếu chúng ta thử thiết kế nó mà không giữ các nguyên lý và mẫu hướng đối tượng trong suy nghĩ, thì thiết kế ban đầu là rất gắn chặt nhau và khó bảo trì. Điều này xảy ra, vì các lớp trên cần phải liên lạc với nhau để hoàn thành công việc. Như trong ví dụ, lớp *Machine* cần yêu cầu lớp *Valve* để mở vòi, hoặc lớp *Motor* cần quay trống *Drum* ở tốc độ theo chế độ chương trình giặt (mà được đặt bởi loại vải trong máy). Một kiểu vải yêu cầu xả vải hoặc xà phòng trong khi kiểu khác thì không hoặc nhiệt độ được đặt tùy theo kiểu vải.

Nếu chúng ta cho phép các lớp trao đổi trực tiếp với nhau, như là bằng cách cung cấp tham chiếu, thiết kế đó sẽ trở nên quá gắn chặt và khó bảo trì. Nó sẽ trở nên khó thay đổi một lớp mà không ảnh hưởng đến lớp khác. Ngay cả tồi tệ hơn là quan hệ giữa các lớp đa dạng theo các chương trình giặt khác nhau, như nhiệt độ khác nhau cho các kiểu vải khác nhau. Như vậy các lớp này không thể tái sử dụng. Cũng tồi tệ hơn, để hỗ trợ mọi chương trình máy giặt chúng ta cần đặt lệnh điều khiển kiểu *if -else* trong *code* mà làm cho *code* phức tạp hơn và khó bảo trì.

Để tách các đối tượng này ra khỏi nhau chúng ta cần một người trung gian mediator, mà sẽ trao đổi với đối tượng này thay mặt đối tượng khác, do đó cung cấp sự gắn kết lỏng lẻo giữa chúng. Một đối tượng chỉ cần biết về trung gian mediator, và thực hiện các thao tác trên nó. *Mediator* này sẽ thực hiện các thao tác theo đối tượng bên dưới yêu cầu để hoàn thành công việc.

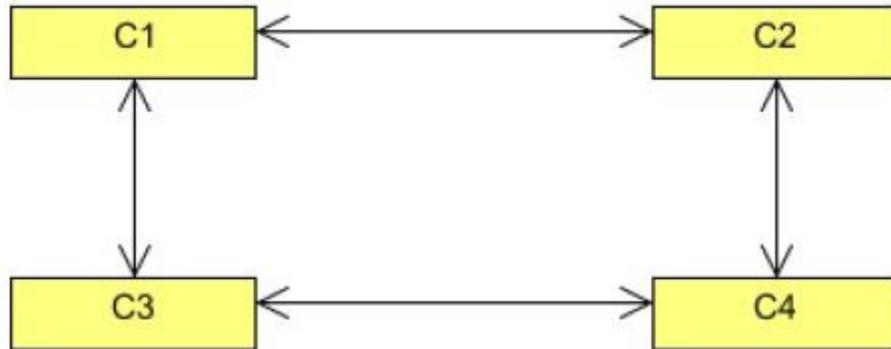
Mẫu Mediator là phù hợp nhất cho điều này, nhưng trước khi cài đặt nó để giải bài toán này, chúng ta sẽ xem xét mẫu thiết kế *Mediator*.

6.8.2 Mẫu thiết kế Mediator là gì

Mẫu *Mediator* định nghĩa một đối tượng mà đóng gói việc một tập các đối tượng tương tác với nhau như thế nào. *Mediator* làm cho gắn kết lỏng lẻo nhờ giữ các đối tượng không tham chiếu đến nhau tường minh, và nó cho phép bạn đa dạng tương tác của chúng một cách độc lập.

Thay vì tương tác trực tiếp với nhau, các đối tượng yêu cầu *Mediator* tương tác thay chúng mà cho kết quả là tính tái sử dụng và gắn kết lỏng lẻo. Nó đóng gói tương tác giữa các đối tượng và làm cho chúng độc lập với nhau. Điều này cho phép chúng đa dạng tương tác của chúng với các đối tượng khác theo các cách hoàn toàn khác nhau bằng cách cài đặt *mediator* khác. Một *Mediator* giúp giảm độ phức tạp của các lớp. Một đối tượng sẽ không biết chi tiết về làm sao tương tác với các đối tượng khác. Sự gắn kết giữa các đối tượng đi từ gắn chặt nhau tới gắn kết lỏng lẻo và linh hoạt.

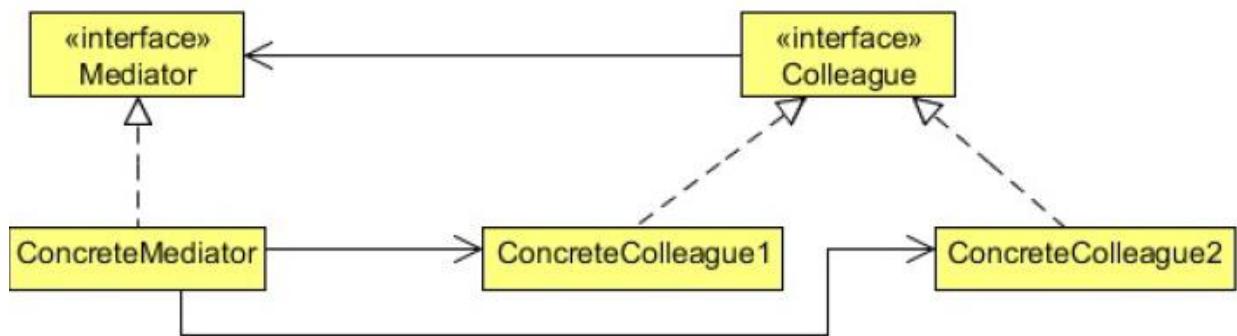
Trước khi dùng *Mediator*, tương tác giữa các lớp trông giống như chúa các tham chiếu đến nhau.



Bây giờ, sau khi cài đặt *Mediator* tương tác giữa các lớp trông như sau, chỉ chúa tham chiếu đến *mediator*.

Mẫu thiết kế *Mediator* cần phải là lựa chọn đầu tiên của bạn bất cứ lúc nào bạn có tập các đối tượng mà gắn chặt với nhau. Nếu mỗi một trong số một dãy các đối tượng cần phải biết chi tiết bên trong về đối tượng khác và duy trì quan hệ này trở thành một vấn đề, hãy nghĩ đến *Mediator*. Sử dụng *Mediator* có nghĩa là code tương tác cần phải nằm ở chỉ một chỗ, và nó làm cho bảo trì dễ hơn. Sử dụng *Mediator* có thể che giấu một vấn đề nghiêm trọng hơn. Nếu bạn có nhiều đối tượng mà quá gắn chặt nhau, đóng gói của bạn có thể bị lỗi. Nó là lúc bạn phải nghĩ lại bạn tách bài toán của bạn thành các đối tượng như thế nào.

Xét cấu trúc hình thức hơn của *Mediator*:



Các lớp mà giữ tham chiếu của *Mediator* được gọi là các đồng nghiệp *colleagues*. Các thành phần chính của mẫu *Mediator* là

- *Mediator*: định nghĩa giao diện trao đổi với các đối tượng *Colleague*.
- *ConcreteMediator*: cài đặt hành vi hợp tác bằng cách điều phối các đối tượng *Colleague*. Nó cũng biết và duy trì các *Colleague* của nó
- *Colleague*: giao diện chung *Colleague*
- *ConcreteColleague*: mỗi lớp *Colleague* biết đối tượng *Mediator*. Mỗi *colleague* trao đổi với *mediator* của nó khi nó muốn trao đổi với các *colleague* khác.

6.8.3 Cài đặt mẫu *Mediator*

Bây giờ chúng ta sẽ thấy mẫu *Mediator* sẽ là cho thiết kế máy giặt tốt hơn, tái sử dụng được, dễ bảo trì và gắn kết lỏng lẻo như thế nào.

```

package com.javacodegeeks.patterns.mediatorpattern;

public interface MachineMediator {

    public void start();
    public void wash();
    public void open();
    public void closed();
    public void on();
    public void off();
    public boolean checkTemperature(int temp);
}

```

MachineMediator là giao diện mà hoạt động như *mediator* tổng quan. Giao diện này chứa lời gọi thao tác bởi một đối tượng đến đối tượng khác.

```
package com.javacodegeeks.patterns.mediatorpattern;

public interface Colleague {

    public void setMediator(MachineMediator mediator);

}
```

Giao diện *Colleague* có một phương thức để đặt *mediator* cho lớp *colleague* cụ thể.

```
package com.javacodegeeks.patterns.mediatorpattern;

public class Button implements Colleague {

    private MachineMediator mediator;

    @Override
    public void setMediator(MachineMediator mediator) {
        this.mediator = mediator;
    }

    public void press(){
        System.out.println("Button pressed.");
        mediator.start();
    }

}
```

Lớp *Button* trên đây là một lớp *colleague* cụ thể mà giữ tham chiếu đến *mediator*. Người sử dụng nhấn nút button mà sẽ gọi phương thức *press()* của lớp này và đến lượt nó sẽ gọi phương thức *start()* của lớp *mediator* cụ thể. Phương thức *start()* của *mediator* sẽ gọi phương thức *start()* của lớp máy thay mặt cho lớp *Button*.

Ta sẽ thấy cấu trúc của lớp *Mediator*. Nhưng trước hết ta xem các lớp *Colleague* cụ thể khác.

```
package com.javacodegeeks.patterns.mediatorpattern;

public class Machine implements Colleague {

    private MachineMediator mediator;

    @Override
    public void setMediator(MachineMediator mediator) {
        this.mediator = mediator;
    }

    public void start(){
        mediator.open();
    }

    public void wash(){
        mediator.wash();
    }

}
```

Lớp *Machine* trên mà giữ tham chiếu đến *mediator* có phương thức *start ()* mà được gọi đến khi nhấn nút của *button* bởi lớp *mediator* như mô tả bên trên. Phương thức *open ()* của *mediator* mà đến lượt gọi phương thức *open ()* của lớp *Valve* để mở *valve* của máy *Machine*.

```
package com.javacodegeeks.patterns.mediatorpattern;

public class Valve implements Colleague {

    private MachineMediator mediator;

    @Override
    public void setMediator(MachineMediator mediator) {
        this.mediator = mediator;
    }

    public void open() {
        System.out.println("Valve is opened...");

        System.out.println("Filling water...");
        mediator.closed();
    }

    public void closed() {
        System.out.println("Valve is closed...");
        mediator.on();
    }
}
```

Lớp *Valve* có hai phương thức, phương thức *open ()* mà được gọi để mở vòi nước và khi nước đầy thì gọi phương thức đóng *closed ()*. Nhưng lưu ý rằng nó không gọi phương thức *closed ()* trực tiếp, nó gọi phương thức *closed ()* của *mediator*, mà nó sẽ gọi phương thức đóng của lớp này

Khi đóng *Valve* nó sẽ mở đun nước nóng *Heater*, nhưng lại được thực hiện bởi triệu gọi phương thức của *mediator* thay vì gọi trực tiếp phương thức của *heater*.

```
package com.javacodegeeks.patterns.mediatorpattern;

public class Heater implements Colleague {

    private MachineMediator mediator;

    @Override
    public void setMediator(MachineMediator mediator) {
        this.mediator = mediator;
    }
    public void on(int temp){
        System.out.println("Heater is on...");
        if(mediator.checkTemperature(temp)){
            System.out.println("Temperature is set to "+temp);
            mediator.off();
        }
    }

    public void off(){
        System.out.println("Heater is off...");
        mediator.wash();
    }
}
```

Phương thức *on ()* của *heater* mà sẽ đun nóng nước đến nhiệt độ theo yêu cầu. Nó sẽ kiểm tra nhiệt độ nếu nhiệt độ đạt như yêu cầu, nó sẽ gọi phương thức *off ()*. Việc kiểm tra nhiệt độ và chuyển *heater* sang *off* được thực hiện thông qua *mediator*.

Sau khi chuyển sang *off*, nó gọi phương thức *wash ()* của *Machine* thông qua *mediator* để bắt đầu giặt.

Khi khảng định bởi công ty, máy giặt có tập các chương trình giặt và phần mềm cần phải hỗ trợ tất cả các chương trình này. *Mediator* dưới đây là *mediator* của một chương trình cho máy giặt. *Mediator* dưới đây được thiết lập như chương trình giặt cho vải bông cottons, vì vậy các tham số như nhiệt độ, tốc độ quay của trống, mức xà phòng được đặt phù hợp. Chúng ta sẽ có các *mediator* khác nhau cho các chương trình giặt khác nhau mà không thay đổi các lớp *colleagues* đang tồn tại và như vậy cung cấp gắn kết lỏng lẻo và tái sử dụng. Mọi lớp *colleague* này có thể được tái sử dụng với các chương trình giặt khác của *Machine*.

```
package com.javacodegeeks.patterns.mediatorpattern;

public class CottonMediator implements MachineMediator{

    private final Machine machine;
    private final Heater heater;
    private final Motor motor;
    private final Sensor sensor;
    private final SoilRemoval soilRemoval;
    private final Valve valve;
```

```
public CottonMediator(Machine machine, Heater heater, Motor motor, Sensor sensor, ←
    SoilRemoval soilRemoval, Valve valve) {
    this.machine = machine;
    this.heater = heater;
    this.motor = motor;
    this.sensor = sensor;
    this.soilRemoval = soilRemoval;
    this.valve = valve;

    System.out.println("Setting up for COTTON program");
}
@Override
public void start() {
    machine.start();
}

@Override
public void wash() {
    motor.startMotor();
    motor.rotateDrum(700);
    System.out.println("Adding detergent");
    soilRemoval.low();
    System.out.println("Adding softener");
}

@Override
public void open() {
    valve.open();
}

@Override
public void closed() {
    valve.closed();
}

@Override
public void on() {
    heater.on(40);
}

@Override
public void off() {
    heater.off();
}

@Override
public boolean checkTemperature(int temp) {
    return sensor.checkTemperature(temp);
}

}
```

Lớp *CottonMediator* cài đặt giao diện *MachineMediator* và cung cấp các phương thức yêu cầu. Các phương thức này là các thao tác mà được thực hiện bởi các đối tượng *colleague* để hoàn thành công việc. Lớp *mediator* trên chỉ gọi phương thức của một đối tượng *colleague* thay mặt một đối tượng *colleague* khác để đạt được điều đó.

Còn có một số lớp hỗ trợ khác:

```
package com.javacodegeeks.patterns.mediatorpattern;

public class Sensor {

    public boolean checkTemperature(int temp) {
        System.out.println("Temperature reached "+temp+" *C");
        return true;
    }
}
```

Lớp *Sensor* được sử dụng bởi lớp *Heater* để kiểm tra nhiệt độ.

```
package com.javacodegeeks.patterns.mediatorpattern;

public class SoilRemoval {

    public void low() {
        System.out.println("Setting Soil Removal to low");
    }

    public void medium() {
        System.out.println("Setting Soil Removal to medium");
    }

    public void high() {
        System.out.println("Setting Soil Removal to high");
    }
}
```

Lớp *SollRemoval* được sử dụng bởi lớp *Machine*.

Để cảm thấy sự ưu việt và sức mạnh của mẫu *Mediator*, chúng ta tạo một *mediator* khác mà cần cho chương trình giặt vải bông chéo. Nay giờ chúng ta cần tạo một *mediator* mới và đặt các qui tắc giặt vải bông chéo: các qui tắc như nhiệt độ, và tốc độ mà trống sẽ quay, có cần xả vải hay không, lượng xà phòng. Chúng ta không cần thay đổi gì trong cấu trúc đang có. Không có lệnh “*if-then*” nào được dùng, những thứ mà làm tăng độ phức tạp.

```
package com.javacodegeeks.patterns.mediatorpattern;

public class DenimMediator implements MachineMediator{

    private final Machine machine;
    private final Heater heater;
    private final Motor motor;
    private final Sensor sensor;
    private final SoilRemoval soilRemoval;
    private final Valve valve;

    public DenimMediator(Machine machine, Heater heater, Motor motor, Sensor sensor, ←
        SoilRemoval soilRemoval, Valve valve){
        this.machine = machine;
        this.heater = heater;
        this.motor = motor;
        this.sensor = sensor;
        this.soilRemoval = soilRemoval;
        this.valve = valve;

        System.out.println("Setting up for DENIM program");
    }

    @Override
    public void start() {
        machine.start();
    }

    @Override

    public void wash() {
        motor.startMotor();
        motor.rotateDrum(1400);
        System.out.println("Adding detergent");
        soilRemoval.medium();
        System.out.println("Adding softener");
    }

    @Override
    public void open() {
        valve.open();
    }

    @Override
    public void closed() {
        valve.closed();
    }

    @Override
    public void on() {
        heater.on(30);
    }

    @Override
    public void off() {
        heater.off();
    }

    @Override
    public boolean checkTemperature(int temp) {
        return sensor.checkTemperature(temp);
    }
}
```

Bạn cần thấy rõ ràng các sự khác biệt giữa hai lớp mediator. Có nhiệt độ khác, tốc độ quay khác và không cần nước xả vải khi giặt vải bằng chéo.

Bây giờ ta kiểm tra các mediator.

```
package com.javacodegeeks.patterns.mediatorpattern;

public class TestMediator {

    public static void main(String[] args) {
        MachineMediator mediator = null;
        Sensor sensor = new Sensor();
        SoilRemoval soilRemoval = new SoilRemoval();
        Motor motor = new Motor();
        Machine machine = new Machine();
        Heater heater = new Heater();
        Valve valve = new Valve();
        Button button = new Button();

        mediator = new CottonMediator(machine, heater, motor, sensor, soilRemoval, ←
            valve);

        button.setMediator(mediator);
        machine.setMediator(mediator);
        heater.setMediator(mediator);
        valve.setMediator(mediator);

        button.press();
    }

    mediator = new DenimMediator(machine, heater, motor, sensor, soilRemoval, ←
        valve);

    button.setMediator(mediator);
    machine.setMediator(mediator);
    heater.setMediator(mediator);
    valve.setMediator(mediator);

    button.press();
}

}
```

Chương trình trên sẽ có kết quả đầu ra như sau:

```
Setting up for COTTON program
Button pressed.
Valve is opened...
Filling water...
Valve is closed...
Heater is on...
Temperature reached 40 C
Temperature is set to 40
Heater is off...
Start motor...
Rotating drum at 700 rpm.
Adding detergent
Setting Soil Removal to low
Adding softener

Setting up for DENIM program
Button pressed.
Valve is opened...
Filling water...
Valve is closed...
Heater is on...
Temperature reached 30 C
Temperature is set to 30
Heater is off...
Start motor...
Rotating drum at 1400 rpm.
Adding detergent
Setting Soil Removal to medium
No softener is required
```

Trong lớp trên, ta tạo các đối tượng như yêu cầu, các *mediator* (hoặc ta có thể nói là các chương trình giặt khác nhau), sau đó ta đặt các chương trình giặt cho các *colleague* và ngược lại, và ta gọi phương thức *start ()* trên đối tượng button để cho bắt đầu chạy máy giặt. Những cái còn lại được làm tự động không cần bắt cứ sự tương tác nào của con người.

Lưu ý rằng, khi làm việc với chương trình giặt khác, một *mediator* khác được thiết lập và mọi thứ vẫn như cũ. Bạn có thể thấy sự khác nhau từ kết quả in ra.

6.8.4 Khi nào sử dụng mẫu Mediator

- Một tập các đối tượng trao đổi theo các cách hoàn toàn xác định, nhưng phức tạp. Các sự phụ thuộc lẫn nhau là không có cấu trúc và khó hiểu.
- Tái sử dụng một đối tượng là khó bởi vì nó tham chiếu đến và trao đổi với rất nhiều đối tượng khác.
- Một hành vi mà phân tán giữa một số lớp cần phải được điều chỉnh không cần nhiều lớp con.

6.9 Mẫu thiết kế PROXY

6.9.1 Mở đầu

Trong bài này chúng ta sẽ bàn luận về một mẫu cấu trúc được gọi là mẫu Proxy. Mẫu Proxy cung cấp một sự thỏa thuận hay một chỗ để cho một đối tượng khác kiểm soát truy cập đến nó.

Mẫu Proxy có nhiều phương án khác nhau. Một số phương án quan trọng là: Remote Proxy, Virtual Proxy và Protection Proxy. Trong bài này chúng ta sẽ biết nhiều hơn về các biến thể này và chúng ta sẽ cài đặt mỗi biến thể đó trong Java. Nhưng trước khi làm điều đó, chúng ta sẽ xem xét kỹ hơn về mẫu Proxy nói chung.

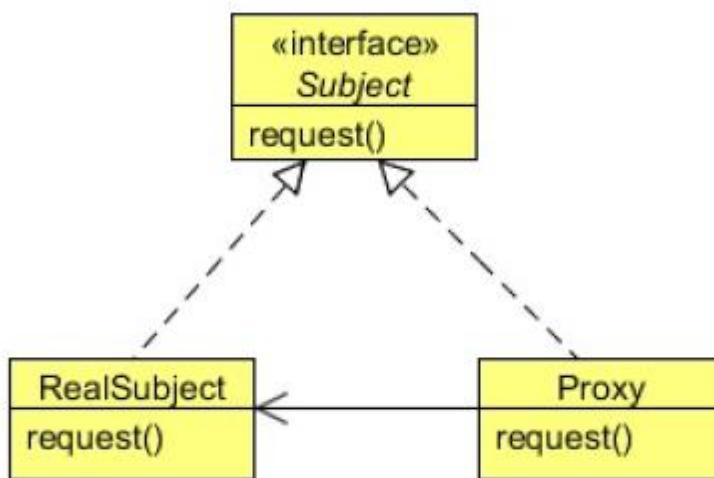
6.9.2 Mẫu Proxy là gì

Mẫu Proxy được sử dụng để tạo một đối tượng đại diện mà kiểm soát truy cập đến một đối tượng khác, mà có thể ở xa, tốn kém khi tạo hoặc cần phải được an toàn.

Một lý do để kiểm soát truy cập là trì hoãn trả giá đầy đủ để tạo và khởi tạo ra nó cho đến khi chúng ta thực sự cần sử dụng nó. Lý do khác có thể là hoạt động như một đại diện địa phương cho một đối tượng mà sống trong máy ảo Java khác. Proxy có thể rất hữu ích trong việc kiểm soát truy cập đến đối tượng gốc, đặc biệt khi các đối tượng này cần có các quyền truy cập khác nhau.

Trong mẫu Proxy, client không nói trực tiếp với đối tượng gốc, nó ủy quyền lời gọi cho đối tượng Proxy mà sẽ gọi các phương thức của đối tượng gốc. Điểm quan trọng là client không biết về proxy, proxy hoạt động như đối tượng gốc đối với client. Nhưng có nhiều biến thể cho cách tiếp cận này mà chúng ta sẽ thấy.

Bây giờ chúng ta xem cấu trúc của mẫu proxy và các thành phần quan trọng của nó.



- Proxy: 1a. Duy trì một tham chiếu mà cho phép proxy truy cập đến đối tượng thực. Proxy có thể tham chiếu đến một Subject, nếu các giao diện của RealSubject và Subject là như nhau. 1b. Cung cấp giao diện đồng nhất với Subject sao cho proxy có thể thay thế cho real subject. 1c, Kiểm soát truy cập đến real subject và có thể có trách nhiệm cho việc tạo và hủy nó.
- Subject: 2a. Xác định giao diện chung cho RealSubject và Proxy sao cho Proxy có thể được sử dụng ở bất cứ nơi nào mà RealSubject được cho phép.
- RealSubject: 3a. Xác định đối tượng thực tế mà proxy đại diện cho nó.

Có ba biến thể chính của mẫu Proxy:

- Remote Proxy cung cấp đại diện địa phương cho một đối tượng ở một không gian địa chỉ khác.
- Virtual Proxy tạo các đối tượng đắt giá theo yêu cầu.
- Protection Proxy kiểm soát truy cập đến đối tượng gốc. Protection Proxy là hữu ích khi các đối tượng có các quyền truy cập khác nhau.

Chúng ta sẽ bàn luận các proxy này trong từng mục sau.

6.9.3 Remote Proxy

Có một công ty Pizza, mà có một số cửa hàng ở các vị trí khác nhau. Người chủ công ty nhận báo cáo hàng ngày do các nhân viên của công ty từ các hàng gửi về. Ứng dụng hiện tại hỗ trợ công ty Pizza là ứng dụng trên desktop, chứ không phải ứng dụng Web. Vì vậy, chủ cửa hàng phải yêu cầu nhân viên của anh ta sinh ra báo cáo và gửi về. Nhưng bây giờ ông chủ muốn tự sinh và kiểm tra báo cáo, sao cho anh ta có thể sinh ra báo cáo bất cứ lúc nào mà không cần ai trợ giúp. Ông chủ muốn bạn phát triển ứng dụng đó cho ông.

Vấn đề ở đây là mọi ứng dụng đều chạy trên máy Java ảo (JVM) tương ứng của chúng. Các máy ảo Java và ứng dụng kiểm tra báo cáo (mà chúng ta sẽ thiết kế bây giờ) cần phải chạy trên hệ thống cục bộ của ông chủ. Đối tượng yêu cầu để sinh ra báo cáo không cần tồn tại trong JVM của hệ thống ông chủ và bạn không cần gọi trực tiếp cho đối tượng ở xa.

Remote Proxy được sử dụng để giải quyết vấn đề này. Chúng ta biết rằng báo cáo được sinh bởi người sử dụng, như vậy có đối tượng mà được yêu cầu để sinh báo cáo. Mọi điều mà chúng ta cần là liên hệ với đối tượng này mà nằm trên máy ở xa để nhận được kết quả mà ta mong muốn. Remote Proxy hoạt động như đại diện địa phương cho máy ở xa. Một đối tượng ở xa là đối tượng mà sống trên heap của máy JVM khác. Bạn gọi các phương thức đến đối tượng cục bộ và nó sẽ chuyển lời gọi đến các đối tượng ở xa.

Đối tượng client của bạn đưa ra lời gọi phương thức ở xa. Nhưng nó gọi phương thức của đối tượng proxy trên heap cục bộ mà sẽ quản lý mọi chi tiết mức thấp của truyền thông mạng.

Java hỗ trợ truyền thông giữa hai đối tượng ở trên hai JVM khác nhau sử dụng RMI. RMI là *Remote Method Invocation* – triệu gọi phương thức từ xa mà được sử dụng để xây dựng đối tượng trơ giúp cho client và dịch vụ, mà tạo đối tượng trơ giúp client với cùng phương thức như một dịch vụ từ xa. Sử dụng RMI bạn không cần tự bạn viết bất cứ điều gì về mạng hoặc code vào ra. Với client của bạn, bạn gọi phương thức ở xa như lời gọi phương thức bình thường trên đối tượng chạy trên JVM cục bộ của client.

RMI cũng cung cấp hạ tầng thực thi để làm cho nó hoạt động, bao gồm cả tìm kiếm dịch vụ mà client có thể sử dụng để tìm và truy cập đối tượng ở xa. Có một khác biệt giữa lời gọi RMI và lời gọi phương thức cục bộ. Trợ giúp của client gửi lời gọi phương thức thông qua mạng, như vậy có đường mạng và vào ra mà được sử dụng trong lời gọi RMI.

Bây giờ ta sẽ xem code. Chúng ta có giao diện *ReportGenerator* và cài đặt cụ thể của nó *ReportGeneratorImpl* đã chạy trên JVM của địa điểm khác. Trước hết để tạo dịch vụ từ xa chúng ta cần thay đổi code.

Giao diện *ReportGenerator* bây giờ trông như sau:

```
package com.javacodegeeks.patterns.proxypattern.remoteproxy;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ReportGenerator extends Remote{

    public String generateDailyReport() throws RemoteException;
}
```

Đây là giao diện *Remote* định nghĩa các phương thức mà *client* có thể gọi từ xa. Đó là cái mà *client* sẽ sử dụng như kiểu lớp đối với dịch vụ của bạn. Cả hai *Stub* và dịch vụ thực tế sẽ cài đặt cái này. Phương thức trong giao diện trả về đối tượng kiểu String. Bạn có thể trả về bất cứ kiểu gì từ phương thức, đối tượng này sẽ được truyền qua mạng từ máy chủ dịch vụ ngược lại cho *client*, như vậy nó cần phải là *Serializable*. Cần lưu ý rằng mọi phương thức trong giao diện này cần đưa ra thông báo lỗi *RemoteException*.

```

package com.javacodegeeks.patterns.proxypattern.remoteproxy;

import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;

public class ReportGeneratorImpl extends UnicastRemoteObject implements ReportGenerator{

    private static final long serialVersionUID = 3107413009881629428L;

    protected ReportGeneratorImpl() throws RemoteException {
    }

    @Override
    public String generateDailyReport() throws RemoteException {
        StringBuilder sb = new StringBuilder();
        sb.append("*****Location X Daily Report*****" ←
            );
        sb.append("\n Location ID: 012");
        sb.append("\n Today's Date: "+new Date());
        sb.append("\n Total Pizza's Sell: 112");
        sb.append("\n Total Price: $2534");
        sb.append("\n Net Profit: $1985");
        sb.append("\n ←
            *****");
        return sb.toString();
    }

    public static void main(String[] args) {
        try {
            ReportGenerator reportGenerator = new ReportGeneratorImpl();
            Naming.rebind("PizzaCoRemoteGenerator", reportGenerator);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

Lớp trên là cài đặt từ xa mà sẽ làm việc trên thực tế. Nó là đối tượng mà *client* muốn gọi phương thức trên đó. Lớp này mở rộng *UnicastRemoteObject* để làm việc như đối tượng dịch vụ từ xa, đối tượng của bạn cần chức năng nào đó liên quan đến việc phải là *remote*. Một cách đơn giản nhất là mở rộng *UnicastRemoteObject* từ gói *Java.rmi.server* và cho phép lớp này làm việc cho bạn.

Hàm tạo của lớp *UnicastRemoteObject* đưa ra báo lỗi *RemoteException*, nên bạn cần viết hàm tạo không đối số mà khai báo *RemoteException*.

Để làm cho dịch vụ từ xa sẵn sàng cho *client*, bạn cần đăng ký dịch vụ này với *RMI registry*. Bạn làm điều đó bằng cách khởi tạo nó và đặt nó vào *RMI registry*. Khi bạn đăng ký đối tượng cài đặt, hệ thống RMI thực tế sẽ đặt nó vào *Stub* trong *registry*, đây là cái *client* cần. Phương thức *Naming.rebind* được sử dụng để đăng ký đối tượng, nó có hai tham số, cái đầu là *string* để đặt tên cho dịch vụ và tham số kia lấy đối tượng mà sẽ gắn vào để *client* sử dụng dịch vụ.

Bây giờ, để tạo *stub* bạn cần chạy *rmic* trên lớp cài đặt *remote*. Công cụ *rmic* đi cùng bộ phát triển phần mềm Java, lấy cài đặt dịch vụ và tạo nên *stub* mới. Bạn cần mở dấu nhắc lệnh của bạn (*cmd*) và chạy *rmic* từ thư mục ở nơi có cài đặt từ xa của bạn.

Bước tiếp theo là chạy *rmiregistry*, mang đến một *terminal* và bắt đầu *registry*, chỉ bằng lệnh *rmiregistry*. Nhưng tin tưởng là bạn *start* nó từ thư mục mà truy cập đến các lớp của bạn.

Bước cuối cùng là *start* dịch vụ này bằng cách chạy cài đặt cụ thể của lớp ở xa, trong trường hợp này là *ReportGeneratorImpl*.

Khi đó bạn đã tạo và chạy dịch vụ. Bây giờ chúng ta xem *client* sẽ sử dụng nó như thế nào. Ứng dụng báo cáo cho ông chủ công ty Pizza sẽ sử dụng dịch vụ này để sinh ra và kiểm tra báo cáo. Chúng ta cần cung cấp giao diện *ReportGenerator* và *stub* cho *client* mà sẽ sử dụng dịch vụ này. Bạn đơn giản cần trao tay *stub* hoặc các lớp hoặc các giao diện mà cần trong dịch vụ này.

```
package com.javacodegeeks.patterns.proxypattern.remoteproxy;

import java.rmi.Naming;

public class ReportGeneratorClient {

    public static void main(String[] args) {
        new ReportGeneratorClient().generateReport();
    }

    public void generateReport(){
        try {
            ReportGenerator reportGenerator = (ReportGenerator)Naming.lookup("rmi://127.0.0.1/PizzaCoRemoteGenerator");
            System.out.println(reportGenerator.generateDailyReport());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Chương trình trên cho ra kết quả đầu ra như sau:

```
*****Location X Daily Report*****
Location ID: 012
Today's Date: Sun Sep 14 00:11:23 IST 2014
Total Pizza Sell: 112
Total Sale: $2534
Net Profit: $1985
*****
```

Lớp trên thực hiện tìm kiếm tên và truy vấn đối tượng mà được sử dụng để sinh ra báo cáo hàng ngày. Bạn cần cung cấp IP của máy chủ và tên được sử dụng để gắn kết với dịch vụ. Những điều còn lại trông như lời gọi phương thức bình thường.

Tóm lại, *Remote Proxy* hoạt động như đại diện địa phương cho đối tượng mà sống ở JVM khác. Một phương thức gọi trên *Proxy* sẽ cho kết quả lời gọi đó được truyền qua mạng, triệu gọi từ xa, và kết quả được trả lại *Proxy* và sau đó là *client*.

6.9.4 Virtual Proxy

Mẫu *Virtual Proxy* là kỹ thuật tiết kiệm bộ nhớ mà được đề xuất trì hoãn tạo đối tượng cho đến khi nó cần. Nó được sử dụng khi việc tạo một đối tượng là đắt theo nghĩa tốn bộ nhớ sử dụng và xử lý. Trong ứng dụng thông thường, các đối tượng khác nhau tạo nên các phân khác nhau của chức năng. Khi một ứng dụng bắt đầu, nó có thể không cần mọi đối tượng của nó phải sẵn sàng ngay tức thì. Trong trường hợp này, mẫu *Virtual Proxy* đề xuất trì hoãn tạo đối tượng cho đến khi nó cần bởi ứng dụng. Một đối tượng mà được tạo, khi lần đầu được tham chiếu trong ứng dụng và cùng khởi tạo đó được tái sử dụng từ thời điểm đó trở đi. Ưu điểm của cách tiếp cận này là thời gian khởi tạo ứng dụng nhanh hơn, vì nó không yêu cầu được tạo và tải mọi đối tượng ứng dụng ngay.

Giả sử có đối tượng *Company* trong ứng dụng của bạn và đối tượng này chứa danh sách mọi nhân viên của Công ty trong đối tượng *ContactList*. Đây có thể là hàng vài ngàn nhân viên của công ty. Tải đối tượng *Company* từ cơ sở dữ liệu với danh sách các nhân viên của nó trong đối tượng *ContactList* có thể rất tốn thời gian. Trong một số trường hợp ngay cả bạn không cần đến danh sách nhân viên, nhưng bạn buộc phải chờ cho đến khi công ty và danh sách nhân viên của nó được tải vào bộ nhớ.

Một cách để tiết kiệm thời gian và bộ nhớ là tránh tải các đối tượng làm việc cho đến khi chúng được yêu cầu, và điều này được thực hiện sử dụng *Virtual Proxy*. Kỹ thuật này được biết đến là Tải Lười khi bạn đẩy dữ liệu vào chỉ khi chúng được yêu cầu.

```
package com.javacodegeeks.patterns.proxypattern.virtualproxy;

public class Company {

    private String companyName;
    private String companyAddress;
    private String companyContactNo;
    private ContactList contactList ;

    public Company(String companyName, String companyAddress, String companyContactNo, ↪
        ContactList contactList) {
        this.companyName = companyName;
        this.companyAddress = companyAddress;
        this.companyContactNo = companyContactNo;
        this.contactList = contactList;

        System.out.println("Company object created...");
    }

    public String getCompanyName() {
        return companyName;
    }

    public String getCompanyAddress() {
        return companyAddress;
    }

    public String getCompanyContactNo() {
        return companyContactNo;
    }

    public ContactList getContactList() {
        return contactList;
    }

}
```

Lớp *Company* trên đây có tham chiếu đến giao diện *ContactList* mà đối tượng thực tế đó sẽ được tải khi có yêu cầu gọi phương thức *getContactList()*.

```
package com.javacodegeeks.patterns.proxypattern.virtualproxy;

import java.util.List;

public interface ContactList {
    public List<Employee> getEmployeeList();
}
```

Giao diện *ContactList* chỉ chứa một phương thức *getEmployeeList()* mà được sử dụng để nhận danh sách các nhân viên của Công ty.

```
package com.javacodegeeks.patterns.proxypattern.virtualproxy;

import java.util.ArrayList;
import java.util.List;

public class ContactListImpl implements ContactList {

    @Override
    public List<Employee> getEmployeeList() {
        return getEmpList();
    }

    private static List<Employee> getEmpList() {
        List<Employee> empList = new ArrayList<Employee>(5);
        empList.add(new Employee("Employee A", 2565.55, "SE"));
        empList.add(new Employee("Employee B", 22574, "Manager"));
        empList.add(new Employee("Employee C", 3256.77, "SSE"));
        empList.add(new Employee("Employee D", 4875.54, "SSE"));
        empList.add(new Employee("Employee E", 2847.01, "SE"));
        return empList;
    }

}
```

Lớp trên sẽ tạo ra đối tượng *ContactList* thực tế mà trả về danh sách các nhân viên của công ty. Đối tượng này sẽ được tải theo yêu cầu và chỉ khi nó được đòi hỏi.

```
package com.javacodegeeks.patterns.proxypattern.virtualproxy;

import java.util.List;

public class ContactListProxyImpl implements ContactList{

    private ContactList contactList;

    @Override
    public List<Employee> getEmployeeList() {
        if(contactList == null){
            System.out.println("Creating contact list and fetching list of ←
                employees...");
            contactList = new ContactListImpl();
        }
        return contactList.getEmployeeList();
    }

}
```

ContactListProxyImpl là lớp proxy mà cũng cài đặt *ContactList* và giữ tham chiếu đến đối tượng *ContactList* thực tế. Trong cài đặt của phương thức *getEmployeeList ()* nó sẽ kiểm tra nếu tham chiếu *ContactList* là null, thì nó sẽ tạo đối tượng *ContactList* thực tế và sau đó triệu gọi phương thức *getEmployeeList ()* trên nó để nhận được danh sách nhân viên.

Lớp *Employee* trông như sau:

```
package com.javacodegeeks.patterns.proxypattern.virtualproxy;

public class Employee {

    private String employeeName;

    private double employeeSalary;
    private String employeeDesignation;

    public Employee(String employeeName, double employeeSalary, String employeeDesignation) {
        this.employeeName = employeeName;
        this.employeeSalary = employeeSalary;
        this.employeeDesignation = employeeDesignation;
    }

    public String getEmployeeName() {
        return employeeName;
    }

    public double getEmployeeSalary() {
        return employeeSalary;
    }

    public String getEmployeeDesignation() {
        return employeeDesignation;
    }

    public String toString(){
        return "Employee Name: "+employeeName+", EmployeeDesignation: "+employeeDesignation+", Employee Salary: "+employeeSalary;
    }
}

package com.javacodegeeks.patterns.proxypattern.virtualproxy;

import java.util.List;

public class TestVirtualProxy {

    public static void main(String[] args) {
        ContactList contactList = new ContactListProxyImpl();
        Company company = new Company("ABC Company", "India", "+91-011-28458965", contactList);

        System.out.println("Company Name: "+company.getCompanyName());
        System.out.println("Company Address: "+company.getCompanyAddress());
        System.out.println("Company Contact No.: "+company.getCompanyContactNo());

        System.out.println("Requesting for contact list");

        contactList = company.getContactList();

        List<Employee> empList = contactList.getEmployeeList();
        for(Employee emp : empList){
            System.out.println(emp);
        }
    }
}
```

Chương trình trên sẽ cho ra kết quả:

```
Company object created...
Company Name: ABC Company
Company Address: India
Company Contact No.: +91-011-28458965

Requesting for contact list
Creating contact list and fetching list of employees...
Employee Name: Employee A, EmployeeDesignation: SE, Employee Salary: 2565.55
Employee Name: Employee B, EmployeeDesignation: Manager, Employee Salary: 22574.0
Employee Name: Employee C, EmployeeDesignation: SSE, Employee Salary: 3256.77
Employee Name: Employee D, EmployeeDesignation: SSE, Employee Salary: 4875.54
Employee Name: Employee E, EmployeeDesignation: SE, Employee Salary: 2847.01
```

Như bạn nhìn thấy trong đầu ra được sinh bởi *TestVirtualProxy*, trước hết chúng ta tạo ra đối tượng *Company* với đối tượng proxy *ContactList*. Tại thời điểm này, đối tượng *Company* giữ tham chiếu proxy, không phải tham chiếu của đối tượng *ContactList* thực tế, như vậy không có danh sách nhân viên được tải vào bộ nhớ. Chúng ta thực hiện mấy lời gọi đối tượng *Company* và sau đó yêu cầu danh sách nhân viên từ đối tượng *contact list proxy* sử dụng phương thức *getEmployeeList()*. Trong lời gọi phương thức này, đối tượng proxy tạo đối tượng *ContactList* thực tế và cung cấp danh sách các nhân viên.

6.9.5 Protection Proxy

Nhìn chung, các đối tượng trong một ứng dụng tương tác với nhau để thực hiện chức năng ứng dụng tổng thể. Hầu hết đối tượng ứng dụng là được truy cập chung cho mọi đối tượng khác trong ứng dụng. Đôi khi, nó có thể được yêu cầu hạn chế truy cập đối tượng cho tập hạn chế các đối tượng client dựa trên quyền truy cập của chúng. Khi một đối tượng thử truy cập đến đối tượng như vậy, *client* chỉ được truy cập đến dịch vụ được cung cấp bởi đối tượng, nếu client có thể cung cấp giấy xác thực đúng. Trong các trường hợp như vậy, một đối tượng tách biệt có thể được chỉ định với trách nhiệm kiểm chứng quyền truy cập của các đối tượng client khác nhau khi họ truy cập đến các đối tượng thực tế. Nói cách khác, mỗi client cần xác thực thành công với đối tượng chỉ định đó để được truy cập đến chức năng đối tượng thực tế. Đối tượng như vậy, mà với nó client cần để xác thực được truy cập đến đối tượng thực tế, có thể được tham chiếu như xác thực đối tượng mà được cài đặt sử dụng *Protection Proxy*.

Trở lại ứng dụng *ReportGenerator* mà chúng ta đã phát triển cho Công ty Pizza, ông chủ bây giờ yêu cầu chỉ có ông có thể sinh ra báo cáo hàng ngày. Không nhân viên nào khác có thể làm như vậy được.

Để cài đặt đặc tính an ninh đó, chúng ta sử dụng *Protection Proxy* mà kiểm tra nếu đối tượng mà thử sinh ra báo cáo có phải là ông chủ không, nếu đúng, thì báo cáo sẽ sinh ra, ngược lại thì không.

```
package com.javacodegeeks.patterns.proxypattern.protectionproxy;

public interface Staff {

    public boolean isOwner();
    public void setReportGenerator(ReportGeneratorProxy reportGenerator);
}
```

Giao diện *Staff* được sử dụng bởi các lớp *Owner* và *Employee* và giao diện này có hai phương thức: *isOwner ()* trả về biến Bool kiểm tra đối tượng gọi là owner hay không. Phương thức khác được sử dụng để đặt *ReportGeneratorProxy* mà được *Protection proxy* sử dụng để sinh ra báo cáo nếu *isOwner ()* trả về *true*.

```
package com.javacodegeeks.patterns.proxypattern.protectionproxy;

public class Employee implements Staff{

    private ReportGeneratorProxy reportGenerator;

    @Override
    public void setReportGenerator(ReportGeneratorProxy reportGenerator) {
        this.reportGenerator = reportGenerator;
    }

    @Override
    public boolean isOwner() {
        return false;
    }

    public String generateDailyReport() {
        try {
            return reportGenerator.generateDailyReport();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return "";
    }
}
```

Lớp *Employee* cài đặt giao diện *Staff*, vì phương thức *isOwner* của *Employee* trả về *false*. Phương thức *generateDailyReport ()* yêu cầu *ReportGenerator* sinh ra báo cáo hàng ngày.

```
package com.javacodegeeks.patterns.proxypattern.protectionproxy;

public class Owner implements Staff {

    private boolean isOwner=true;
    private ReportGeneratorProxy reportGenerator;

    @Override
    public void setReportGenerator(ReportGeneratorProxy reportGenerator) {
        this.reportGenerator = reportGenerator;
    }

    @Override
    public boolean isOwner(){
        return isOwner;
    }

    public String generateDailyReport() {
        try {
            return reportGenerator.generateDailyReport();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return "";
    }

}
```

Lớp *Owner* trông giống lớp *Employee*, nhưng sự khác biệt chỉ ở chỗ phương thức *isOwner* trả về true.

```
package com.javacodegeeks.patterns.proxypattern.protectionproxy;

public interface ReportGeneratorProxy {

    public String generateDailyReport();
}
```

ReportGeneratorProxy hoạt động như một *Protection Proxy* mà chỉ có một phương thức *generateDailyReport ()* mà được sử dụng để sinh báo cáo.

```
package com.javacodegeeks.patterns.proxypattern.protectionproxy;

import java.rmi.Naming;

import com.javacodegeeks.patterns.proxypattern.remoteproxy.ReportGenerator;

public class ReportGeneratorProtectionProxy implements ReportGeneratorProxy{
```

```
ReportGenerator reportGenerator;
Staff staff;

public ReportGeneratorProtectionProxy(Staff staff) {
    this.staff = staff;
}

@Override
public String generateDailyReport() {
    if(staff.isOwner()){
        ReportGenerator reportGenerator = null;
        try {
            reportGenerator = (ReportGenerator)Naming.lookup("rmi ↳
                ://127.0.0.1/PizzaCoRemoteGenerator");
            return reportGenerator.generateDailyReport();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return "";
    }
    else{
        return "Not Authorized to view report.";
    }
}
}
```

Lớp trên là cài đặt cụ thể của *ReportGeneratorProxy*, mà giữ tham chiếu đến giao diện *ReportGenerator* mà là proxy từ xa. Trong phương thức *generateDailyReport()*. Nó kiểm tra nếu *Staff* được tham chiếu đến *Owner*, thì yêu cầu *remote proxy* sinh ra báo cáo, ngược lại thì trả về xâu với thông báo “*Not Authorized to view report*”.

```
package com.javacodegeeks.patterns.proxypattern.protectionproxy;

public class TestProtectionProxy {

    public static void main(String[] args) {

        Owner owner = new Owner();
        ReportGeneratorProxy reportGenerator = new ReportGeneratorProtectionProxy( ↳
            owner);
        owner.setReportGenerator(reportGenerator);

        Employee employee = new Employee();
        reportGenerator = new ReportGeneratorProtectionProxy(employee);
        employee.setReportGenerator(reportGenerator);
        System.out.println("For owner:");
        System.out.println(owner.generateDailyReport());
        System.out.println("For employee:");
        System.out.println(employee.generateDailyReport());

    }
}
```

Chương trình trên cho ra kết quả:

```
For owner:  
*****Location X Daily Report*****  
Location ID: 012  
Today's Date: Sun Sep 14 13:28:12 IST 2014  
Total Pizza Sell: 112  
Total Sale: $2534
```

```
Net Profit: $1985  
*****  
For employee:  
Not Authorized to view report.
```

Đầu ra trên đây chỉ rõ rằng ông chủ có thể sinh ra báo cáo, nhân viên thì không. *Protection Proxy* bảo vệ truy cập sinh ra báo cáo và chỉ cho phép đối tượng có chủ quyền mới sinh được báo cáo.

6.9.6 Khi nào sử dụng mẫu Proxy

Proxy được áp dụng khi có nhu cầu một tham chiếu hay thay đổi hay triết lý hơn đến một đối tượng so với bình thường. Ở đây có một số tình huống chung ở đó mẫu *proxy* được áp dụng:

- *Remote Proxy* cung cấp đại diện địa phương cho một đối tượng ở không gian địa chỉ khác
- *Virtual Proxy* tạo các đối tượng đắt theo nhu cầu.
- *Protection proxy* kiểm soát truy cập đến đối tượng gốc. *Protection proxy* là hữu ích khi các đối tượng có các quyền truy cập khác nhau.

6.10 Mẫu thiết kế CHAIN OF RESPONSIBILITY

6.10.1 Mẫu Chain of Responsibility

Mẫu dây trách nhiệm là mẫu hành vi mà ở đó một nhóm các đối tượng được móc nối với nhau thành dây và mỗi trách nhiệm (hay một yêu cầu) được cung cấp để xử lý bởi nhóm đó. Nếu một đối tượng trong nhóm có thể xử lý một yêu cầu riêng biệt, nó làm điều đó và trả về phản hồi tương ứng. Ngược lại nó sẽ chuyển tiếp yêu cầu cho đối tượng tiếp theo trong nhóm.

Đối với kịch bản đời sống thực tế, để hiểu được mẫu này, giả sử bạn có một vấn đề cần giải quyết. Nếu bản thân bạn có thể xử lý nó, bạn sẽ làm điều đó, ngược lại bạn sẽ nói với bạn của bạn giải quyết nó. Nếu anh ta có thể giải quyết việc đó, anh ta sẽ làm hoặc anh ta sẽ chuyển tiếp nó cho người bạn khác. Vấn đề đó sẽ được chuyển tiếp cho đến khi nó được giải quyết bởi một người bạn hoặc mọi người bạn của bạn đều đã thấy bài toán, nhưng không ai có thể giải được, trong trường hợp này vấn đề vẫn còn chưa được giải quyết.

Xét một kịch bản thực tế. Công ty của bạn có một hợp đồng cung cấp ứng dụng phân tích cho một công ty y tế. Ứng dụng sẽ nói cho người sử dụng về một vấn đề sức khỏe cụ thể, lịch sử của nó, thuốc chữa, lời phỏng vấn của người bệnh, mọi thứ mà cần thiết để biết về nó. Để làm điều đó, công ty của bạn nhận số lượng dữ liệu rất lớn. Dữ liệu này có thể ở trong định dạng bất kỳ, nó có thể là text file, doc file, excel file, audio, video, bất cứ kiểu file gì mà bạn nghĩ có thể cần thiết ở đây.

Bây giờ công việc của bạn là bạn cần phát triển các bộ xử lý khác nhau để lưu các định dạng khác nhau của dữ liệu này. Chẳng hạn, bộ xử lý lưu file text sẽ không biết gì về việc lưu file mp3.

Để giải quyết vấn đề này bạn có thể sử dụng mẫu thiết kế *Chain of Responsibility*. Bạn có thể tạo các đối tượng khác nhau mà xử lý các định dạng khác nhau của dữ liệu và móc nối chúng với nhau. Khi yêu cầu được gửi đến một đối tượng đơn nào đó, nó sẽ kiểm tra xem có thể xử lý để lưu định dạng chuyên biệt đó không. Nếu nó có thể, thì nó sẽ xử lý, ngược lại nó sẽ chuyển tiếp cho đối tượng móc nối tiếp theo. Mẫu thiết kế này cũng phân tách người sử dụng ra khỏi đối tượng mà phục vụ yêu cầu đó, người sử dụng không quan tâm đối tượng nào thực tế đã phục vụ yêu cầu đó.

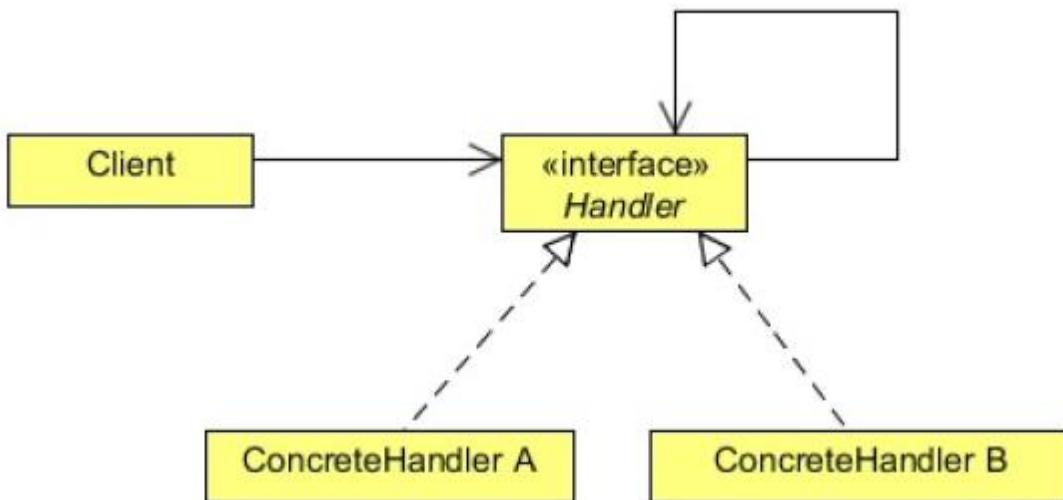
Trước khi giải bài toán đó, chúng ta sẽ tìm hiểu về mẫu thiết kế *Chain of Responsibility*.

6.10.2 Mẫu Chain of Responsibility là gì

Mục đích của mẫu này là tránh gắn kết giữa người gửi yêu cầu với người nhận bằng cách cho nhiều hơn một đối tượng cơ hội xử lý yêu cầu đó. Chúng ta móc nối các đối tượng nhận và truyền yêu cầu dọc theo dây móc nối đến khi một đối tượng xử lý được nó.

Mẫu này là tất cả về kết nối các đối tượng trong danh sách mốc nối, một thông báo sẽ đi dọc theo danh sách, nó được xử lý bởi đối tượng đầu tiên mà được thiết lập để làm việc với loại thông báo này.

Khi có nhiều hơn một đối tượng có thể xử lý hoặc thực hiện yêu cầu của *client*, mẫu đê xuất cho mỗi đối tượng một cơ hội để xử lý yêu cầu theo thứ tự tuần tự nào đó. Áp dụng mẫu này trong trường hợp như vậy, các bộ xử lý tiềm năng cần được sắp xếp vào thành một dãy, với mỗi đối tượng có tham chiếu đến đối tượng tiếp theo trong dãy. Đối tượng đầu tiên của dãy nhận được yêu cầu và quyết định sẽ xử lý hoặc chuyển tiếp cho đối tượng tiếp theo trong dãy. Yêu cầu sẽ đi qua mọi đối tượng trong dãy cái nọ sau cái kia cho đến khi nó được xử lý bởi một đối tượng trong dãy hoặc yêu cầu đạt đến cuối dãy mà không được xử lý.



Handler:

- Định nghĩa giao diện để xử lý yêu cầu
- (Tùy chọn) Cài đặt tham chiếu tiếp theo

Concrete Handler:

- Xử lý yêu cầu nếu nó có trách nhiệm
- Có thể truy cập tiếp theo của nó
- Nếu *ConcreteHandler* có thể xử lý yêu cầu, nó làm điều đó, ngược lại nó chuyển tiếp yêu cầu cho tiếp theo của nó.

Client

- Khởi tạo yêu cầu đến đối tượng *ConcreteHandler* trên chuỗi

Khi *client* đưa ra yêu cầu, yêu cầu được lan truyền trong chuỗi cho đến khi một đối tượng *ConcreteHandler* lãnh trách nhiệm xử lý nó.

6.10.3 Cài đặt chuỗi trách nhiệm

Để cài đặt *Chain of Responsibility* để giải quyết bài toán trên, chúng ta sẽ tạo giao diện *Handler*:

```
package com.javacodegeeks.patterns.chainofresponsibility;

public interface Handler {

    public void setHandler(Handler handler);
    public void process(File file);
    public String getHandlerName();
}
```

Giao diện trên chứa hai phương thức chính, *setHandler* và phương thức xử lý *process*. Phương thức *setHandler* được dùng để đặt *handler tiếp theo* trong chuỗi, trong khi phương thức *process* được dùng để xử lý yêu cầu đó, chỉ nếu *handler* này có thể xử lý được yêu cầu đó. Tùy chọn, ta có thể có phương thức *getHandlerName* mà được sử dụng để trả về tên của *handler*.

Các *handler* được thiết kế để xử lý các file mà chứa dữ liệu. *Concrete handler* kiểm tra nếu nó có thể xử lý file bằng cách kiểm tra kiểu file, ngược lại nó chuyển tiếp cho *handler tiếp theo* trên chuỗi.

Lớp File trông như sau:

```
package com.javacodegeeks.patterns.chainofresponsibility;

public class File {

    private final String fileName;
    private final String fileType;
    private final String filePath;

    public File(String fileName, String fileType, String filePath) {
        this.fileName = fileName;
        this.fileType = fileType;
        this.filePath = filePath;
    }

    public String getFileName() {
        return fileName;
    }

    public String getFileType() {
        return fileType;
    }

    public String getFilePath() {
        return filePath;
    }
}
```

Lớp *File* tạo các đối tượng *file* đơn giản mà chứa *tên file*, *kiểu file* và *đường dẫn file*. Kiểu file có thể được sử dụng bởi *handler* để kiểm tra *file* này có thể được xử lý bởi nó hay không. Nếu *handler* có thể, nó sẽ xử lý và lưu nó, hoặc nó sẽ được chuyển tiếp cho *handler tiếp theo* trên chuỗi.

```
package com.javacodegeeks.patterns.chainofresponsibility;

public class TextFileHandler implements Handler {

    private Handler handler;
    private String handlerName;

    public TextFileHandler(String handlerName) {
        this.handlerName = handlerName;
    }

    @Override
    public void setHandler(Handler handler) {
        this.handler = handler;
    }

    @Override
    public void process(File file) {
        if(file.getFileType().equals("text")){
            System.out.println("Process and saving text file... by " + handlerName);
        }else if(handler!=null){
            System.out.println(handlerName+" forwards request to "+handler. +
                getHandlerName());
            handler.process(file);
        }else{
            System.out.println("File not supported");
        }
    }

    @Override
    public String getHandlerName() {
        return handlerName;
    }
}
```

TextFileHandler được dùng để xử lý *file text*. Nó cài đặt giao diện *Handler* và ghi đè các phương thức của nó. Nó giữ tham chiếu đến *handler tiếp theo* trong chuỗi. Trong phương thức *process*, nó kiểm tra kiểu file, nếu kiểu *file* là *text*, thì nó xử lý, nếu không nó chuyển tiếp cho *handler tiếp theo*.

Các *handler* khác tương tự như *handler* trên.

```
package com.javacodegeeks.patterns.chainofresponsibility;

public class DocFileHandler implements Handler{

    private Handler handler;
    private String handlerName;

    public DocFileHandler(String handlerName) {
        this.handlerName = handlerName;
    }

    @Override
    public void setHandler(Handler handler) {
        this.handler = handler;
    }

    @Override
    public void process(File file) {

        if(file.getFileType().equals("doc")){
            System.out.println("Process and saving doc file... by "+handlerName);
        }
        else if(handler!=null){
            System.out.println(handlerName+" forwards request to "+handler);
            handler.process(file);
        }
        else{
            System.out.println("File not supported");
        }
    }

    @Override
    public String getHandlerName() {
        return handlerName;
    }
}
```

```
package com.javacodegeeks.patterns.chainofresponsibility;

public class AudioFileHandler implements Handler {

    private Handler handler;
    private String handlerName;

    public AudioFileHandler(String handlerName) {
        this.handlerName = handlerName;
    }

    @Override
    public void setHandler(Handler handler) {
        this.handler = handler;
    }

    @Override
    public void process(File file) {

        if(file.getFileType().equals("audio")){
            System.out.println("Process and saving audio file... by "+ handlerName);
        }else if(handler!=null){
            System.out.println(handlerName+" forwards request to "+handler. +
                getHandlerName());
            handler.process(file);
        }else{
            System.out.println("File not supported");
        }
    }

    @Override
    public String getHandlerName() {
        return handlerName;
    }

}

package com.javacodegeeks.patterns.chainofresponsibility;

public class ExcelFileHandler implements Handler{

    private Handler handler;
    private String handlerName;

    public ExcelFileHandler(String handlerName) {
        this.handlerName = handlerName;
    }

    @Override
    public void setHandler(Handler handler) {
        this.handler = handler;
    }

    @Override
    public void process(File file) {

        if(file.getFileType().equals("excel")){
            System.out.println("Process and saving excel file... by "+ handlerName);
        }
    }
}
```

```
        }else if(handler!=null){
            System.out.println(handlerName+" forwards request to "+handler. ←
                getHandlerName());
            handler.process(file);
        }else{
            System.out.println("File not supported");
        }
    }

@Override
public String getHandlerName() {
    return handlerName;
}
}

package com.javacodegeeks.patterns.chainofresponsibility;

public class ImageFileHandler implements Handler {

    private Handler handler;
    private String handlerName;

    public ImageFileHandler(String handlerName) {
        this.handlerName = handlerName;
    }

    @Override
    public void setHandler(Handler handler) {
        this.handler = handler;
    }

    @Override
    public void process(File file) {

        if(file.getFileType().equals("image")){
            System.out.println("Process and saving image file... by "+ ←
                handlerName);
        }else if(handler!=null){
            System.out.println(handlerName+" forwards request to "+handler. ←
                getHandlerName());
            handler.process(file);
        }else{
            System.out.println("File not supported");
        }
    }

    @Override
    public String getHandlerName() {
        return handlerName;
    }
}

package com.javacodegeeks.patterns.chainofresponsibility;

public class VideoFileHandler implements Handler {

    private Handler handler;
    private String handlerName;
```

```
public VideoFileHandler(String handlerName) {
    this.handlerName = handlerName;
}

@Override
public void setHandler(Handler handler) {
    this.handler = handler;
}

@Override
public void process(File file) {

    if(file.getFileType().equals("video")){
        System.out.println("Process and saving video file... by "+ handlerName);
    }else if(handler!=null){
        System.out.println(handlerName+" forwards request to "+handler. getHandlerName());
        handler.process(file);
    }else{
        System.out.println("File not supported");
    }
}

@Override
public String getHandlerName() {
    return handlerName;
}
}
```

Bây giờ ta kiểm tra code ở trên.

```
package com.javacodegeeks.patterns.chainofresponsibility;

public class TestChainofResponsibility {

    public static void main(String[] args) {
        File file = null;
        Handler textHandler = new TextFileHandler("Text Handler");
        Handler docHandler = new DocFileHandler("Doc Handler");
        Handler excelHandler = new ExcelFileHandler("Excel Handler");
        Handler audioHandler = new AudioFileHandler("Audio Handler");
        Handler videoHandler = new VideoFileHandler("Video Handler");
        Handler imageHandler = new ImageFileHandler("Image Handler");

        textHandler.setHandler(docHandler);
        docHandler.setHandler(excelHandler);
        excelHandler.setHandler(audioHandler);
        audioHandler.setHandler(videoHandler);
        videoHandler.setHandler(imageHandler);

        file = new File("Abc.mp3", "audio", "C:");
        textHandler.process(file);

        file = new File("Abc.jpg", "video", "C:");
        textHandler.process(file);

        file = new File("Abc.doc", "doc", "C:");
        textHandler.process(file);

        file = new File("Abc.bat", "bat", "C:");
    }
}
```

```
        textHandler.process(file);
    }
}
```

Kết quả in ra là:

```
Text Handler fowards request to Doc Handler
Doc Handler fowards request to Excel Handler
Excel Handler fowards request to Audio Handler
Process and saving audio file... by Audio Handler

Text Handler fowards request to Doc Handler
Doc Handler fowards request to Excel Handler
Excel Handler fowards request to Audio Handler
Audio Handler fowards request to Video Handler
Process and saving video file... by Video Handler

Text Handler fowards request to Doc Handler
Process and saving doc file... by Doc Handler

Text Handler fowards request to Doc Handler
Doc Handler fowards request to Excel Handler
Excel Handler fowards request to Audio Handler
Audio Handler fowards request to Video Handler
Video Handler fowards request to Image Handler
File not supported
```

Trong Ví dụ ở trên, trước hết ta tạo các *handler* khác nhau và móc nối chúng lại. Chuỗi này bắt đầu từ *text handler*, mà dùng để xử lý *file text*, đến *doc handler* và tiếp tục và cuối cùng là *image handler*.

Sau đó ta tạo ra các đối tượng file và truyền nó cho *text handler*. Nếu file đó có thể xử lý bằng *text handler* thì nó xử lý, ngược lại nó chuyển tiếp file cho *handler* tiếp theo trên chuỗi. Bạn có thể thấy ở đầu ra yêu cầu đó đã được chuyển tiếp như thế nào bởi các đối tượng trên chuỗi cho đến khi đạt được *handler* phù hợp.

Cũng lưu ý rằng, chúng ta không tạo *handler* để xử lý *file bat*. Vì vậy, nó chuyển qua tất cả mọi *handler* và kết quả đầu ra là “*File not supported*”.

Client code được tách khỏi các đối tượng phục vụ. Nó chỉ gửi yêu cầu và yêu cầu đó được phục vụ bởi bất cứ một *handler* nào trong chuỗi hoặc không được xử lý vì không hỗ trợ nó.

6.10.4 Khi nào sử dụng mẫu Chain of Responsibility

Sử dụng mẫu *Chain of Responsibility* khi

- Nhiều hơn một đối tượng có thể xử lý yêu cầu và *handler* không cần được biết trước. *Handler* cần được xác định một cách tự động.
- Bạn muốn đưa ra yêu cầu cho một số các đối tượng mà không đặc tả người nhận tường minh.
- Tập các đối tượng mà có thể xử lý yêu cầu cần được đặc tả động.

6.11 Mẫu thiết kế FLYWEIGHT

6.11.1 Mẫu thiết kế Flyweight

Lập trình hướng đối tượng làm cho lập trình dễ dàng và thú vị. Nó làm cho công việc lập trình dễ dàng hơn bởi mô hình đưa các thực thể thế giới thực vào thế giới lập trình. Lập trình viên tạo ra một lớp và khởi tạo nó bằng việc tạo ra đối tượng của nó. Đối tượng này mô hình một thực thể của thế giới thực và các đối tượng bên trong ứng dụng tương tác với nhau để hoàn thành một công việc được yêu cầu.

Nhưng đôi khi quá nhiều đối tượng có thể làm chậm mọi việc. Nhiều đối tượng có thể chiếm bộ nhớ lớn và có thể làm chậm ứng dụng hoặc ngay cả gấp vấn đề về vượt quá bộ nhớ. Như một lập trình viên tốt, bạn cần theo dõi các đối tượng khởi tạo và kiểm soát việc tạo đối tượng trong ứng dụng. Điều này đặc biệt đúng, khi ta có nhiều đối tượng tương tự và hai đối tượng cùng loại không có sự khác biệt nhiều lầm giữa chúng.

Đôi khi các đối tượng trong ứng dụng có thể có nhiều đặc tính giống nhau và là kiểu tương tự (kiểu tương tự ở đây nghĩa là hầu hết các tính chất của chúng có cùng giá trị và chỉ có một số ít khác nhau về giá trị). Có lúc, chúng cũng là các đối tượng nặng để tạo ra, chúng cần được kiểm soát bởi người phát triển ứng dụng. Ngược lại, chúng có thể chiếm nhiều bộ nhớ và tất nhiên làm chậm toàn bộ ứng dụng.

Mẫu Flyweight được thiết kế để kiểm soát việc tạo đối tượng kiểu như vậy và cung cấp cho bạn một cơ chế lưu giữ cơ bản. Nó cho phép bạn tạo một đối tượng cho mỗi kiểu (kiểu ở đây phân biệt bởi tính chất của đối tượng đó) và nếu bạn yêu cầu một đối tượng với tính chất giống đã có, thì nó sẽ trả về cùng đối tượng đó thay vì tạo đối tượng mới.

Trước khi đi sâu tìm hiểu mẫu Flyweight, chúng ta sẽ xét kịch bản sau: một trang web cho phép người sử dụng lập trình và chạy chương trình trực tuyến trên nhiều ngôn ngữ khác nhau. Chúng ta sẽ bàn về kịch bản này bây giờ và sẽ giải quyết vấn đề sử dụng mẫu Flyweight.

Trang lập trình X-programming cho phép người sử dụng tạo và chạy các chương trình sử dụng ngôn ngữ ưa thích của mình. Nó cung cấp cho bạn nhiều lựa chọn ngôn ngữ lập trình khác nhau. Bạn chọn một, viết chương trình với nó và chạy nó để xem kết quả.

Nhưng bây giờ Trang này bắt đầu mất dần người sử dụng của nó, nguyên nhân là sự chậm trễ của Trang. Người sử dụng không quan tâm đến nó nữa. Trang rất nổi tiếng và đôi khi ở đây có hàng ngàn lập trình viên sử dụng nó. Vì vậy, Trang chạy như bò. Nhưng việc sử dụng quá tải không phải là vấn đề đằng sau sự chậm trễ của Trang. Chúng ta sẽ xem code lập trình cốt lõi của Trang mà cho phép người sử dụng chạy và thực thi các chương trình của họ và vấn đề ở đâu sẽ được nhận biết.

```
package com.javacodegeeks.patterns.flyweightpattern;

public class Code {

    private String code;

    public String getCode() {
        return code;
    }

    public void setCode(String code) {

        this.code = code;
    }
}
```

Lớp trên được sử dụng để đặt *code* được tạo bởi lập trình viên để làm cho nó chạy. Đối tượng *code* là đối tượng nhẹ có tính chất *code* với các phương thức *set* và *get*.

```
package com.javacodegeeks.patterns.flyweightpattern;

public interface Platform {

    public void execute(Code code);
}
```

Giao diện *Platform* được cài đặt bởi nền tảng chuyên biệt của một ngôn ngữ để thực thi *code*. Nó có một phương thức *execute*, mà dùng đối tượng *code* như tham số.

```
package com.javacodegeeks.patterns.flyweightpattern;

public class JavaPlatform implements Platform {

    public JavaPlatform() {
        System.out.println("JavaPlatform object created");
    }

    @Override
    public void execute(Code code) {
        System.out.println("Compiling and executing Java code.");
    }
}
```

Lớp trên cài đặt giao diện *Platform* và cung cấp cài đặt cho phương thức *execute* để thực thi *code* trong Java.

Để thực thi *code*, đối tượng *Code* mà chứa *code* và đối tượng *Platform* sẽ thực thi *code* được tạo. Code để chạy trông như sau:

```
Platform platform = new JavaPlatform();
platform.execute(code);
```

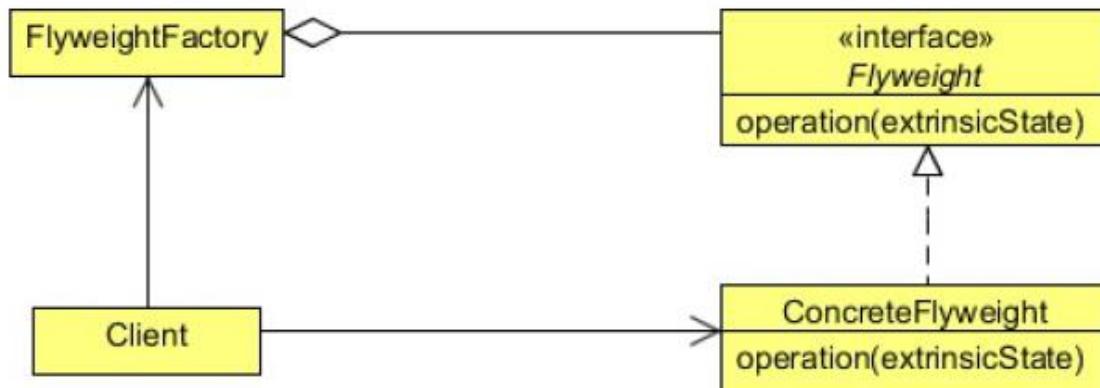
Bây giờ giả sử, có khoảng 2k người sử dụng trực tuyến và chạy *code* của họ mà theo đó có 2k đối tượng *code* và 2k đối tượng *platform*. Các đối tượng *code* là nhẹ và cần phải có mỗi đối tượng *code* cho mỗi người sử dụng. Nhưng, *Platform* là đối tượng nặng mà được sử dụng để tạo môi trường thực thi. Tạo quá nhiều đối tượng *Platform* là tốn thời gian và là nhiệm vụ khó khăn. Chúng ta cần kiểm soát việc tạo đối tượng *Platform* mà có thể được thực hiện sử dụng mẫu *Flyweight*, nhưng trước hết chúng ta tìm hiểu mẫu *Flyweight*.

6.11.2 Mẫu Flyweight là gì

Mục đích của mẫu *Flyweight* là để sử dụng các đối tượng chia sẻ mà hỗ trợ số lượng lớn các đối tượng phân loại nhỏ một cách hiệu quả. *Flyweight* là đối tượng chia sẻ mà có thể được sử dụng trong nhiều ngữ cảnh một cách đồng thời. *Flyweight* hoạt động như đối tượng độc lập trong mỗi ngữ cảnh – không phân biệt được việc tạo ra đối tượng mà nó không chia sẻ. *Flyweight* không tạo ra các điều kiện về ngữ cảnh mà ở đó chúng vận hành. Khái niệm chính ở đây là sự phân biệt giữa trạng thái bên trong và bên ngoài. Trạng thái bên trong là được lưu giữ trong *Flyweight*, nó chứa các thông tin mà độc lập với ngữ cảnh của *flyweight*, do đó tạo nên nó chia sẻ được. Trạng thái bên ngoài (*extrinsic state*) phụ thuộc vào và đa dạng với ngữ cảnh *flyweight* và do đó không chia sẻ được. *Client object* có trách nhiệm truyền trạng thái bên ngoài cho *flyweight* khi nó cần đến nó.

Xét kịch bản ứng dụng mà bao gồm tạo ra số lượng lớn các đối tượng mà là duy nhất chỉ theo giá trị một số ít tham số. Nói cách khác, các đối tượng này chứa dữ liệu bên trong và bất biến mà là chung giữa mọi đối tượng. Dữ liệu bên trong cần được tạo và duy trì như một phần của mỗi đối tượng mà được tạo ra. Việc tạo tổng thể và duy trì nhóm lớn các đối tượng như vậy có thể là rất đắt theo chi phí bộ nhớ và hiệu năng. Mẫu *Flyweight* có thể được sử dụng trong mỗi kịch bản để thiết kế cách tạo đối tượng hiệu quả hơn.

Đây là sơ đồ lớp của mẫu *Flyweight*:



Flyweight

- Khai báo giao diện qua đó các *flyweight* có thể nhận được và hoạt động trên trạng thái ngoài.

Concrete Flyweight

- Cài đặt giao diện *Flyweight* và bổ sung lưu giữ cho trạng thái bên trong. Một đối tượng *ConcreteFlyweight* cần phải là chia sẻ được. Bất cứ trạng thái mà nó lưu giữ cần là bên trong, mà là độc lập với ngữ cảnh của đối tượng *ConcreteFlyweight*.

FlyweightFactory

- Tạo và quản lý các đối tượng *flyweight*
- Tin tưởng rằng *flyweight* là chia sẻ đúng đắn. Khi một *client* yêu cầu một *flyweight*, đối tượng *FlyweightFactory* cung cấp khởi tạo đã có hoặc tạo ra cái mới, nếu nó chưa có.

Client

- Duy trì tham chiếu đến *flyweight*
- Tính hoặc lưu trạng thái ngoài của *flyweight*

6.11.3 Giải pháp cho bài toán

Để giải bài toán trên, chúng ta sẽ tạo lớp Platform Factory mà sẽ kiểm soát việc tạo các đối tượng *Platform*.

```
package com.javacodegeeks.patterns.flyweightpattern;

import java.util.HashMap;
import java.util.Map;

public final class PlatformFactory {

    private static Map<String, Platform> map = new HashMap<>();
    private PlatformFactory(){
        throw new AssertionError("Cannot instantiate the class");
    }

    public static synchronized Platform getPlatformInstance(String platformType){
        Platform platform = map.get(platformType);
        if(platform==null){
            switch(platformType){
                case "C" : platform = new CPlatform();
                             break;
                case "CPP" : platform = new CPPPlatform();
                              break;
                case "JAVA" : platform = new JavaPlatform();
                               break;
                case "RUBY" : platform = new RubyPlatform();
                               break;
            }
            map.put(platformType, platform);
        }
        return platform;
    }

}
```

Lớp trên chứa *map* tĩnh mà giữ đối tượng *String* như khóa và đối tượng *Platform* như giá trị của nó. Chúng ta không muốn tạo khởi tạo của lớp này nên chỉ giữ hàm tạo của nó là *private* và đưa ra báo lỗi *AssertionError* chỉ để tránh bắt cứ việc tạo lỗi nào của đối tượng đó ngay cả bên trong lớp.

Phương thức chính và chỉ một phương thức của lớp này là *getPlatformInstance*. Nó là phương thức tĩnh mà có *PlatformType* là tham số. *PlatformType* này được sử dụng như khóa của *map*, trước hết nó kiểm tra *map* xem có đối tượng platform nào mà có khóa key đã tồn tại hay chưa. Nếu không có đối tượng như vậy được tìm thấy, *platform* thích hợp sẽ được tạo, nó sẽ đặt vào trong *map* và sau đó phương thức này sẽ trả về đối tượng đó. Lần sau, khi cùng một đối tượng kiểu *Platform* được yêu cầu, đối tượng tồn tại sẽ được trả về, thay vì tạo mới.

Cũng lưu ý rằng phương thức *getPlatformInstance* là *synchronized* để cung cấp an toàn luồng khi kiểm tra và tạo khởi tạo của đối tượng. Trong ví dụ trên, không có tính chất bên trong nào của đối tượng này mà được chia sẻ, nhưng chỉ có tính chất bên ngoài mà là đối tượng code được cung cấp bởi *client code*.

Bây giờ ta sẽ kiểm tra *code* trên.

```
package com.javacodegeeks.patterns.flyweightpattern;

public class TestFlyweight {

    public static void main(String[] args) {

        Code code = new Code();
        code.setCode("C Code...");
        Platform platform = PlatformFactory.getPlatformInstance("C");
        platform.execute(code);
        System.out.println("*****");
        code = new Code();
        code.setCode("C Code2...");
        platform = PlatformFactory.getPlatformInstance("C");
        platform.execute(code);
        System.out.println("*****");
        code = new Code();
        code.setCode("JAVA Code...");
        platform = PlatformFactory.getPlatformInstance("JAVA");
        platform.execute(code);
        System.out.println("*****");

        code = new Code();
        code.setCode("JAVA Code2...");
        platform = PlatformFactory.getPlatformInstance("JAVA");
        platform.execute(code);
        System.out.println("*****");
        code = new Code();
        code.setCode("RUBY Code...");
        platform = PlatformFactory.getPlatformInstance("RUBY");
        platform.execute(code);
        System.out.println("*****");
        code = new Code();
        code.setCode("RUBY Code2...");
        platform = PlatformFactory.getPlatformInstance("RUBY");
        platform.execute(code);
    }

}
```

Đoạn code trên sẽ cho kết quả đầu ra như sau:

```
CPlatform object created
Compiling and executing C code.
*****
Compiling and executing C code.
*****
JavaPlatform object created
Compiling and executing Java code.
*****
Compiling and executing Java code.
*****
RubyPlatform object created
Compiling and executing Ruby code.
*****
Compiling and executing Ruby code.
```

Trong lớp trên, trước hết ta tạo đối tượng code và đặt nó là *C code* trong đó. Sau đó ta yêu cầu *PlatformFactory* cung cấp platform để thực thi code này. Sau đó, ta gọi phương thức *execute* trên đối tượng trả về, truyền đối tượng code cho nó.

Chúng ta thực hiện cùng thủ tục như vậy, tức là tạo và đặt đối tượng Code đó và sau đó yêu cầu đối tượng *Platform*, chuyên biệt cho code này. Đầu ra chỉ rõ rằng đối tượng *platform* chỉ được tạo lần đầu khi được yêu cầu, lần sau nó trả về cùng đối tượng đã có.

Các lớp chuyên biệt nền tảng khác cũng tương tự như *JavaPlatform* cũng được nêu ra.

```
package com.javacodegeeks.patterns.flyweightpattern;

public class CPlatform implements Platform {

    public CPlatform() {
        System.out.println("CPlatform object created");
    }

    @Override
    public void execute(Code code) {
        System.out.println("Compiling and executing C code.");
    }

}

package com.javacodegeeks.patterns.flyweightpattern;

public class CPPPlatform implements Platform{

    public CPPPlatform() {
        System.out.println("CPPPlatform object created");
    }

    @Override
    public void execute(Code code) {
        System.out.println("Compiling and executing CPP code.");
    }

}

package com.javacodegeeks.patterns.flyweightpattern;

public class RubyPlatform implements Platform{

    public RubyPlatform() {
        System.out.println("RubyPlatform object created");
    }

    @Override
    public void execute(Code code) {
        System.out.println("Compiling and executing Ruby code.");
    }

}
```

Nếu chúng ta xét có 2k người sử dụng đồng thời cùng sử dụng Trang này, chính xác 2k đối tượng *code* nhẹ sẽ được tạo nhưng chỉ 4 đối tượng *platform* nặng được khởi tạo. Lưu ý rằng, chúng ta nói bốn đối tượng *platform* bởi có tối thiểu một người nào đó sử dụng một ngôn ngữ. Nếu không có ai sử dụng *Ruby*, thì chỉ có 3 đối tượng nền tảng được tạo.

6.11.4 Khi nào sử dụng mẫu Flyweight

Tính hiệu quả của mẫu *Flyweight* phụ thuộc nhiều vào việc làm như thế nào và ở đâu chúng được sử dụng. Áp dụng mẫu *Flyweight* khi mọi điều sau đề đúng:

- Ứng dụng sử dụng số lượng lớn các đối tượng.
- Giá lưu trữ là cao vì số lượng lớn các đối tượng.
- Hầu hết trạng thái đối tượng có thể được tạo bên ngoài.
- Nhiều nhóm các đối tượng có thể được thay thế bằng một số ít các đối tượng chia sẻ khi trạng thái bên ngoài được hủy bỏ.
- Ứng dụng này không phụ thuộc vào định danh đối tượng. Vì vậy các đối tượng *flyweight* có thể được chia sẻ, kiểm tra định danh sẽ trả về *true* cho các đối tượng khác nhau về bản chất.

6.12 Mẫu thiết kế **BUILDER**

6.12.1 Mẫu thiết kế **Builder**

Nói chung, chi tiết việc cấu tạo đối tượng, như khởi tạo và khởi động các thành phần mà tạo nên đối tượng, được giữ bên trong đối tượng, như một phần của hàm tạo. Kiểu thiết kế này gắn chặt quá trình tạo đối tượng với các thành phần mà làm nên đối tượng. Cách tiếp cận này là phù hợp khi việc tạo đối tượng là đơn giản và quá trình tạo đối tượng là xác định và luôn sinh ra cùng một đại diện của đối tượng.

Tuy nhiên, thiết kế này có thể không hiệu quả khi đối tượng được tạo là phức tạp và một dãy các bước làm nên quá trình tạo đối tượng có thể được cài đặt theo các cách khác nhau, do đó tạo ra các đại diện khác nhau của đối tượng. Vì các cài đặt khác nhau của quá trình tạo này được giữ bên trong đối tượng, đối tượng có thể trở nên đồ sộ và kém có tính cấu phần. Hệ quả là bổ sung một cài đặt mới hay tạo một số thay đổi cho cài đặt hiện tại yêu cầu thay đổi đến code hiện tại.

Sử dụng mẫu *Builder*, quá trình tạo một đối tượng như vậy có thể được thiết kế hiệu quả hơn. Mẫu *Builder* đề xuất chuyển *logic* tạo ra khỏi lớp đối tượng thành một lớp riêng được tham chiếu như *class Builder*. Ở đây có thể có nhiều hơn một lớp *Builder*, mỗi lớp với cài đặt khác nhau cho dãy các bước tạo nên đối tượng. Mỗi cài đặt *Builder* cho kết quả là đại diện khác nhau của đối tượng.

Để sử dụng mẫu *Builder*, thử trợ giúp công ty ôtô mà triển lãm các ôtô khác nhau của nó sử dụng mô hình đồ họa cho khách hàng. Công ty có công cụ đồ họa mà trưng bày ôtô trên màn hình. Yêu cầu của công cụ này là cung cấp đối tượng *car* cho nó. Đối tượng *car* cần chứa đặc tả của *car*. Công cụ đồ họa sẽ sử dụng đặc tả để hiển thị ôtô. Công ty cần phân loại các ôtô của nó thành các loại khác nhau như ôtô *Sedan* hay ôtô *Sports*. Chỉ có một đối tượng *car* và công việc của chúng ta là tạo đối tượng *car* này tuân theo phân lớp đó. Chẳng hạn, đối với ôtô *Sedan*, một đối tượng *car* tuân theo đặc tả phân loại *Sedan* cần được xây dựng hoặc nếu ôtô *Sport* được yêu cầu, thì đối tượng *car* tuân theo đặc tả ôtô *Sport* cần được xây dựng. Hiện tại, Công ty muốn chỉ hai kiểu ôtô này, nhưng nó có thể yêu cầu các kiểu ôtô khác trong tương lai.

Chúng ta sẽ tạo hai *Builder* khác nhau, mỗi một của một phân loại, tức là *Sedan* và *Sport*. Hai *builder* này sẽ giúp xây dựng đối tượng ôtô tuân theo đặc tả của nó. Nhưng trước hết, chúng ta sẽ bàn chi tiết về mẫu *Builder*.

6.12.2 Mẫu *Builder* là gì

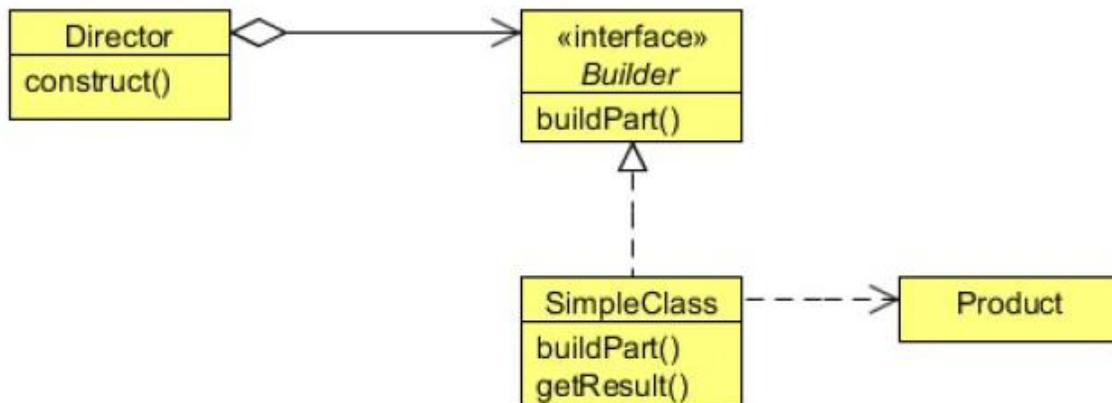
Mục đích của mẫu *Builder* là tách việc tạo một đối tượng phức tạp ra khỏi biểu diễn của nó, sao cho cùng một quá trình tạo có thể tạo được các biểu diễn khác nhau. Kiểu tách này sẽ giảm kích thước của đối tượng. Thiết kế này trở nên có cấu trúc hơn với mỗi cài đặt được chứa trong một đối tượng *builder* khác nhau. Bổ sung cài đặt mới (tức là bổ sung *builder* mới) trở nên dễ dàng. Quá trình tạo đối tượng trở nên độc lập khỏi các thành phần mà tạo nên đối tượng. Nó cung cấp kiểm soát tốt hơn quá trình tạo đối tượng.

Theo thuật ngữ cài đặt, mỗi bước khác nhau trong quá trình tạo có thể được khai báo như các phương thức của một giao diện chung để được cài đặt bằng các *builder* cụ thể khác nhau.

Một đối tượng *client* có thể tạo một khởi tạo của một *builder* cụ thể và triệu gọi tập các phương thức được yêu cầu để xây dựng các phần khác nhau của đối tượng cuối cùng. Cách tiếp cận này yêu cầu mỗi đối tượng *client* sẽ quan tâm đến logic tạo đó. Mỗi khi logic tạo trải qua thay đổi, mọi đối tượng *client* cần được thay đổi phù hợp.

Mẫu *Builder* đưa ra mức độ tách bạch khác để giải quyết vấn đề này. Thay vì các đối tượng *client* triệu gọi các phương thức tạo khác nhau trực tiếp, mẫu *Builder* đề xuất sử dụng một đối tượng được chỉ định được tham chiếu như *Director* mà có trách nhiệm triệu gọi các phương thức *builder* khác nhau được yêu cầu để xây dựng đối tượng cuối cùng. Các đối tượng *client* khác nhau có thể sử dụng đối tượng *Director* để tạo đối tượng được yêu cầu. Một khi một đối tượng được tạo, đối tượng *client* có thể yêu cầu trực tiếp từ *builder* đối tượng được xây dựng đầy đủ. Để làm thuận tiện quá trình này, một phương thức mới *getObject* có thể được khai báo trong giao diện *Builder* chung để được cài đặt bởi các *builder* cụ thể khác nhau.

Thiết kế mới này loại bỏ sự không cần thiết cho đối tượng *client* phải làm việc với các phương thức tạo dựng trong quá trình tạo đối tượng và đóng gói chi tiết cách mà đối tượng được tạo khỏi *client*.



Builder

- Đặc tả giao diện tổng quát cho các phần khác nhau của đối tượng *Product*

ConcreteBuilder

- Tạo và lắp ghép các phần của *Product* bằng cách cài đặt giao diện *Builder*
- Định nghĩa và theo dõi các thể hiện mà nó tạo ra
- Cung cấp giao diện để truy cập *Product*

Director

- Tạo một đối tượng sử dụng giao diện *Builder*

Product

- Biểu diễn đối tượng phức tạp trong quá trình xây dựng. *ConcreteBuilder* xây dựng biểu diễn bên trong của *Product* và định nghĩa quá trình mà nó được lắp ghép.
- Bao gồm các lớp mà định nghĩa các thành phần tạo nên, chứa cả giao diện để lắp ghép các phần thành kết quả cuối cùng.

6.12.3 Cài đặt mẫu Builder

Dưới đây là lớp *Car (Product)* mà chứa một số thành phần quan trọng của ôtô mà được yêu cầu để xây dựng đối tượng *car* hoàn chỉnh.

```
package com.javacodegeeks.patterns.builderpattern;

public class Car {

    private String bodyStyle;
    private String power;
    private String engine;
    private String breaks;
    private String seats;
    private String windows;
    private String fuelType;
    private String carType;

    public Car (String carType) {
        this.carType = carType;
    }

    public String getBodyStyle() {
        return bodyStyle;
    }
    public void setBodyStyle(String bodyStyle) {
        this.bodyStyle = bodyStyle;
    }
    public String getPower() {
        return power;
    }
    public void setPower(String power) {
        this.power = power;
    }
    public String getEngine() {
        return engine;
    }
    public void setEngine(String engine) {
        this.engine = engine;
    }
    public String getBreaks() {
        return breaks;
    }
    public void setBreaks(String breaks) {
        this.breaks = breaks;
    }
    public String getSeats() {
        return seats;
    }
    public void setSeats(String seats) {
        this.seats = seats;
    }
    public String getWindows() {
        return windows;
    }
    public void setWindows(String windows) {
        this.windows = windows;
    }
    public String getFuelType() {
        return fuelType;
    }
}
```

```
public void setFuelType(String fuelType) {
    this.fuelType = fuelType;
}

@Override
public String toString(){
    StringBuilder sb = new StringBuilder();
    sb.append("-----"+carType+"----- \\\n");
    sb.append(" Body: ");
    sb.append(bodyStyle);
    sb.append("\\n Power: ");
    sb.append(power);
    sb.append("\\n Engine: ");
    sb.append(engine);
    sb.append("\\n Breaks: ");
    sb.append(breaks);
    sb.append("\\n Seats: ");
    sb.append(seats);
    sb.append("\\n Windows: ");
    sb.append(windows);
    sb.append("\\n Fuel Type: ");
    sb.append(fuelType);

    return sb.toString();
}
}
```

CarBuilder là giao diện *Builder* chứa tập các phương thức được sử dụng để xây dựng đối tượng *car* và các thành phần của nó.

```
package com.javacodegeeks.patterns.builderpattern;

public interface CarBuilder {

    public void buildBodyStyle();
    public void buildPower();
    public void buildEngine();
    public void buildBreaks();
    public void buildSeats();
    public void buildWindows();
    public void buildFuelType();
    public Car getCar();
}
```

Phương thức *getCar* được sử dụng để trả về đối tượng *car* cuối cùng sau khi tạo.

Ta xem hai cài đặt của giao diện *CarBuilder*, một cho mỗi loại *car*, tức là *Sedan* và *Sport*.

```
package com.javacodegeeks.patterns.builderpattern;

public class SedanCarBuilder implements CarBuilder{

    private final Car car = new Car("SEDAN");

    @Override
    public void buildBodyStyle() {
        car.setBodyStyle("External dimensions: overall length (inches): 202.9, " +
                        "overall width (inches): 76.2, overall height (inches): 60.7, wheelbase (inches): 112.9, " +
                        "front track (inches): 65.3, rear track (inches): 65.5 and curb to curb turning circle (feet): 39.5");
    }

    @Override
    public void buildPower(){
        car.setPower("285 hp @ 6,500 rpm; 253 ft lb of torque @ 4,000 rpm");
    }

    @Override
    public void buildEngine() {
        car.setEngine("3.5L Duramax V 6 DOHC");
    }

    @Override
    public void buildBreaks() {
        car.setBreaks("Four-wheel disc brakes: two ventilated. Electronic brake distribution");
    }

    @Override
    public void buildSeats() {
        car.setSeats("Front seat center armrest.Rear seat center armrest.Split-folding rear seats");
    }

    @Override
    public void buildWindows() {
        car.setWindows("Laminated side windows.Fixed rear window with defroster");
    }

    @Override
    public void buildFuelType() {
        car.setFuelType("Gasoline 19 MPG city, 29 MPG highway, 23 MPG combined and 437 mi. range");
    }

    @Override
    public Car getCar(){
        return car;
    }
}
```

```
package com.javacodegeeks.patterns.builderpattern;

public class SportsCarBuilder implements CarBuilder{

    private final Car car = new Car("SPORTS");

    @Override
    public void buildBodyStyle() {
        car.setBodyStyle("External dimensions: overall length (inches): 192.3, " +
                        " overall width (inches): 75.5, overall height (inches): 54.2, wheelbase (inches): 112.3, " +
                        " front track (inches): 63.7, rear track (inches): 64.1 and curb to curb turning circle (feet): 37.7");
    }

    @Override
    public void buildPower(){
        car.setPower("323 hp @ 6,800 rpm; 278 ft lb of torque @ 4,800 rpm");
    }

    @Override
    public void buildEngine() {
        car.setEngine("3.6L V 6 DOHC and variable valve timing");
    }

    @Override
    public void buildBreaks() {
        car.setBreaks("Four-wheel disc brakes: two ventilated. Electronic brake distribution. StabiliTrak stability control");
    }

    @Override
    public void buildSeats() {
        car.setSeats("Driver sports front seat with one power adjustments manual height, front passenger seat sports front seat with one power adjustments");
    }

    @Override
    public void buildWindows() {
        car.setWindows("Front windows with one-touch on two windows");
    }

    @Override
    public void buildFuelType() {
        car.setFuelType("Gasoline 17 MPG city, 28 MPG highway, 20 MPG combined and 380 mi. range");
    }

    @Override
    public Car getCar(){
        return car;
    }
}
```

Đây là lớp Director, lưu giữ các bước tương ứng tạo nên các thành phần để tạo ra sản phẩm car.

```
public class CarDirector {  
    private CarBuilder carBuilder;  
  
    public CarDirector(CarBuilder carBuilder){  
        this.carBuilder = carBuilder;  
    }  
  
    public void build(){  
        carBuilder.buildBodyStyle();  
        carBuilder.buildPower();  
        carBuilder.buildEngine();  
        carBuilder.buildBreaks();  
        carBuilder.buildSeats();  
        carBuilder.buildWindows();  
        carBuilder.buildFuelType();  
    }  
}
```

Hai lớp *Builder* trên tạo và xây dựng *Product*, tức là đối tượng *car* tuân theo đặc tả được yêu cầu. Bây giờ ta kiểm tra *Builder*.

```
package com.javacodegeeks.patterns.builderpattern;  
  
public class TestBuilderPattern {  
  
    public static void main(String[] args) {  
        CarBuilder carBuilder = new SedanCarBuilder();  
        CarDirector director = new CarDirector(carBuilder);  
        director.build();  
        Car car = carBuilder.getCar();  
        System.out.println(car);  
  
        carBuilder = new SportsCarBuilder();  
        director = new CarDirector(carBuilder);  
        director.build();  
        car = carBuilder.getCar();  
        System.out.println(car);  
    }  
}
```

Code trên cho kết quả sau:

```
-----SEDAN-----  
Body: External dimensions: overall length (inches): 202.9, overall width (inches): 76.2, ↵  
overall height (inches): 60.7, wheelbase (inches): 112.9, front track (inches): 65.3, ↵  
rear track (inches): 65.5 and curb to curb turning circle (feet): 39.5  
Power: 285 hp @ 6,500 rpm; 253 ft lb of torque @ 4,000 rpm  
Engine: 3.5L Duramax V 6 DOHC  
Breaks: Four-wheel disc brakes: two ventilated. Electronic brake distribution  
Seats: Front seat center armrest.Rear seat center armrest.Split-folding rear seats  
Windows: Laminated side windows.Fixed rear window with defroster  
Fuel Type: Gasoline 19 MPG city, 29 MPG highway, 23 MPG combined and 437 mi. range  
-----SPORTS-----  
Body: External dimensions: overall length (inches): 192.3, overall width (inches): 75.5, ↵  
overall height (inches): 54.2, wheelbase (inches): 112.3, front track (inches): 63.7, ↵  
rear track (inches): 64.1 and curb to curb turning circle (feet): 37.7  
Power: 323 hp @ 6,800 rpm; 278 ft lb of torque @ 4,800 rpm  
Engine: 3.6L V 6 DOHC and variable valve timing  
Breaks: Four-wheel disc brakes: two ventilated. Electronic brake distribution. StabiliTrak ↵  
stability control  
Seats: Driver sports front seat with one power adjustments manual height, front passenger ↵  
seat sports front seat with one power adjustments  
Windows: Front windows with one-touch on two windows  
Fuel Type: Gasoline 17 MPG city, 28 MPG highway, 20 MPG combined and 380 mi. range
```

Trong lớp trên, trước hết chúng ta tạ *SedanBuilder* và *CarDirector*. Sau đó chúng ta yêu cầu *CarDirector* xây dựng car tuân theo *builder* mà truyền cho nó. Cuối cùng, chúng ta trực tiếp yêu cầu *builder* cung cấp đối tượng car được tạo ra.

Chúng ta làm như vậy với *SportBuilder* mà trả về đối tượng car tuân theo đặc tả *Sport car*.

Cách tiếp cận sử dụng mẫu *Builder* là linh hoạt đủ để bổ sung bất cứ loại ôtô mới nào trong tương lai mà không thay đổi bất cứ code nào đã tồn tại. Tất cả những gì chúng ta cần là tạo ra *builder* mới tuân theo đặc tả của ôtô mới và cung cấp nó cho *Director* để xây dựng.

6.12.4 Dạng khác của mẫu Builder

Có dạng khác của mẫu *Builder* khác với những gì mà chúng ta đã nhìn thấy. Đôi khi có một đối tượng với danh sách dài các tính chất, và hầu hết các tính chất này đều là tùy chọn. Xét form trực tuyến mà cần phải điền vào để trở thành thành viên của Trang. Bạn cần điền vào mọi ô bắt buộc, nhưng bạn có thể bỏ qua các ô tùy chọn hoặc đôi khi có thể đáng để nhập một vài ô tùy chọn.

Hãy kiểm tra lớp *Form* dưới đây, mà chứa danh sách dài các tính chất và một số tính chất là tùy chọn. Các ô bắt buộc trong *Form* có *firstName*, *lastName*, *userName* và *password*, nhưng các ô khác là tùy chọn.

```
package com.javacodegeeks.patterns.builderpattern;

import java.util.Date;

public class Form {

    private String firstName;
    private String lastName;
    private String userName;
    private String password;
    private String address;
    private Date dob;
    private String email;
    private String backupEmail;
    private String spouseName;
    private String city;

    private String state;
    private String country;
    private String language;
    private String passwordHint;
    private String securityQuestion;
    private String securityAnswer;

}
```

Câu hỏi là, dạng hàm tạo như thế nào chúng ta cần viết cho lớp này. Viết hàm tạo với danh sách tham số dài không là lựa chọn tốt, nó làm cho client thất vọng đặc biệt nếu các trường quan trọng thì có ít. Điều này làm tăng phạm vi lỗi, *client* có thể cung cấp lỗi một giá trị cho một trường sai chẽ.

Cách khác là sử dụng lồng ghép hàm tạo, ở đó bạn cần cung cấp cho một hàm tạo với các tham số bắt buộc, cái khác với một tham số tùy chọn, cái thứ ba với hai tham số tùy chọn, và cứ như vậy và đến cuối cùng là hàm tạo với mọi tham số tùy chọn.

Lồng ghép hàm tạo có thể trông như sau:

```
public Form(String firstName, String lastName) {
    this(firstName, lastName, null, null);
}

public Form(String firstName, String lastName, String userName, String password) {
    this(firstName, lastName, userName, password, null, null);
}

public Form(String firstName, String lastName, String userName, String password, String address ←
, Date dob) {
    this(firstName, lastName, userName, password, address, dob, null, null);
}

public Form(String firstName, String lastName, String userName, String password, String address ←
, Date dob, String email, String backupEmail) {
    ...
}
```

Khi bạn muốn tạo một khởi tạo, bạn sử dụng hàm tạo với danh sách ngắn tham số chứa mọi tham số bạn muốn nhập.

Hàm tạo lồng nhau hoạt động, nhưng khó viết *code client* khi có nhiều tham số, và ngay cả rất khó đọc nó. Người đọc cần xem xét, các giá trị có ý nghĩa gì và để tìm ra cái phù hợp. Các dãy dài các tham số kiểu xác định có thể gây ra các lỗi, Nếu *client* vô tình đảo ngược hai tham số, chương trình dịch sẽ không có vấn đề gì, nhưng chương trình sẽ có hành vi không đúng tại thời gian chạy.

Cách thứ hai, khi bạn đổi mặt với nhiều tham số hàm tạo, là mẫu JavaBeans, ở đó bạn gọi hàm tạo ít tham số để tạo đối tượng và sau đó gọi các phương thức để nhập mỗi tham số bắt buộc và mỗi tham số quan tâm tùy chọn.

Không may mắn, mẫu JavaBeans có nhược điểm nghiêm trọng của chính nó. Vì hàm tạo được tách ra nhiều lời gọi, JavaBeans có thể ở trong trạng thái nửa vời không đúng đắn qua hàm tạo của nó. Lớp này không có tùy chọn áp đặt tính đúng đắn bằng việc kiểm tra tính đúng đắn của các tham số hàm tạo. Thủ sử dụng đối tượng khi nó ở trong trạng thái không đúng đắn có thể gây ra lỗi mà đã được loại bỏ khỏi code chừa lỗi, vì vậy rất khó bắt lỗi.

Có cách thứ ba mà kết hợp an toàn của hàm tạo lồng nhau với tính đọc được của mẫu JavaBeans. Nó là một dạng *form* của mẫu *Builder*. Thay vì tạo trực tiếp đối tượng mong muốn, *client* gọi hàm tạo với mọi tham số bắt buộc và nhận được đối tượng *builder*. Sau đó *client* gọi các hàm giống *set* trên đối tượng *builder* để nhập mỗi giá trị tham số quan tâm. Cuối cùng, *client* gọi phương thức ít tham số để tạo đối tượng.

```
package com.javacodegeeks.patterns.builderpattern;  
import java.util.Date;
```

```
public class Form {

    private String firstName;
    private String lastName;
    private String userName;
    private String password;
    private String address;
    private Date dob;
    private String email;
    private String backupEmail;
    private String spouseName;
    private String city;
    private String state;
    private String country;
    private String language;
    private String passwordHint;
    private String securityQuestion;
    private String securityAnswer;

    public static class FormBuilder {

        private String firstName;
        private String lastName;
        private String userName;
        private String password;
        private String address;
        private Date dob;
        private String email;
        private String backupEmail;
        private String spouseName;
        private String city;
        private String state;
        private String country;
        private String language;
        private String passwordHint;
        private String securityQuestion;
        private String securityAnswer;

        public FormBuilder(String firstName, String lastName, String userName, ←
            String password) {
            this.firstName = firstName;
            this.lastName = lastName;
            this.userName = userName;
            this.password = password;
        }

        public FormBuilder address(String address) {
            this.address = address;
            return this;
        }

        public FormBuilder dob(Date dob) {
            this.dob = dob;
            return this;
        }

        public FormBuilder email(String email) {
            this.email = email;
            return this;
        }

        public FormBuilder backupEmail(String backupEmail) {
```

```
        this.backupEmail = backupEmail;
        return this;
    }

    public FormBuilder spouseName(String spouseName) {
        this.spouseName = spouseName;
        return this;
    }

    public FormBuilder city(String city) {
        this.city = city;
        return this;
    }

    public FormBuilder state(String state) {
        this.state = state;
        return this;
    }

    public FormBuilder country(String country){
        this.country = country;
        return this;
    }

    public FormBuilder language(String language){
        this.language = language;
        return this;
    }

    public FormBuilder passwordHint(String passwordHint){
        this.passwordHint = passwordHint;
        return this;
    }

    public FormBuilder securityQuestion(String securityQuestion){
        this.securityQuestion = securityQuestion;
        return this;
    }

    public FormBuilder securityAnswer(String securityAnswer){
        this.securityAnswer = securityAnswer;
        return this;
    }

    public Form build(){
        return new Form(this);
    }
}

private Form(FormBuilder formBuilder){

    this.firstName = formBuilder.firstName;
    this.lastName = formBuilder.lastName;
    this.userName = formBuilder.userName;
    this.password = formBuilder.password;
    this.address = formBuilder.address;
    this.dob = formBuilder.dob;
    this.email = formBuilder.email;
    this.backupEmail = formBuilder.backupEmail;
    this.spouseName = formBuilder.spouseName;
    this.city = formBuilder.city;
    this.state = formBuilder.state;
```

```
        this.country = formBuilder.country;
        this.language = formBuilder.language;
        this.passwordHint = formBuilder.passwordHint;
        this.securityQuestion = formBuilder.securityQuestion;
        this.securityAnswer = formBuilder.securityAnswer;
    }

@Override
public String toString(){
    StringBuilder sb = new StringBuilder();
    sb.append(" First Name: ");
    sb.append(firstName);
    sb.append("\n Last Name: ");
    sb.append(lastName);
    sb.append("\n User Name: ");
    sb.append(userName);
    sb.append("\n Password: ");
    sb.append(password);

    if(this.address!=null){
        sb.append("\n Address: ");
        sb.append(address);
    }
    if(this.dob!=null){
        sb.append("\n DOB: ");
        sb.append(dob);
    }
    if(this.email!=null){
        sb.append("\n Email: ");
        sb.append(email);
    }
    if(this.backupEmail!=null){
        sb.append("\n Backup Email: ");
        sb.append(backupEmail);
    }
    if(this.spouseName!=null){
        sb.append("\n Spouse Name: ");
        sb.append(spouseName);
    }
    if(this.city!=null){
        sb.append("\n City: ");
        sb.append(city);
    }
    if(this.state!=null){
        sb.append("\n State: ");
        sb.append(state);
    }
    if(this.country!=null){
        sb.append("\n Country: ");
        sb.append(country);
    }
    if(this.language!=null){
        sb.append("\n Language: ");
        sb.append(language);
    }
    if(this.passwordHint!=null){
        sb.append("\n Password Hint: ");
        sb.append(passwordHint);
    }
    if(this.securityQuestion!=null){
        sb.append("\n Security Question: ");
        sb.append(securityQuestion);
    }
}
```

```
        }
        if(this.securityAnswer!=null) {
            sb.append("\n Security Answer: ");
            sb.append(securityAnswer);
        }

        return sb.toString();
    }

    public static void main(String[] args) {
        Form form = new Form.Builder("Dave", "Carter", "DavCarter", "DAvCaEr123 ←
            ").passwordHint("MyName").city("NY").language("English").build();
        System.out.println(form);
    }

}
```

Code trên tạo kết quả đầu ra như sau:

```
First Name: Dave
Last Name: Carter
User Name: DavCarter
Password: DAvCaEr123
City: NY
Language: English
Password Hint: MyName
```

Như bạn đã thấy rõ, bây giờ *client* chỉ cần cung cấp các trường bắt buộc và các trường mà quan trọng với anh ta. Để tạo đối tượng *form*, bây giờ chúng ta triệu gọi hàm tạo *FormBuilder* gồm tất cả các trường bắt buộc và sau đó chúng ta cần gọi tập các phương thức được yêu cầu trên nó và cuối cùng phương thức *build* để nhận đối tượng *form*.

6.12.5 Khi nào sử dụng mẫu Builder

Sử dụng mẫu *Builder* khi

- Thuật toán để tạo đối tượng phức tạp cần là độc lập khỏi các phần mà tạo nên đối tượng và chúng được lắp ghép như thế nào
- Quá trình tạo cần cho phép các thể hiện khác nhau cho các đối tượng mà được tạo.

6.13 Mẫu thiết kế FACTORY METHOD

6.13.1 Mẫu thiết kế Factory Method

Trong thế giới ngày nay, mỗi người sử dụng phần mềm hỗ trợ công việc của họ. Mới đây, một công ty sản xuất buộc phải thay đổi cách họ sử dụng để nhận đơn đặt hàng của khách hàng. Công ty bây giờ xem xét việc sử dụng ứng dụng để nhận đơn đặt hàng từ khách hàng. Họ nhận đơn đặt, lỗi đơn đặt, phản hồi đơn đặt trước và trả lời đơn đặt dưới định dạng XML. Công ty yêu cầu bạn phát triển ứng dụng và hiển thị kết quả cho họ.

Thách thức chính đối với bạn là phân tích thông điệp XML và hiển thị nội dung của nó cho người sử dụng. Có các định dạng XML khác nhau phụ thuộc vào các kiểu khác nhau của thông điệp mà công ty nhận được từ khách hàng. Chẳng hạn, XML kiểu đơn có các tập xml tag khác khi so sánh với XML trả lời hoặc XML lỗi. Nhưng công việc lỗi là như nhau, đó là hiển thị cho người sử dụng thông điệp được chuyển tải trong XML.

Mặc dù công việc lỗi là như nhau, đối tượng này cần được sử dụng đa dạng tùy theo kiểu của XML, mà ứng dụng này nhận được từ khách hàng. Như vậy, đối tượng ứng dụng có vẻ chỉ biết rằng nó cần truy cập đến một lớp từ bên trong một phân cấp lớp (phân cấp của các bộ phân tích cú pháp khác nhau), nhưng không biết chính xác lớp nào từ trong tập các lớp con của lớp cha sẽ được chọn.

Trong trường hợp này, tốt hơn là cung cấp một nhà máy - factory để tạo ra các bộ phân tích, và trong thời gian chạy một parser sẽ được khởi tạo để thực hiện công việc đó, tùy theo kiểu của XML, mà ứng dụng nhận được từ người sử dụng.

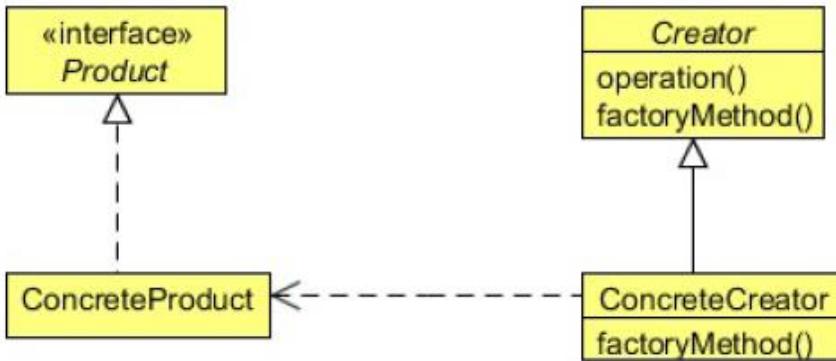
Mẫu Factory Method, phù hợp với tình huống này, định nghĩa một giao diện để tạo ra đối tượng, nhưng cho phép các lớp con quyết định lớp nào được khởi tạo. Factory Method cho phép lớp trì hoãn khởi tạo của lớp con.

Chúng ta xem xét chi tiết về mẫu Factory Method và sau đó sẽ sử dụng nó cài đặt bộ phân tích XML cho ứng dụng.

6.13.2 Mẫu Factory Method là gì

Mẫu *Factory Method* cho chúng ta cách đóng gói khởi tạo các kiểu cụ thể. Mẫu *Factory Method* đóng gói chức năng được yêu cầu để chọn và khởi tạo một lớp phù hợp, bên trong một phương thức chỉ định được tham chiếu như phương thức *factory method*. *Factory Method* chọn một lớp phù hợp của phân cấp lớp dựa trên ngữ cảnh của ứng dụng và các yếu tố ảnh hưởng khác. Sau đó nó khởi tạo lớp được chọn và trả về nó như khởi tạo của kiểu lớp cha.

Ưu điểm của cách tiếp cận này là các đối tượng ứng dụng có thể sử dụng phương thức *Factory method* để nhận truy cập khởi tạo của lớp phù hợp. Nó loại bỏ nhu cầu mà đối tượng ứng dụng cần xử lý với đa dạng tiêu chí lựa chọn lớp.



Product

- Định nghĩa giao diện của các đối tượng mà *factory method* tạo.

Concrete Product

- Cài đặt giao diện *Product*

Creator

- Khai báo *factory method*, mà trả về đối tượng kiểu *Product*. *Creator* cũng định nghĩa cài đặt mặc định của phương thức *factory method* mà trả về đối tượng *ConcreteProduct* mặc định.
- Có thể gọi phương thức *factory method* để tạo đối tượng *Product*.

ConcreteCreator

- Ghi đè *factory method* để trả về khởi tạo *ConcreteProduct*.

Phương thức *Factory Method* loại bỏ cần thiết gắn kết các lớp chuyên biệt ứng dụng với code của bạn. Code này chỉ làm việc với giao diện *Product*, do đó nó có thể làm việc với bất cứ các lớp *ConcreteProduct* mà người sử dụng định nghĩa nào khác.

6.13.3 Cài đặt mẫu Factory Method

Để cài đặt giải pháp cho ứng dụng đã bàn luận trên, trước hết ta kiểm tra sản phẩm ta có

```
package com.javacodegeeks.patterns.factorymethodpattern;

public interface XMLParser {

    public String parse();

}
```

Giao diện trên sẽ được sử dụng bởi các bộ phân tích XML khác nhau.

```
package com.javacodegeeks.patterns.factorymethodpattern;

public class ErrorXMLParser implements XMLParser{

    @Override
    public String parse() {

        System.out.println("Parsing error XML...");
        return "Error XML Message";
    }

}
```

ErrorXMLParser cài đặt *XMLParser* và được sử dụng để phân tích thông điệp XML lỗi.

```
package com.javacodegeeks.patterns.factorymethodpattern;

public class FeedbackXML implements XMLParser{

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

}
```

Lớp trên được sử dụng để phân tích thông điệp XML phản hồi.

Các bộ phân tích XML khác là:

```
package com.javacodegeeks.patterns.factorymethodpattern;

public class OrderXMLParser implements XMLParser{

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

}

package com.javacodegeeks.patterns.factorymethodpattern;

public class ResponseXMLParser implements XMLParser{

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

}
```

Để hiển thị các thông điệp được phân tích từ các bộ phân tích, một lớp dịch vụ trùu tượng được tạo mà sẽ mở rộng bởi dịch vụ chuyên biệt như phân tích chuyên biệt, hiện thị các lớp.

```
package com.javacodegeeks.patterns.factorymethodpattern;

public abstract class DisplayService {

    public void display(){
        XMLParser parser = getParser();
        String msg = parser.parse();
        System.out.println(msg);
    }

    protected abstract XMLParser getParser();

}
```

Lớp trên được sử dụng để hiển thị thông điệp được đưa ra từ bộ phân tích XML cho người sử dụng. Lớp trên là lớp trùu tượng mà chứa hai phương thức quan trọng. Phương thức *display* được dùng để hiển thị thông điệp cho người sử dụng. Phương thức *getParser* là *Factory method* mà được cài đặt bởi các lớp con để khởi tạo các đối tượng phân tích và phương thức này được sử dụng bởi phương thức *display* để phân tích XML và nhận thông điệp để hiển thị.

Dưới đây là các lớp con của *DisplayService* mà cài đặt phương thức *getParser*.

```
package com.javacodegeeks.patterns.factorymethodpattern;

public class ErrorXMLDisplayService extends DisplayService{

    @Override
    public XMLParser getParser() {
        return new ErrorXMLParser();
    }

}

package com.javacodegeeks.patterns.factorymethodpattern;

public class FeedbackXMLDisplayService extends DisplayService{

    @Override
    public XMLParser getParser() {
        return new FeedbackXML();
    }

}

package com.javacodegeeks.patterns.factorymethodpattern;

public class OrderXMLDisplayService extends DisplayService{

    @Override
    public XMLParser getParser() {
        return new OrderXMLParser();
    }

}

package com.javacodegeeks.patterns.factorymethodpattern;

public class ResponseXMLDisplayService extends DisplayService{

    @Override
    public XMLParser getParser() {
        return new ResponseXMLParser();
    }

}
```

Bây giờ chúng ta sẽ kiểm tra phương thức *Factory Method*.

```
package com.javacodegeeks.patterns.factorymethodpattern;

public class TestFactoryMethodPattern {

    public static void main(String[] args) {
```

```
        DisplayService service = new FeedbackXMLDisplayService();
        service.display();

        service = new ErrorXMLDisplayService();
        service.display();

        service = new OrderXMLDisplayService();
        service.display();

        service = new ResponseXMLDisplayService();
        service.display();

    }

}
```

Nó cho kết quả đầu ra như sau:

```
Parsing feedback XML...
Feedback XML Message
Parsing error XML...
Error XML Message
Parsing order XML...
Order XML Message
Parsing response XML...
Response XML Message
```

Trong lớp trên, bạn có thể thấy rõ là bằng việc cho phép các lớp con cài đặt *factory method* để tạo ra các khởi tạo khác nhau của bộ phân tích mà có thể được sử dụng trong thời gian chạy tùy theo nhu cầu.

6.13.4 Khi nào sử dụng mẫu Factory Method

Sử dụng mẫu *Factory Method* khi

- Một lớp không biết trước được lớp của các đối tượng mà cần được tạo.
- Một lớp muốn các lớp con của nó đặc tả các đối tượng mà nó tạo.
- Các lớp ủy quyền tương ứng cho một trong số các lớp con hỗ trợ và bạn muốn cục bộ hóa hiểu biết này của lớp con hỗ trợ mà được ủy quyền.

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.

Abstract 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.

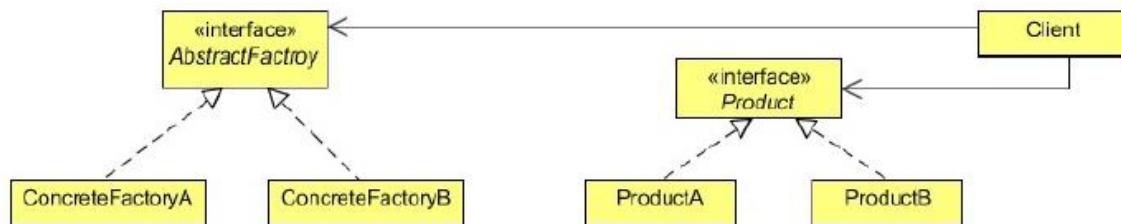
6.14.2 Mẫu Abstract Factory Method là gì

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 *Abstract 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 *factories* 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. Mọi *factory* cụ thể cài đặt giao diện của *abstract factory*.



AbstractFactory

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

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 "TWERRO": return new TWErrorXMLParser();
            case "TWFEEDBACK": return new TWFeedbackXMLParser ();
            case "TWORDER": return new TWORDerXMLParser();
            case "TWRESPONSE": return new TWResponseXMLParser();
        }

        return null;
    }

}
```

Hai *factory* 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("TWFEEBACK");
        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 *NYfactory* 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.

6.15 Mẫu thiết kế **PROTOTYPE**

6.15.1 Mở đầu

Trong lập trình hướng đối tượng, chúng ta cần các đối tượng để làm việc với chúng; các đối tượng tương tác với nhau để hoàn thành công việc. Nhưng đôi khi, tạo ra một đối tượng nặng có thể phải trả giá đắt, và nếu ứng dụng của bạn cần quá nhiều các đối tượng (chứa hầu hết các tính chất tương tự), nó có thể tạo ra một số vấn đề về hiệu năng.

Giả sử chúng ta xem xét kịch bản sau, ở đó ứng dụng yêu cầu kiểm soát quyền truy cập. Các đặc tính này của ứng dụng có thể được sử dụng bởi các người sử dụng tuân thủ theo quyền truy cập được cấp cho họ. Chẳng hạn, một số người sử dụng được quyền truy cập đến báo cáo được sinh bởi ứng dụng, trong khi một số khác thì không. Một số trong họ ngay cả có thể sửa báo cáo, trong khi một số chỉ có thể đọc nó. Một số người sử dụng cũng có quyền hành chính như bổ sung hoặc loại bỏ các người sử dụng khác.

Mỗi đối tượng người sử dụng sẽ có đối tượng kiểm soát quyền truy cập, mà được sử dụng để cung cấp hoặc hạn chế quyền truy cập của ứng dụng. Đối tượng kiểm soát quyền truy cập này là đối tượng đồ sộ, nặng và việc tạo ra nó rất tốn kém, vì nó yêu cầu dữ liệu được đẩy vào từ tài nguyên bên ngoài nào đó, như cơ sở dữ liệu hoặc một số file về sở hữu.

Chúng ta cũng không thể chia sẻ đối tượng quyền truy cập với người sử dụng cùng cấp độ, vì các quyền có thể thay đổi trong thời gian chạy bởi người quản trị và người sử dụng ở cùng cấp độ có thể có các quyền truy cập khác nhau. Một đối tượng người sử dụng có một đối tượng quyền truy cập.

Chúng ta có thể sử dụng mẫu thiết kế *Prototype* để giải quyết bài toán này bằng cách tạo các đối tượng kiểm soát truy cập trên mọi mức độ một lần và sau đó cung cấp bản sao của đối tượng này cho người sử dụng khi được yêu cầu. Trong trường hợp này, dữ liệu đẩy vào từ tài nguyên bên ngoài chỉ xảy ra một lần. Lần sau, đối tượng kiểm soát quyền truy cập được tạo bằng cách copy một đối tượng đã có. Đối tượng kiểm soát quyền truy cập này là không được tạo từ đầu mỗi lần yêu cầu được gửi đi, cách tiếp cận này sẽ giảm thời gian tạo đối tượng.

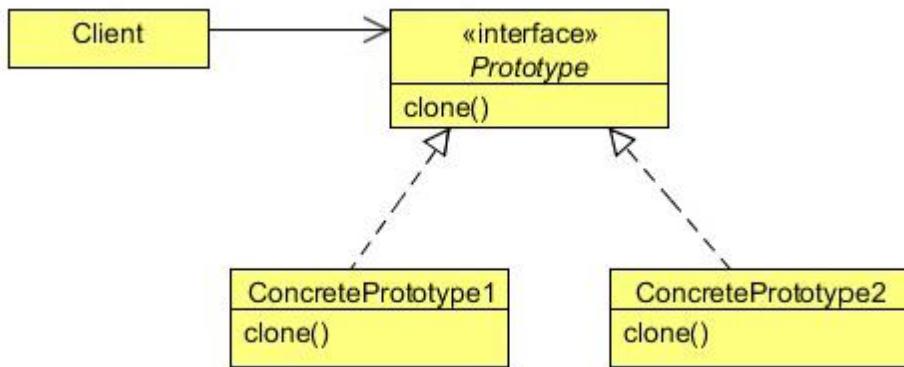
Trước khi đưa ra giải pháp, chúng ta sẽ tìm hiểu thêm về mẫu thiết kế *Prototype*.

6.15.2 Mẫu thiết kế Prototype là gì

Mẫu thiết kế *Prototype* được sử dụng để đặc tả mọi kiểu đối tượng được tạo sử dụng một khởi tạo nguyên mẫu và tạo đối tượng mới bằng cách copy bản mẫu này.

Khái niệm là copy một đối tượng đã có thay vì tạo mới một đối tượng từ đầu, đôi khi có thể bao gồm các thao tác đắt giá. Các đối tượng đã có đóng vai trò như nguyên mẫu và chứa trạng thái của đối tượng. Đối tượng mới được sao chép có thể thay đổi một số tính chất chỉ nếu nó được yêu cầu. Cách tiếp cận này tiết kiệm tài nguyên và thời gian đáng kể, đặc biệt khi việc tạo đối tượng là quá trình phức tạp.

Trong Java, có một số cách copy một đối tượng để tạo cái mới. Một cách đạt được điều này là sử dụng giao diện Cloneable. Java cung cấp phương thức clone, mà đối tượng kế thừa từ lớp Object. Bạn cần cài đặt giao diện Cloneable và viết đè phương thức clone này theo yêu cầu của bạn.



Prototype

- Khai báo một giao diện để sao chép chính nó

ConcretePrototype

- Cài đặt thao tác *clone* để sao chép chính nó

Client

- Tạo một đối tượng mới bằng cách yêu cầu một *prototype* *clone* chính nó

Prototype cho phép bạn kết hợp một lớp *product* cụ thể mới vào một hệ thống đơn giản bằng cách đăng ký một khởi tạo nguyên mẫu với *client*.

6.15.3 Giải pháp cho bài toán

Trong lời giải này, chúng ta sẽ sử dụng phương thức *clone* để giải quyết bài toán trên

```
package com.javacodegeeks.patterns.prototypepattern;

public interface Prototype extends Cloneable {
    public AccessControl clone() throws CloneNotSupportedException;
}
```

Giao diện trên mở rộng giao diện *Cloneable* và chứa phương thức *clone*. Giao diện này được cài đặt bởi các lớp mà muốn tạo đối tượng nguyên mẫu *prototype*.

```
package com.javacodegeeks.patterns.prototypepattern;

public class AccessControl implements Prototype {

    private final String controlLevel;
    private String access;

    public AccessControl(String controlLevel, String access) {
        this.controlLevel = controlLevel;
        this.access = access;
    }

    @Override
    public AccessControl clone() {
        try {
            return (AccessControl) super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return null;
    }

    public String getControlLevel() {
        return controlLevel;
    }

    public String getAccess() {
        return access;
    }

    public void setAccess(String access) {
        this.access = access;
    }

}
```

Lớp *AccessControl* cài đặt giao diện *Prototype* và ghi đè phương thức *clone*. Phương thức này gọi phương thức *clone* của lớp cha và trả về đối tượng sau khi ép xuống về kiểu *AccessControl*. Phương thức *clone* đưa ra báo lỗi *CloneNotSupportedException* bên trong bản thân phương thức.

Lớp này cũng chứa hai tính chất; *controlLevel* được sử dụng để chuyên biệt mức kiểm soát mà đối tượng chứa. Mức này phụ thuộc vào kiểu người sử dụng mà dùng nó, chẳng hạn, USER, ADMIN, MANAGER.

Ma trận kiểm soát quyền truy cập

	Program1	...	SegmentA	SegmentB
Process1	Read Execute		Read Execute	
Process2				Read

Tính chất khác là *access*. Nó chứa quyền truy cập cho người sử dụng. Lưu ý rằng, ở đây để đơn giản, chúng ta sử dụng *access* như thuộc tính kiểu *String*. Nó có thể là kiểu *Map* mà có thể chứa cặp khóa-giá trị của danh sách dài quyền truy cập được trao cho người sử dụng.

```
package com.javacodegeeks.patterns.prototypepattern;

public class User {

    private String userName;
    private String level;
    private AccessControl accessControl;

    public User(String userName, String level, AccessControl accessControl) {
        this.userName = userName;
        this.level = level;
        this.accessControl = accessControl;
    }

    public String getUserName() {
        return userName;
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }
    public String getLevel() {
        return level;
    }
    public void setLevel(String level) {
        this.level = level;
    }

    public AccessControl getAccessControl() {
        return accessControl;
    }
    public void setAccessControl(AccessControl accessControl) {
        this.accessControl = accessControl;
    }

    @Override
    public String toString() {
        return "Name: "+userName+", Level: "+level+", Access Control Level:"+←
            accessControl.getControlLevel()+"+, Access: "+accessControl.getAccess();
    }
}
```

Lớp *User* có *username*, *level* và một tham chiếu đến *AccessControl* được gán cho nó.

Chúng ta đã sử dụng lớp *AccessControlProvider* mà tạo và lưu giữ các đối tượng *AccessControl* từ trước. Và khi có yêu cầu đến đối tượng *AccessControl*, nó trả về đối tượng mới được tạo bởi *copy* các nguyên mẫu đã lưu.

```
package com.javacodegeeks.patterns.prototypepattern;

import java.util.HashMap;
import java.util.Map;

public class AccessControlProvider {

    private static Map<String, AccessControl>map = new HashMap<String, AccessControl>() {
        ;

        static{
            System.out.println("Fetching data from external resources and creating access control objects...");
            map.put("USER", new AccessControl("USER", "DO_WORK"));
            map.put("ADMIN", new AccessControl("ADMIN", "ADD/REMOVE USERS"));
            map.put("MANAGER", new AccessControl("MANAGER", "GENERATE/READ REPORTS"));
            map.put("VP", new AccessControl("VP", "MODIFY REPORTS"));
        }

        public static AccessControl getAccessControlObject(String controlLevel){
            AccessControl ac = null;
            ac = map.get(controlLevel);
            if(ac!=null){
                return ac.clone();
            }
            return null;
        }
    }
}
```

Phương thức *getAccessControlObject* đưa ra một đối tượng nguyên mẫu được lưu giữ tùy theo *controlLevel* truyền cho nó, từ map đó và trả về đối tượng được nhân bản tạo mới cho code của client.

Bây giờ chúng ta kiểm tra code trên.

```
package com.javacodegeeks.patterns.prototypepattern;

public class TestPrototypePattern {

    public static void main(String[] args) {
        AccessControl userAccessControl = AccessControlProvider.getAccessControlObject("USER");
        User user = new User("User A", "USER Level", userAccessControl);
    }
}
```

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

userAccessControl = AccessControlProvider.getAccessControlObject ("USER");
user = new User("User B", "USER Level", userAccessControl);
System.out.println("Changing access control of: "+user.getUserName());
user.getAccessControl().setAccess("READ REPORTS");
System.out.println(user);

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

AccessControl managerAccessControl = AccessControlProvider. ←
    getAccessControlObject ("MANAGER");
user = new User("User C", "MANAGER Level", managerAccessControl);
System.out.println(user);
}

}
```

Nó cho kết quả đầu ra như sau:

```
Fetching data from external resources and creating access control objects...
*****
Name: User A, Level: USER Level, Access Control Level:USER, Access: DO_WORK
Changing access of: User B
Name: User B, Level: USER Level, Access Control Level:USER, Access: READ REPORTS
*****
Name: User C, Level: MANAGER Level, Access Control Level:MANAGER, Access: GENERATE/READ ←
    REPORTS
```

Trong code trên, chúng ta đã tạo đối tượng *AccessControl* tại mức USER và gán nó cho *User A*. Sau đó lại đổi tượng *AccessControl* khác *User B*, nhưng lần này chúng ta thay đổi quyền truy cập của *User B*. Cuối cùng, quyền truy cập mức MANAGER cho *User C*.

Đối tượng *getAccessControlObject* được sử dụng để lấy bản sao mới của đối tượng *AccessControl* và điều này được nhìn thấy rõ khi chúng ta thay đổi quyền truy cập của *User B*, quyền truy cập của *User A* không thay đổi. Điều này được khẳng định rằng phương thức *clone* làm việc đúng, như nó trả về bản copy mới của đối tượng chứ không phải tham chiếu mà trả đến cùng một đối tượng.

6.15.4 Khi nào dùng mẫu thiết kế Prototype

Sử dụng mẫu thiết kế *Prototype* khi một hệ thống cần là độc lập với việc các sản phẩm của nó được tạo, kết hợp và thể hiện như thế nào và

- Khi các lớp để khởi tạo được đặc tả trong thời gian thực, chẳng hạn, bằng tay động hoặc
- Để tránh việc xây dựng một phân cấp lớp các factory mà chạy song song với phân cấp lớp của sản phẩm.
- Khi các khởi tạo của một lớp có thể có một trong chỉ một số ít kết hợp khác nhau của trạng thái. Nó có thể là thuận tiện hơn để cài đặt một số tương ứng các nguyên mẫu và nhân bản clone chúng hơn là khởi tạo lớp bằng tay, mỗi lần với một trạng thái phù hợp.

6.16 Mẫu thiết kế MEMENTO

6.16.1 Mở đầu

Đôi khi cần thiết ghi lại trạng thái bên trong của đối tượng. Điều này được đòi hỏi khi cài đặt các điểm kiểm tra *checkpoints* và cơ chế “*undo*” mà cho phép người sử dụng quay lại các thao tác tạm thời hoặc khôi phục từ lỗi. Bạn cần lưu thông tin trạng thái ở đâu đó, sao cho bạn có thể khôi phục đối tượng về điều kiện trước đó của chúng. Nhưng các đối tượng thường đóng gói một số hoặc tất cả về trạng thái của chúng, làm cho nó không truy cập được đến các đối tượng này và không thể lưu từ bên ngoài. Mở trạng thái đó có thể vi phạm tính đóng gói, mà cần được thỏa hiệp tính tin cậy và tính mở rộng được của ứng dụng.

Mẫu *Memento* có thể được sử dụng để thực hiện điều đó mà không cần mở cấu trúc bên trong của đối tượng. Đối tượng mà trạng thái cần nắm bắt được tham chiếu như gốc *originator*.

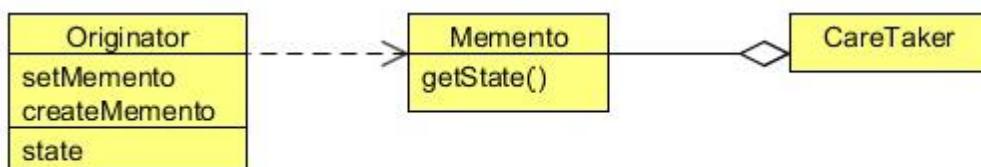
Để minh họa sử dụng mẫu *Memento*, chúng ta xét ví dụ sau. Chúng ta tạo lớp mà chứa hai trường kiểu số thực và chúng ta sẽ chạy một số phép toán trên nó. Chúng ta sẽ cung cấp cho người sử dụng thao tác *undo*. Nếu kết quả sau một số phép toán mà không thỏa mãn người sử dụng, thì người sử dụng có thể gọi thao tác *undo* mà sẽ khôi phục trạng thái của đối tượng về điểm lưu trước đó.

Ví dụ cũng sẽ bao gồm cả cơ chế điểm sao lưu mà được sử dụng bởi người sử dụng để lưu trạng thái của đối tượng. Chúng ta cũng sẽ cung cấp sự đa dạng của thao tác *undo*. *Undo* đơn giản có thể khôi phục trạng thái đối tượng về điểm lưu trước đó. *Undo* với điểm lưu được chỉ rõ có thể khôi phục một trạng thái cụ thể của đối tượng đó và *undo all* sẽ xóa mọi trạng thái lưu của đối tượng và khôi phục đối tượng trong trạng thái khởi tạo, khi đối tượng được tạo ra.

Trước khi cài đặt mẫu, chúng ta sẽ tìm hiểu mẫu thiết kế *Memento*.

6.16.2 Mẫu thiết kế Memento là gì

Mục đích của mẫu *Memento* là không vi phạm tính đóng gói, hãy nắm bắt và xuất ra trạng thái bên trong của đối tượng sao cho đối tượng đó có thể được khôi phục đến trạng thái này về sau.



Memento

- Lưu trạng thái bên trong của đối tượng gốc *Originator*. *Memento* có thể lưu giữ nhiều hoặc ít về trạng thái bên trong *originator* như cần thiết theo quyết định của *originator*.
- Bảo vệ chống truy cập bởi các đối tượng khác ngoài *originator*. *Memento* có hai giao diện hiệu quả. Người chăm lo thấy giao diện hẹp đến *Memento* nó có thể truyền *memento* cho các đối tượng khác. *Originator*, ngược lại, nhìn thấy giao diện rộng, người ta cho phép nó truy cập mọi dữ liệu cần thiết để khôi phục bản thân nó về trạng thái trước đó của nó. Ly tưởng là, chỉ *originator* mà tạo ra *memento* có thể được cho phép truy cập trạng thái bên trong của *memento*.

Originator

- Tạo *memento* chứa bản chụp trạng thái bên trong hiện thời của nó
- Sử dụng *memento* để khôi phục trạng thái bên trong của nó.

Người chăm lo

- Có trách nhiệm lưu giữ *memento*.
- Không bao giờ thao tác trên hoặc kiểm tra nội dung của *memento*.

Khi *client* muốn lưu trạng thái của *originator*, nó yêu cầu trạng thái hiện tại từ *originator*. *Originator* lưu giữ mọi thuộc tính mà được yêu cầu để khôi phục trạng thái của nó trong đối tượng riêng biệt được tham chiếu như *Memento* và trả về nó cho *client*. Như vậy *Memento* có thể được xem như một đối tượng mà chứa trạng thái bên trong của một đối tượng khác tại một thời điểm nhất định. Đối tượng *Memento* cần che giấu giá trị biến đổi của *originator* khỏi mọi đối tượng trừ *originator*. Nói cách khác, nó cần phải bảo vệ trạng thái bên trong của nó chống việc truy cập bởi các đối tượng khác ngoài *originator*. Từ nay về sau, *Memento* cần được thiết kế để cung cấp truy cập hạn chế cho các đối tượng khác, trong khi *originator* được cho phép truy cập đến trạng thái bên trong của nó.

Khi *client* muốn khôi phục *originator* trở lại trạng thái trước đó, nó đơn giản truyền *memento* trở lại cho *originator*. *Originator* sử dụng thông tin trạng thái này trong *memento* và đẩy bản thân nó trở về trạng thái được lưu trong đối tượng *memento*.

6.16.3 Cài đặt mẫu Memento

```
package com.javacodegeeks.patterns.mementopattern;

public class Originator {

    private double x;
    private double y;

    private String lastUndoSavepoint;
    CareTaker careTaker;

    public Originator(double x, double y,CareTaker careTaker){
        this.x = x;
        this.y = y;

        this.careTaker = careTaker;

        createSavepoint("INITIAL");
    }

    public double getX(){
        return x;
    }

    public double getY(){
        return y;
    }
}
```

```
public void setX(double x) {
    this.x = x;
}

public void setY(double y) {
    this.y = y;
}

public void createSavepoint(String savepointName) {
    careTaker.saveMemento(new Memento(this.x, this.y), savepointName);
    lastUndoSavepoint = savepointName;
}

public void undo(){
    setOriginatorState(lastUndoSavepoint);
}

public void undo(String savepointName){
    setOriginatorState(savepointName);
}

public void undoAll(){
    setOriginatorState("INITIAL");
    careTaker.clearSavepoints();
}

private void setOriginatorState(String savepointName){
    Memento mem = careTaker.getMemento(savepointName);
    this.x = mem.getX();
    this.y = mem.getY();
}

@Override
public String toString(){
    return "X: "+x+", Y: "+y;
}
}
```

Trên đây là lớp *originator*, trạng thái đối tượng của nó cần được lưu trong *memento*. Lớp này chứa hai trường số thực x và y, và cũng có một tham chiếu của người chăm lo *CareTaker*. *CareTaker* được sử dụng để lưu và truy vấn đối tượng *memento* mà biểu diễn trạng thái của đối tượng *originator*.

Trong hàm tạo, chúng ta có lưu trạng thái ban đầu của đối tượng sử dụng phương thức *createSavepoint*. Phương thức này tạo đối tượng *memento* và yêu cầu *CareTaker* quan tâm đến đối tượng. Chúng ta đã sử dụng biến *LastUndoSavepoint* mà được sử dụng để lưu tên chính của *memento* được lưu lần cuối để cài đặt thao tác *undo*.

Lớp này cung cấp các kiểu của thao tác *undo*. Phương thức *undo* không có tham số khôi phục trạng thái lưu cuối cùng, *undo* với tên *savepoint* như tham số khôi phục trạng thái được lưu với tên *savepoint* cụ thể đó. Phương thức *undoAll* yêu cầu *CareTaker* xóa mọi điểm *savepoints* và đặt nó vào trạng thái ban đầu (trạng thái tại thời điểm khởi tạo đối tượng đó).

```
package com.javacodegeeks.patterns.mementopattern;

public class Memento {
    private double x;
    private double y;

    public Memento(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }
}
```

Lớp *Memento* được sử dụng để lưu trạng thái *originator* và lưu giữ bởi *CareTaker*. Lớp này không có phương set nào cả, nó chỉ được sử dụng để get trạng thái của đối tượng.

```
package com.javacodegeeks.patterns.mementopattern;

import java.util.HashMap;
import java.util.Map;

public class CareTaker {
    private final Map<String, Memento> savepointStorage = new HashMap<String, Memento>();
    ;

    public void saveMemento(Memento memento, String savepointName) {
        System.out.println("Saving state..." + savepointName);
        savepointStorage.put(savepointName, memento);
    }

    public Memento getMemento(String savepointName) {
        System.out.println("Undo at ..." + savepointName);
        return savepointStorage.get(savepointName);
    }

    public void clearSavepoints() {
        System.out.println("Clearing all save points...");
        savepointStorage.clear();
    }
}
```

Lớp trên là người chăm lo *CareTaker* được sử dụng để lưu và cung cấp đối tượng *memento* được yêu cầu. Lớp này chứa phương thức *saveMemento* được sử dụng để lưu đối tượng *Memento*, phương thức *getMemento* được sử dụng để trả về đối tượng *memento* yêu cầu và phương thức *clearSavepoints* mà được sử dụng để xóa mọi điểm *savepoints* và nó xóa các đối tượng *memento* đã lưu.

Bây giờ chúng ta kiểm tra ví dụ trên.

```
package com.javacodegeeks.patterns.mementopattern;

public class TestMementoPattern {

    public static void main(String[] args) {

        CareTaker careTaker = new CareTaker();
        Originator originator = new Originator(5, 10, careTaker);

        System.out.println("Default State: "+originator);

        originator.setX(originator.getY()*51);

        System.out.println("State: "+originator);
        originator.createSavepoint ("SAVE1");

        originator.setY(originator.getX()/22);
        System.out.println("State: "+originator);

        originator.undo();
        System.out.println("State after undo: "+originator);

        originator.setX(Math.pow(originator.getX(), 3));
        originator.createSavepoint ("SAVE2");
        System.out.println("State: "+originator);
        originator.setY(originator.getX()-30);
        originator.createSavepoint ("SAVE3");
        System.out.println("State: "+originator);
        originator.setY(originator.getX()/22);
        originator.createSavepoint ("SAVE4");
        System.out.println("State: "+originator);

        originator.undo("SAVE2");
        System.out.println("Retrieving at: "+originator);

        originator.undoAll();
        System.out.println("State after undo all: "+originator);
    }
}
```

Nó cho kết quả đầu ra như sau:

```
Saving state...INITIAL
Default State: X: 5.0, Y: 10.0
State: X: 510.0, Y: 10.0
Saving state...SAVE1
State: X: 510.0, Y: 23.181818181818183
Undo at ...SAVE1
State after undo: X: 510.0, Y: 10.0
Saving state...SAVE2
State: X: 1.32651E8, Y: 10.0
Saving state...SAVE3
State: X: 1.32651E8, Y: 1.3265097E8
Saving state...SAVE4
State: X: 1.32651E8, Y: 6029590.909090909
Undo at ...SAVE2
Retrieving at: X: 1.32651E8, Y: 10.0
Undo at ...INITIAL
Clearing all save points...
State after undo all: X: 5.0, Y: 10.0
```

Trong lớp trên, chúng ta đã tạo đối tượng *CareTaker* và sau đó gán nó cho đối tượng *Originator*. Và chúng ta đặt giá trị của x và y tương ứng là 5 và 10. Sau đó, chúng ta áp dụng một thao tác nào đó trên x và lưu trạng thái của đối tượng là “SAVE1”.

Sau một số thao tác, chúng ta gọi phương thức *undo* và khôi phục trạng thái cuois của đối tượng mà rõ ràng được chỉ ra ở đầu ra. Sau đó ta áp dụng một vài thao tác và lại lưu các trạng thái của đối tượng như “SAVE2”, “SAVE3”, và “SAVE4”.

Sau đó chúng ta yêu cầu *originator* khôi phục trạng thái ‘SAVE2’ và gọi phương thức *undoAll* mà đặt lại trạng thái ban đầu của đối tượng và xóa mọi *savepoints*.

Lưu ý rằng, trong ví dụ trên, *Originator* là có trách nhiệm cung cấp *memento* của nó cho *CareTaker*. Lý do là như vậy chúng ta không muốn trao trách nhiệm này cho người sử dụng. Trong ví dụ của chúng ta, người sử dụng chỉ có thể đòi hỏi yêu cầu cho *savepoint* và khôi phục trạng thái của đối tượng. Trong nhiều trường hợp, *CareTaker* thao tác bên ngoài *originator* bởi một lớp khác (như được chỉ ra trong sơ đồ lớp bên trên).

6.16.4 Khi nào dùng mẫu Memento

Sử dụng mẫu *Memento* trong các trường hợp sau:

- Bản chụp nhanh trạng thái (hoặc một phần) của đối tượng cần được lưu sao cho nó có thể được khôi phục về trạng thái này về sau và
- Một giao diện trực tiếp để nhận trạng thái có thể mở ra chi tiết cài đặt và phá vỡ tính đóng gói của đối tượng.

6.17 Mẫu thiết kế TEMPLATE

6.17.1 Mở đầu

Mẫu thiết kế Template là mẫu hành vi và như tên nó đề xuất, nó cung cấp bản nháp template hoặc cấu trúc của một thuật toán mà được sử dụng bởi user. Người sử dụng cung cấp cài đặt của riêng họ mà không thay đổi cấu trúc của thuật toán.

Sẽ dễ hiểu hơn mẫu này thông qua một ví dụ. Chúng ta sẽ hiểu kịch bản trong mục này và sẽ cài đặt lời giải sử dụng mẫu Template trong mục sau nữa.

Đã khi nào bạn kết nối với cơ sở dữ liệu quan hệ sử dụng ứng dụng Java chưa. Thủ nhắc lại một số bước quan trọng mà được yêu cầu để kết nối và chèn dữ liệu vào cơ sở dữ liệu. Trước hết, bạn cần một driver phù hợp với cơ sở dữ liệu mà chúng ta muốn kết nối với nó. Sau đó, chúng ta truyền một số chứng từ ủy nhiệm cho cơ sở dữ liệu, và chúng ta chuẩn bị lệnh, set dữ liệu cho lệnh chèn và insert sử dụng lệnh chèn. Cuối cùng ta đóng mọi kết nối và tùy chọn xóa mọi đối tượng kết nối.

Bạn cần phải viết mọi bước này bắt kể cơ sở dữ liệu quan hệ của nhà cung cấp nào. Xem xét bài toán ở đó chèn dữ liệu nào đó vào các cơ sở dữ liệu khác nhau. Bạn cần đầy dữ liệu từ một file CSV (file ở đó các giá trị được phân cách bởi dấu phẩy) và phải chèn vào cơ sở dữ liệu MySQL. Một số dữ liệu đến từ file text và nó cần được chèn vào cơ sở dữ liệu Oracle. Sự khác biệt chỉ ở driver và dữ liệu, còn lại là các bước giống nhau, như JDBC cung cấp tập chung các giao diện để trao đổi với cơ sở dữ liệu quan hệ chuyên biệt của bất kỳ nhà cung cấp nào.

Chúng ta có thể tạo một bản nháp template mà sẽ thực hiện một số bước cho client và chúng ta sẽ để lại một số bước cho phép client cài đặt chúng theo cách của riêng họ. Tùy chọn, client có thể ghi đè hành vi mặc định của một số bước đã được định nghĩa trước.

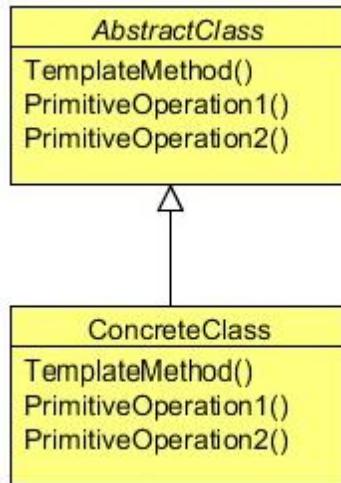
Bây giờ trước khi cài đặt code, chúng ta sẽ tìm hiểu thêm về mẫu thiết kế *Template*.

6.17.2 Mẫu Template là gì

Mẫu *Template* định nghĩa bộ khung của một thuật toán trong thao tác mà trì hoãn một số bước cho các lớp con. Phương thức *Template* cho phép các lớp con định nghĩa lại một số bước của một thuật toán mà không thay đổi cấu trúc của thuật toán.

Mẫu *Template Method* có thể được sử dụng trong một số tình huống khi mà có một thuật toán, một số bước của nó cần được cài đặt theo nhiều cách khác nhau. Trong kịch bản này, mẫu *Template Method* đề xuất giữ bộ khung của thuật toán trong một phương thức riêng biệt được tham chiếu như *Template Method* bên trong một lớp mà có thể được tham chiếu như *Template Class*, gác lại cài đặt chuyên biệt một số phần khác nhau (các bước mà có thể được cài đặt theo các cách khác nhau) của thuật toán cho các lớp con của lớp đó.

Lớp *Template* không cần thiết phải gác lại cài đặt cho các lớp con một cách toàn bộ. Thay vào đó, như một phần cung cấp bộ khung của thuật toán, lớp *Template* có thể cũng cung cấp một số phần cài đặt mà có thể được coi là bất biến qua các cài đặt khác nhau. Nó có thể ngay cả cung cấp cài đặt mặc định cho một số phần thay đổi, nếu nó phù hợp. Chi tiết chuyên biệt sẽ được cài đặt bên trong các lớp con. Kiểu cài đặt này loại bỏ cần thiết lặp lại code mà có nghĩa là khói lượng tối thiểu code được viết.



AbstractClass

- Định nghĩa các thao tác nguyên thủy trừu tượng mà các lớp con cụ thể sẽ cài đặt các bước của thuật toán

ConcreteClass.

- Cài đặt *Template Method* xác định bộ khung của thuật toán. *Template method* gọi các thao tác nguyên thủy là các thao tác được định nghĩa trong *AbstractClass* hoặc các thao tác của các đối tượng *ConcreteClass* khác
- Cài đặt các thao tác nguyên thủy để chạy.

6.17.3 Cài đặt mẫu thiết kế Template

Dưới đây chúng ta có thể thấy lớp *Template* kết nối được sử dụng để cung cấp *template* cho *client* kết nối và trao đổi thông tin với các cơ sở dữ liệu khác nhau.

```
package com.javacodegeeks.patterns.templatepattern;

public abstract class ConnectionTemplate {

    public final void run() {
        setDBDriver();
        setCredentials();
        connect();
        prepareStatement();
        setData();
        insert();
        close();
        destroy();
    }

    public abstract void setDBDriver();

    public abstract void setCredentials();

    public void connect() {
        System.out.println("Setting connection...");
    }

    public void prepareStatement() {
        System.out.println("Preparing insert statement...");

    }

    public abstract void setData();

    public void insert() {
        System.out.println("Inserting data...");
    }

    public void close() {
        System.out.println("Closing connections...");
    }

    public void destroy() {
        System.out.println("Destroying connection objects...");
    }
}
```

Lớp trừu tượng cung cấp các bước để kết nối, trao đổi và sau này là đóng kết nối. Tất cả các bước này là được yêu cầu theo thứ tự đó để hoàn thành công việc. Lớp này cung cấp cài đặt mặc định cho một số bước chung và gác lại các bước chuyên biệt như trừu tượng mà buộc *client* phải cung cấp cài đặt cho chúng.

Phương thức *setDBDriver* cần được cài đặt bởi người sử dụng để cung cấp *Driver* chuyên biệt cho cơ sở dữ liệu. Các tài liệu chúng thực có thể là khác nhau đối với các cơ sở dữ liệu khác nhau, do đó *setCredentials* cũng được để lại như trừu tượng cho người sử dụng cài đặt chúng.

Tương tự, kết nối với cơ sở dữ liệu sử dụng *JDBC API* và chuẩn bị một lệnh là thường gấp. Nhưng, dữ liệu có thể chuyên biệt, vì người sử dụng sẽ tạo ra nó, và phần các bước còn lại như chạy lệnh *insert*, đóng kết nối, hủy đổi tượng là tương tự với mọi cơ sở dữ liệu, vì vậy cài đặt chúng sẽ giữ là chung bên trong *Template*.

Phương thức chính của lớp trên là phương thức *run*. Phương thức *run* được sử dụng để chạy các bước này theo thứ tự. Phương thức này được đặt là không đổi *final*, vì các bước cần phải được giữ an toàn và không được thay đổi bởi người sử dụng.

Hai lớp dưới đây mở rộng lớp *template* và cung cấp cài đặt chuyên biệt cho một số phương thức.

```
package com.javacodegeeks.patterns.templatepattern;

public class MySqlCSVCon extends ConnectionTemplate {

    @Override
    public void setDBDriver() {
        System.out.println("Setting MySQL DB drivers...");
    }

    @Override
    public void setCredentials() {
        System.out.println("Setting credentials for MySQL DB...");
    }

    @Override
    public void setData() {
        System.out.println("Setting up data from csv file....");
    }
}
```

Lớp trên được sử dụng để kết nối cơ sở dữ liệu MySQL và cung cấp dữ liệu bằng cách đọc file CSV.

```
package com.javacodegeeks.patterns.templatepattern;

public class OracleTxtCon extends ConnectionTemplate {

    @Override
    public void setDBDriver() {
        System.out.println("Setting Oracle DB drivers...");
    }

    @Override
    public void setCredentials() {
        System.out.println("Setting credentials for Oracle DB...");
    }

    @Override
    public void setData() {
        System.out.println("Setting up data from txt file....");
    }
}
```

Lớp trên được sử dụng để kết nối cơ sở dữ liệu Oracle và cung cấp dữ liệu bằng cách đọc file text.

Bây giờ chúng ta kiểm tra code trên.

```
package com.javacodegeeks.patterns.templatepattern;

public class TestTemplatePattern {

    public static void main(String[] args) {
        System.out.println("For MYSQL....");
        ConnectionTemplate template = new MySqlCSVCon();
        template.run();
        System.out.println("For Oracle....");
        template = new OracleTxtCon();
        template.run();
    }
}
```

Code trên cho kết quả đầu ra như sau:

```
For MYSQL....
Setting MySQL DB drivers...
Setting credentials for MySQL DB...
Setting connection...
Preparing insert statement...
Setting up data from csv file....
Inserting data...
Closing connections...
Destroying connection objects...

For Oracle...
Setting Oracle DB drivers...
Setting credentials for Oracle DB...
Setting connection...
Preparing insert statement...
Setting up data from txt file....
Inserting data...
Closing connections...
Destroying connection objects...
```

Đầu ra bên trên chỉ rõ mẫu *Template* làm việc để kết nối và trao đổi với cơ sở dữ liệu khác nhau sử dụng cách tương tự. Mẫu giữ code chung dưới một lớp và hỗ trợ tái sử dụng code. Nó đặt bộ khung và kiểm soát nó cho người sử dụng và cho phép người sử dụng mở rộng *template* theo thứ tự để cung cấp cài đặt chuyên biệt của chúng cho một số bước.

Bây giờ, nếu chúng ta phát triển ví dụ trên bằng cách bổ sung cơ chế ghi lưu lại. Nhưng một số người sử dụng code này không muốn tiện ích này, để cài đặt nó chúng ta có thể sử dụng một mèo nhỏ. Mèo là một phương thức bên trong lớp *Template* với hành vi mặc định, hành vi này có thể được sử dụng để thay đổi một số bước tùy chọn. Người sử dụng cần cài đặt phương thức này mà có thể tạo mèo bên trong lớp *Template* để thay đổi các bước tùy chọn của thuật toán.

6.17.4 Tạo mạo bên trong Template

Thử bổ sung vào code trên với mạo nhỏ:

```
package com.javacodegeeks.patterns.templatepattern;

import java.util.Date;

public abstract class ConnectionTemplate {

    private boolean isLoggingEnable = true;

    public ConnectionTemplate() {
        isLoggingEnable = disableLogging();
    }

    public final void run() {
        setDBDriver();
        logging("Drivers set [" + new Date() + "]");
        setCredentials();
        logging("Credentails set [" + new Date() + "]");
        connect();
        logging("Conencted");
        prepareStatement();
        logging("Statement prepared [" + new Date() + "]");
        setData();
        logging("Data set [" + new Date() + "]");
        insert();
        logging("Inserted [" + new Date() + "]");
        close();
        logging("Conenctions closed [" + new Date() + "]");
        destroy();
        logging("Object destoryed [" + new Date() + "]");
    }

    public abstract void setDBDriver();

    public abstract void setCredentials();

    public void connect() {
        System.out.println("Setting connection...");
    }

    public void prepareStatement() {
        System.out.println("Preparing insert statement...");
    }

    public abstract void setData();

    public void insert() {
        System.out.println("Inserting data...");
    }

    public void close() {
        System.out.println("Closing connections...");
    }

    public void destroy() {
        System.out.println("Destroying connection objects...");
    }

    public boolean disableLogging() {
```

```
        return true;
    }

    private void logging(String msg) {
        if (isLoggingEnable) {
            System.out.println("Logging....: " + msg);
        }
    }
}
```

Chúng ta đưa ra hai phương thức mới bên trong lớp *Template*. Phương thức *disableLogging* là mèo mà trả về giá trị Bool. Mặc định, giá trị Bool *isLoggingEnable*, mà làm cho có khả năng ghi lưu lại là true. Người sử dụng ghi đè phương thức này nếu ghi lưu lại cần phải là không thể có cho code của anh ta. Phương thức khác là phương thức *Private* được sử dụng để ghi lưu thông điệp.

Lớp dưới đây cài đặt phương thức mèo và trả về false, một số cái được tắt khỏi cơ chế ghi lưu cho công việc chuyên biệt đó.

```
package com.javacodegeeks.patterns.templatepattern;

public class MySqlCSVCon extends ConnectionTemplate {

    @Override
    public void setDBDriver() {
        System.out.println("Setting MySQL DB drivers...");
    }

    @Override
    public void setCredentials() {
        System.out.println("Setting credentials for MySQL DB...");
    }

    @Override
    public void setData() {
        System.out.println("Setting up data from csv file....");
    }

    @Override
    public boolean disableLogging() {
        return false;
    }
}
```

Hãy kiểm tra code trên

```
package com.javacodegeeks.patterns.templatepattern;

public class TestTemplatePattern {
    public static void main(String[] args) {
        System.out.println("For MYSQL....");
        ConnectionTemplate template = new MySqlCSVCon();
        template.run();
        System.out.println("For Oracle....");
        template = new OracleTxtCon();
        template.run();
    }
}
```

Lớp trên trả về kết quả sau.

```
For MYSQL....  
Setting MySQL DB drivers...  
Setting credentials for MySQL DB...  
  
Setting connection...  
Preparing insert statement...  
Setting up data from csv file....  
Inserting data...  
Closing connections...  
Destroying connection objects...  
  
For Oracle...  
Setting Oracle DB drivers...  
Logging....: Drivers set [Sat Nov 08 23:53:47 IST 2014]  
Setting credentials for Oracle DB...  
Logging....: Credentails set [Sat Nov 08 23:53:47 IST 2014]  
Setting connection...  
Logging....: Conencted  
Preparing insert statement...  
Logging....: Statement prepared [Sat Nov 08 23:53:47 IST 2014]  
Setting up data from txt file....  
Logging....: Data set [Sat Nov 08 23:53:47 IST 2014]  
Inserting data...  
Logging....: Inserted [Sat Nov 08 23:53:47 IST 2014]  
Closing connections...  
Logging....: Conenctions closed [Sat Nov 08 23:53:47 IST 2014]  
Destroying connection objects...  
Logging....: Object destoryed [Sat Nov 08 23:53:47 IST 2014]
```

Bạn có thể thấy rõ trong đâu ra, rằng logging được tắt cho cài đặt MySQL, trong khi đó là bật cho cài đặt Oracle.

6.17.5 Khi nào sử dụng mẫu Template

Mẫu *Template Method* có thể được sử dụng trong các trường hợp sau:

- Cài đặt các phần bắt biến của thuật toán một lần và gác lại cho các lớp con hành vi mà nó đa dạng.
- Khi hành vi chung giữa các lớp con cần được phân tách ra và địa phương hóa trong lớp chung để tránh bị lặp code. Trước hết bạn định danh sự khác biệt trong code đã có và sau đó tách sự khác biệt ra thành thao tác riêng. Cuối cùng bạn thay code trì hoãn với các phương thức *Template* mà gọi một trong các thao tác mới đó.
- Để kiểm soát sự mở rộng của các lớp con. Bạn có thể định nghĩa phương thức *template* mà gọi thao tác “mẹo” tại các điểm chuyên biệt, bằng cách ấy cho phép mở rộng chỉ tại các điểm đó.

6.18 Mẫu thiết kế STATE

6.18.1 Mở đầu

Để minh họa sử dụng mẫu thiết kế trạng thái, giả sử chúng ta giúp một công ty mà đang xây dựng một robot nấu ăn. Công ty muốn một con robot đơn giản mà chỉ là đi lại và nấu ăn. Người sử dụng cần vận hành con robot sử dụng tập các lệnh thông qua điều khiển từ xa. Hiện tại, con robot cần làm ba việc walk, cook hoặc nó có thể được tắt.

Công ty thiết lập các giao thức để định nghĩa chức năng của robot. Nếu robot đang ở trạng thái on, bạn có thể ra lệnh cho nó walk, nếu yêu cầu cook, trạng thái của nó chuyển sang “cook” hoặc nó ở trạng thái “off”, nếu nó được tắt đi.

Tương tự, khi đang ở trạng thái “cook”, nó có thể walk hoặc cook, nhưng không thể bị tắt. Và cuối cùng, khi đang ở trạng thái “off” nó sẽ tự động trở thành “on” và walk, khi người sử dụng ra lệnh cho nó walk, nhưng không thể cook ở trạng thái “off”.

Có thể trông là dễ cài đặt lớp robot với tập các phương thức walk, cook, off và các trạng thái như là on, cook và off. Bạn có thể sử dụng các rẽ nhánh if-else hoặc switch để cài đặt các giao thức thiết lập bởi công ty. Nhưng rất nhiều lệnh if-else hoặc switch sẽ tạo ra một cơn ác mộng bao trù và độ phức tạp sẽ tăng lên trong tương lai.

Bạn sẽ nghĩ là chúng ta có thể cài đặt với các lệnh if-then mà không gặp vấn đề gì, nhưng một sự thay đổi sẽ làm cho code trở nên phức tạp hơn. Yêu cầu này chỉ ra rõ ràng rằng hành vi của đối tượng là hoàn toàn dựa trên trạng thái của đối tượng. Chúng ta có thể sử dụng mẫu thiết kế State mà đóng gói các trạng thái của đối tượng vào lớp riêng khác và giữ lớp context độc lập với bất cứ thay đổi trạng thái nào.

Trước hết tìm hiểu về mẫu thiết kế State và sau đó chúng ta cài đặt nó để giải bài toán bên trên.

6.18.2 Mẫu thiết kế State là gì

Mẫu thiết kế *State* cho phép một đối tượng thay đổi hành vi của nó khi trạng thái bên trong của đối tượng thay đổi. Đối tượng sẽ dường như là thay đổi lớp của nó.

Trạng thái của đối tượng có thể được định nghĩa như điều kiện chính xác của nó tại một thời điểm cho trước, phụ thuộc vào giá trị của các tính chất hoặc các thuộc tính. Tập các phương thức được cài đặt bởi một lớp tạo thành hành vi của các khai tạo của nó. Khi có thay đổi giá trị của các thuộc tính của nó, chúng ta nói trạng thái của đối tượng đã thay đổi.

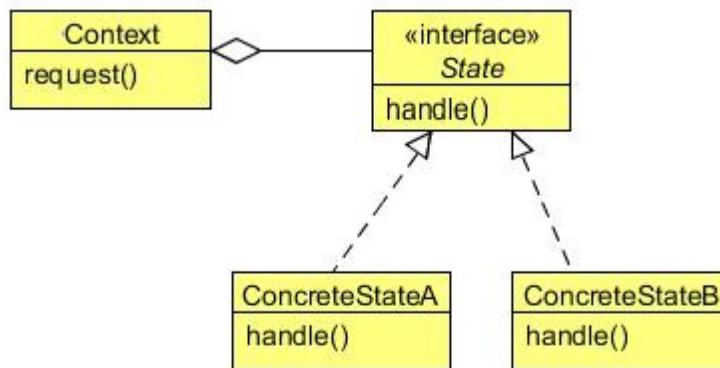
Mẫu *State* là hữu ích trong thiết kế cấu trúc hiệu quả cho một lớp, một khai tạo tiêu biểu của nó có thể tồn tại trong nhiều trạng thái khác nhau và thể hiện các hành vi khác nhau phụ thuộc vào trạng thái nào nó đang ở trong đó. Nói cách khác, trong trường hợp của một đối tượng của lớp như vậy, một số hoặc tất cả các hành vi là hoàn toàn bị ảnh hưởng bởi trạng thái hiện tại. Trong mẫu thiết kế *State* thuật ngữ, như một lớp được tham chiếu là lớp *Context*. Một đối

tượng *Context* có thể thay đổi hành vi của nó khi có sự thay đổi ở trạng thái bên trong và nó cũng được tham chiếu là đối tượng có trạng thái Stateful.

Mẫu trạng thái đề xuất chuyển hành vi chuyên biệt trạng thái ra khỏi lớp *Context* vào tập các lớp riêng được tham chiếu là các lớp *State*. Mỗi trong số nhiều trạng thái mà đối tượng *Context* có thể ở trong đó có thể được ánh xạ vào một lớp *State* riêng biệt. Việc cài đặt lớp *State* chứa hành vi *context* mà chuyên biệt cho *state* đã cho, không chứa hành vi tổng thể của bản thân *Context*. *Context* hoạt động như *client* đến tập các đối tượng *State* theo nghĩa là nó sử dụng các đối tượng trạng thái khác nhau để thể hiện hành vi chuyên biệt trạng thái cần thiết cho đối tượng ứng dụng mà sử dụng *context*.

Bằng việc đóng gói hành vi chuyên biệt trạng thái vào các lớp riêng, cài đặt *context* trở nên đơn giản hơn để đọc: không có quá nhiều các lệnh điều kiện như *if-else* hoặc cấu trúc *switch-case*. Khi một đối tượng *Context* được tạo đầu tiên, nó khởi tạo nó với đối tượng trạng thái *State* ban đầu. Đối tượng *State* này trở thành đối tượng trạng thái hiện tại cho *Context*. Bằng cách thay đổi trạng thái hiện tại với đối tượng trạng thái mới, *context* chuyển sang trạng thái mới.

Ứng dụng *client* sử dụng *context* không có trách nhiệm đặc tả đối tượng trạng thái hiện tại cho *context*, nhưng thay vào đó, mỗi lớp trạng thái mà thể hiện trạng thái chuyên biệt là được mong đợi để cung cấp cài đặt cần thiết để chuyển *context* sang trạng thái khác. Khi đối tượng ứng dụng gọi tới một phương thức của *context* (hành vi), nó chuyển tiếp lời gọi phương thức cho đối tượng trạng thái hiện tại.



Context

- Định nghĩa giao diện quan tâm cho client
- Duy trì một khởi tạo của một lớp con *ConcreteState* mà xác định trạng thái hiện tại.

State

- Định nghĩa giao diện để đóng gói hành vi gắn kết với một trạng thái cụ thể của *Context*

Các lớp con ConcreteState

- Mỗi lớp con thể hiện hành vi gắn kết với một trạng thái của *Context*

6.18.3 Cài đặt mẫu thiết kế State

Sau đây là giao diện RoboticState mà chưa hành vi của Robot.

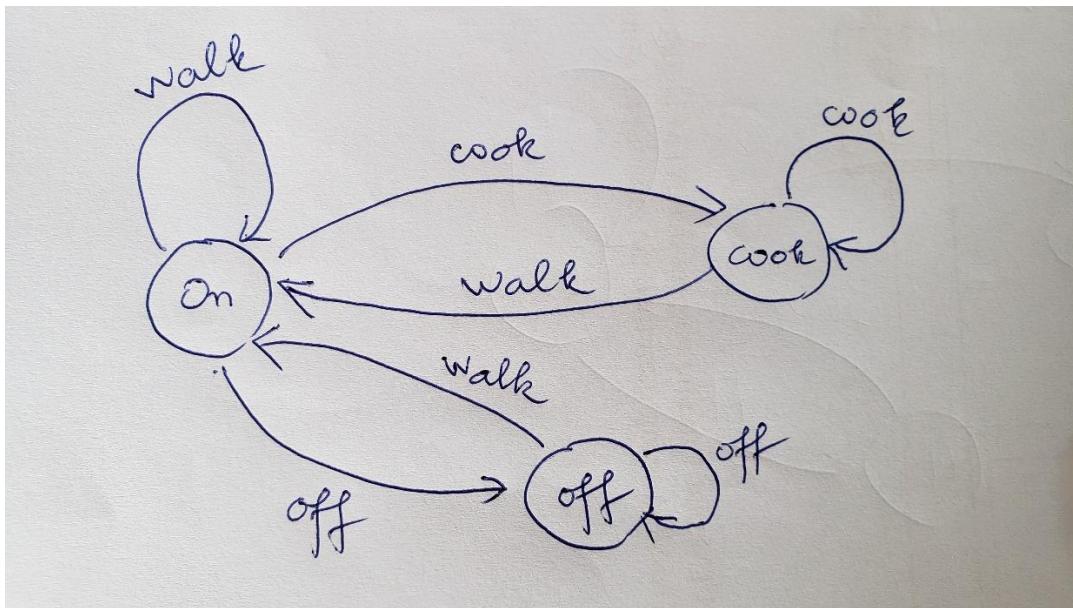
```
package com.javacodegeeks.patterns.statepattern;

public interface RoboticState {

    public void walk();
    public void cook();
    public void off();

}
```

Lớp Robot là lớp cụ thể cài đặt giao diện *RoboticsState*. Lớp này chứa tập các trạng thái mà robot có thể ở trong đó.



```
package com.javacodegeeks.patterns.statepattern;

public class Robot implements RoboticState{

    private RoboticState roboticOn;
    private RoboticState roboticCook;
    private RoboticState roboticOff;

    private RoboticState state;

    public Robot(){
        this.roboticOn = new RoboticOn(this);
        this.roboticCook = new RoboticCook(this);
        this.roboticOff = new RoboticOff(this);

        this.state = roboticOn;
    }

    public void setRoboticState(RoboticState state){
        this.state = state;
    }

    @Override
    public void walk() {
        state.walk();
    }

    @Override
    public void cook() {
        state.cook();
    }

    @Override
    public void off() {
        state.off();
    }

    public RoboticState getRoboticOn() {
        return roboticOn;
    }

    public void setRoboticOn(RoboticState roboticOn) {
        this.roboticOn = roboticOn;
    }

    public RoboticState getRoboticCook() {
        return roboticCook;
    }

    public void setRoboticCook(RoboticState roboticCook) {
        this.roboticCook = roboticCook;
    }

    public RoboticState getRoboticOff() {
        return roboticOff;
    }

    public void setRoboticOff(RoboticState roboticOff) {
```

```
        this.roboticOff = roboticOff;
    }

    public RoboticState getState() {
        return state;
    }

    public void setState(RoboticState state) {
        this.state = state;
    }

}
```

Lớp này khởi tạo mọi trạng thái và đặt trạng thái hiện tại là on.

Bây giờ ta sẽ thấy mọi trạng thái cụ thể của Robot. Một robot sẽ ở một trong số các trạng thái ở thời điểm bất kỳ.

```
package com.javacodegeeks.patterns.statepattern;

public class RoboticOn implements RoboticState{

    private final Robot robot;

    public RoboticOn(Robot robot){
        this.robot = robot;
    }

    @Override
    public void walk() {
        System.out.println("Walking...");
    }

    @Override
    public void cook() {
        System.out.println("Cooking...");
        robot.setRoboticState(robot.getRoboticCook());
    }

    @Override
    public void off() {
        robot.setState(robot.getRoboticOff());
        System.out.println("Robot is switched off");
    }

}
```

```
package com.javacodegeeks.patterns.statepattern;

public class RoboticCook implements RoboticState{

    private final Robot robot;

    public RoboticCook(Robot robot){
        this.robot = robot;
    }

    @Override
    public void walk() {
        System.out.println("Walking...");
        robot.setRoboticState(robot.getRoboticOn());
    }

    @Override
    public void cook() {
        System.out.println("Cooking...");
    }

    @Override
    public void off() {
        System.out.println("Cannot switched off while cooking...");
    }
}

package com.javacodegeeks.patterns.statepattern;

public class RoboticOff implements RoboticState{

    private final Robot robot;

    public RoboticOff(Robot robot){
        this.robot = robot;
    }

    @Override
    public void walk() {
        System.out.println("Walking...");
        robot.setRoboticState(robot.getRoboticOn());
    }

    @Override
    public void cook() {
        System.out.println("Cannot cook at off state.");
    }

    @Override
    public void off() {
        System.out.println("Already switched off...");
    }
}
```

Bây giờ ta kiểm tra code trên

```
package com.javacodegeeks.patterns.statepattern;

public class TestStatePattern {

    public static void main(String[] args) {
        Robot robot = new Robot();
        robot.walk();
        robot.cook();
        robot.walk();
        robot.off();

        robot.walk();
        robot.off();
        robot.cook();

    }
}
```

Code trên cho đầu ra như sau:

```
Walking...
Cooking...
Walking...
Robot is switched off
Walking...
Robot is switched off
Cannot cook at Off state.
```

Trong ví dụ trên, chúng ta đã nhìn thấy là bằng việc đóng gói các trạng thái của một đối tượng vào các lớp khác nhau làm cho code dễ quản trị và linh hoạt hơn.

Bất cứ thay đổi trong một trạng thái sẽ chỉ tác động đến một lớp cụ thể và chúng ta có thể bổ sung trạng thái mới mà không thay đổi nhiều trong code hiện tại. Giả sử chúng ta thêm trạng thái stand-by. Sau khi walk hoặc cook robot sẽ chuyển sang chế độ stand-by để tiết kiệm năng lượng, và chúng ta có thể ra lệnh walk, cook hoặc tắt từ chế độ stand-by.

Để cài đặt thêm điều này, chúng ta cần đưa ra lớp trạng thái mới và bổ sung trạng thái này vào lớp robot. Sau đây là các thay đổi.

```

package com.javacodegeeks.patterns.statepattern;

public class Robot implements RoboticState{

    private RoboticState roboticOn;
    private RoboticState roboticCook;
    private RoboticState roboticOff;
    private RoboticState roboticStandby;

    private RoboticState state;

    public Robot(){
        this.roboticOn = new RoboticOn(this);
        this.roboticCook = new RoboticCook(this);
        this.roboticOff = new RoboticOff(this);
        this.roboticStandby = new RoboticStandby(this);

        this.state = roboticOn;
    }

    public void setRoboticState(RoboticState state){
        this.state = state;
    }

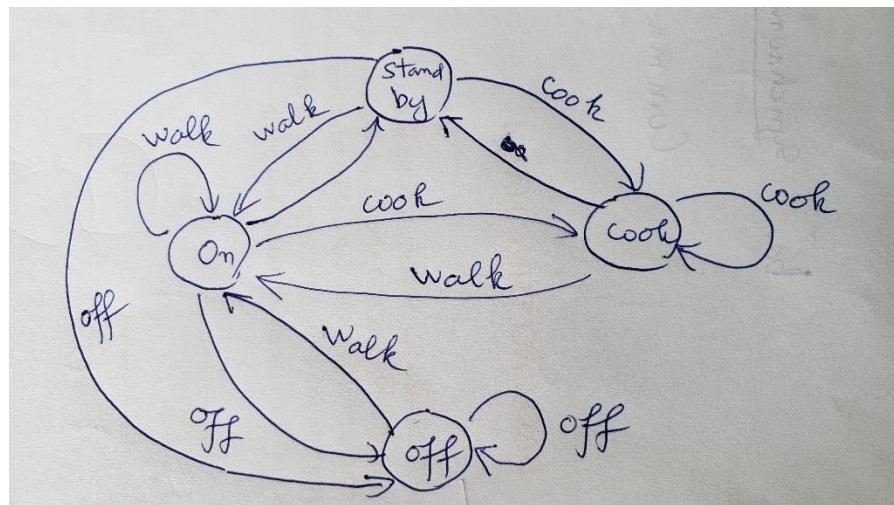
    @Override
    public void walk() {
        state.walk();
        setState(getRoboticStandby());
    }

    @Override
    public void cook() {
        state.cook();
        setState(getRoboticStandby());
    }

    @Override
    public void off() {
        state.off();
    }

    public RoboticState getRoboticOn() {
        return roboticOn;
    }
}

```



```
public void setRoboticOn(RoboticState roboticOn) {
    this.roboticOn = roboticOn;
}

public RoboticState getRoboticCook() {
    return roboticCook;
}

public void setRoboticCook(RoboticState roboticCook) {
    this.roboticCook = roboticCook;
}

public RoboticState getRoboticOff() {
    return roboticOff;
}

public void setRoboticOff(RoboticState roboticOff) {
    this.roboticOff = roboticOff;
}

public RoboticState getState() {
    return state;
}

public void setState(RoboticState state) {
    this.state = state;
}

public RoboticState getRoboticStandby() {
    return roboticStandby;
}

public void setRoboticStandby(RoboticState roboticStandby) {
    this.roboticStandby = roboticStandby;
}

}
```

```
package com.javacodegeeks.patterns.statepattern;

public class RoboticStandby implements RoboticState{

private final Robot robot;

    public RoboticStandby(Robot robot) {
        this.robot = robot;
    }

    @Override
    public void walk() {
        System.out.println("In standby state...");
        robot.setState(robot.getRoboticOn());
        System.out.println("Walking...");
    }

    @Override
    public void cook() {
        System.out.println("In standby state...");
        robot.setRoboticState(robot.getRoboticCook());
        System.out.println("Cooking...");
    }
}
```

```
@Override  
public void off() {  
    System.out.println("In standby state...");  
    robot.setState(robot.getRoboticOff());  
    System.out.println("Robot is switched off");  
}  
}
```

Thay đổi code trên cho kết quả đầu ra:

```
Walking...  
In standby state...  
Cooking...  
In standby state...  
Walking...  
In standby state...  
Robot is switched off  
Walking...  
In standby state...  
Robot is switched off  
Cannot cook at Off state.
```

6.18.4 Khi nào sử dụng mẫu thiết kế State

Sử dụng mẫu *State* cho một trong các trường hợp sau:

- Một hành vi của một đối tượng phụ thuộc vào trạng thái của nó, và nó cần phải thay đổi hành vi của nó trong thời gian chạy phụ thuộc vào trạng thái đó.
- Các thao tác có các lệnh điều kiện nhiều phần, lớn mà phụ thuộc vào trạng thái đối tượng. Trạng thái này thường được biểu diễn bởi một hoặc nhiều ràng buộc. Thông thường, một số thao tác sẽ chứa các cấu trúc điều kiện. Mẫu *State* đặt mỗi nhánh của điều kiện thành một lớp riêng. Điều này cho phép bạn xử lý trạng thái đối tượng như một đối tượng ở trong chính nó mà có thể đa dạng phụ thuộc vào các đối tượng khác.

6.19 Mẫu thiết kế **STRATEGY**

6.19.1 Mở đầu

Mẫu thiết kế *Strategy* trông đơn giản nhất trong các mẫu thiết kế, nó cung cấp tính linh hoạt rất lớn cho code của bạn. Mẫu này được sử dụng hầu khắp mọi nơi, ngay cả trong việc kết hợp với các mẫu thiết kế khác. Các mẫu mà chúng ta đã bàn trước đây có quan hệ với mẫu này, hoặc trực tiếp hoặc gián tiếp. Sau này bạn sẽ nhận được ý tưởng về mẫu này quan trọng như thế nào.

Để hiểu mẫu thiết kế *Strategy*, giả sử chúng ta muốn tạo định dạng văn bản cho trình soạn thảo văn bản. Mỗi người cần phải quan tâm đến trình soạn thảo văn bản. Một trình soạn thảo văn bản có các định dạng văn bản khác nhau để định dạng văn bản. Chúng ta có thể tạo các định dạng văn bản khác nhau và sau đó truyền cái được yêu cầu cho trình soạn thảo văn bản, như vậy trình soạn thảo có khả năng định dạng văn bản như đã yêu cầu.

Trình soạn thảo văn bản (*text editor*) sẽ giữ tham chiếu đến giao diện chung của định dạng văn bản (*text formatter*) và công việc của trình soạn thảo là sẽ truyền *text* cho *formatter* để định dạng *text*.

Giả sử chúng ta cài đặt điều này sử dụng mẫu thiết kế *Strategy* mà sẽ là cho code linh hoạt và dễ bảo trì. Nhưng trước đó, chúng ta sẽ tìm hiểu thêm về mẫu thiết kế *Strategy*.

6.19.2 Mẫu thiết kế *Strategy* là gì

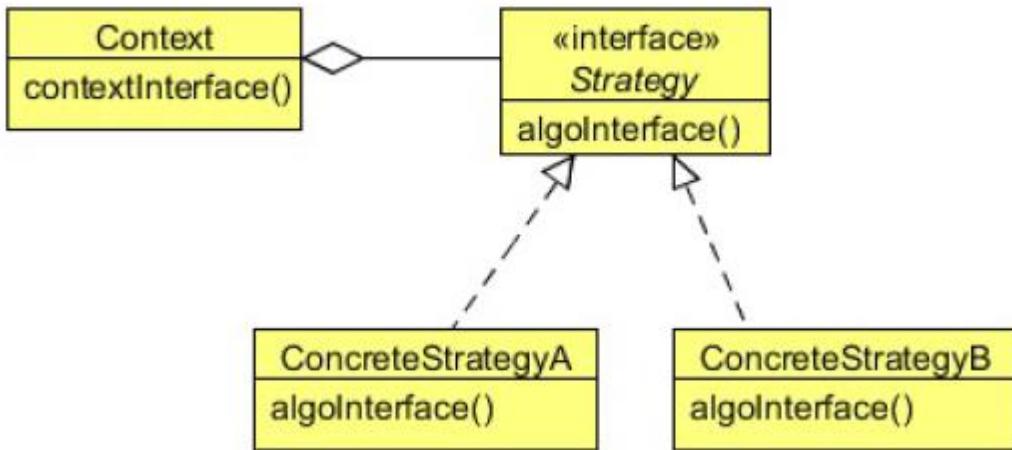
Mẫu thiết kế *Strategy* định nghĩa một họ các thuật toán, đóng gói mỗi một trong chúng và làm cho chúng có thể trao đổi nhau được. Chiến lược *Strategy* cho phép thuật toán đa dạng độc lập khỏi *client* mà sử dụng nó.

Mẫu *Strategy* là hữu ích khi có một tập các thuật toán liên quan và đối tượng *client* cần có khả năng lựa chọn động một thuật toán từ tập đó mà phù hợp với yêu cầu hiện tại. Mẫu *Strategy* đề xuất giữ cài đặt của mỗi thuật toán trong một lớp riêng. Mỗi thuật toán này đóng gói trong một lớp riêng được tham chiếu như một chiến lược *Strategy*. Một đối tượng mà sử dụng đối tượng *Strategy* thường được tham chiếu như *context object*.

Với các đối tượng *Strategy* khác nhau, thay đổi hành vi cho đối tượng *Context* đơn giản là thay đổi hành vi đối tượng *Strategy* của nó đến cái mà cài đặt thuật toán mong muốn. Để cho phép đối tượng *Context* truy cập đến các đối tượng *Strategy* khác nhau theo cách tron tru, mọi đối tượng *Strategy* cần phải được thiết kế để có cùng một giao diện. Trong lập trình Java, điều này được thực hiện bằng cách thiết kế mỗi đối tượng *Strategy* hoặc như cài đặt của một giao diện chung hoặc như lớp con của lớp trừu tượng chung mà khai báo giao diện chung được yêu cầu.

Một lớp các thuật toán liên quan được đóng gói trong tập các lớp *Strategy* trong một phân cấp lớp, mỗi *client* có thể chọn trong số các thuật toán đó bằng cách lấy và khởi tạo một lớp *Strategy* phù hợp. Để thay đổi hành vi của *Context*, đối tượng *client* cần cấu hình *context* với khởi tạo

được chọn. Kiểu bố trí này hoàn toàn tách cài đặt thuật toán ra khỏi *context* mà sử dụng nó. Kết quả là khi cài đặt thuật toán đang có được thay đổi hoặc thuật toán mới được bổ sung vào nhóm, cả hai *context* và đối tượng *client* mà sử dụng *context* đều không bị ảnh hưởng.



Strategy

- Khai báo giao diện chung cho mọi thuật toán hỗ trợ. *Context* sử dụng giao diện này để gọi thuật toán được định nghĩa bởi *ConcreteStrategy*.

ConcreteStrategy

- Cài đặt thuật toán sử dụng giao diện *Strategy*.

Context

- Được cấu hình với đối tượng *ConcreteStrategy*.
- Bảo trì tham chiếu đến đối tượng *Strategy*.
- Có thể định nghĩa một giao diện mà cho phép *Strategy* truy cập đến dữ liệu của nó.

6.19.3 Cài đặt mẫu thiết kế Strategy

Dưới đây là giao diện định dạng văn bản *text formatter* mà được cài đặt bởi mọi *concrete formatter*.

```
package com.javacodegeeks.patterns.strategypattern;

public interface TextFormatter {
    public void format(String text);
}
```

Giao diện trên đây chứa chỉ một phương thức *format*, được dùng để định dạng văn bản.

```
package com.javacodegeeks.patterns.strategypattern;

public class CapTextFormatter implements TextFormatter{

    @Override
    public void format(String text) {
        System.out.println("[CapTextFormatter]: "+text.toUpperCase());
    }

}
```

Lớp trên đây *CapTextFormatter*, là một *concrete formatter* mà cài đặt giao diện *TextFormatter* và lớp này được sử dụng để thay đổi văn bản thành chữ in hoa.

```
package com.javacodegeeks.patterns.strategypattern;

public class LowerTextFormatter implements TextFormatter{

    @Override
    public void format(String text) {
        System.out.println("[LowerTextFormatter]: "+text.toLowerCase());
    }

}
```

LowerTextFormatter, là một *concrete formatter* mà cài đặt giao diện *TextFormatter* và lớp này được sử dụng để thay đổi văn bản thành chữ viết nhỏ.

```
package com.javacodegeeks.patterns.strategypattern;

public class TextEditor {

    private final TextFormatter textFormatter;

    public TextEditor(TextFormatter textFormatter) {
        this.textFormatter = textFormatter;
    }

    public void publishText(String text){
        textFormatter.format(text);
    }

}
```

Lớp trên là lớp *TextEditor*, mà giữ tham chiếu đến giao diện *TextFormatter*. Lớp này chứa phương thức *publishText* mà chuyển tiếp *text* đến cho *Formatter* để in văn bản theo định dạng mong muốn.

Bây giờ, chúng ta sẽ kiểm tra code trên.

```
package com.javacodegeeks.patterns.strategypattern;

public class TestStrategyPattern {

    public static void main(String[] args) {
        TextFormatter formatter = new CapTextFormatter();
        TextEditor editor = new TextEditor(formatter);
        editor.publishText("Testing text in caps formatter");

        formatter = new LowerTextFormatter();
        editor = new TextEditor(formatter);
        editor.publishText("Testing text in lower formatter");
    }
}
```

Code trên sẽ in ra kết quả sau:

```
[CapTextFormatter]: TESTING TEXT IN CAPS FORMATTER
[LowerTextFormatter]: testing text in lower formatter
```

Trong lớp trên, trước hết chúng ta tạo *CapTextFormatter* và gán nó cho khởi tạo *TextEditor*. Sau đó chúng ta gọi phương thức *publishText* để truyền *text* đầu vào cho nó.

Một lần nữa, chúng ta lại làm lại như vậy, nhưng lần này đối tượng *LowerTextFormatter* sẽ được truyền cho *TextEditor*.

Đầu ra rõ ràng chỉ ra rằng, định dạng văn bản khác nhau đã được tạo ra bởi các *text editor* khác nhau vì nó sử dụng các *text formatter* khác nhau.

Ưu điểm chính của mẫu thiết kế *Strategy* là chúng ta nâng cao code mà không phải lo lắng gì nhiều. Chúng ta có thể bổ sung *text Formatter* mới mà không ảnh hưởng đến code hiện thời. Điều này sẽ làm cho code của chúng ta được bảo trì và linh hoạt. Mẫu thiết kế này cũng cung cấp nguyên lý thiết kế “mã cho giao diện” - “code to interface”.

6.19.4 Khi nào sử dụng mẫu thiết kế *Strategy*

Sử dụng mẫu *Strategy* khi

- Nhiều lớp liên quan khác nhau chỉ ở hành vi của chúng. Các chiến lược *Strategies* cung cấp cách cấu hình một lớp với một trong nhiều hành vi.
- Bạn cần các phương án khác nhau của một thuật toán. Chẳng hạn, bạn cần định nghĩa thuật toán phản ánh các cân bằng không gian – thời gian khác nhau. Các chiến lược *Strategies* có thể được sử dụng khi các phương án được cài đặt như một phân cấp lớp của thuật toán.
- Một thuật toán sử dụng dữ liệu mà *client* không cần biết về nó. Sử dụng mẫu *Strategy* để tránh mở thuật toán phức tạp chuyên biệt cho cấu trúc dữ liệu.
- Một lớp định nghĩa nhiều hành vi, và nó trông như lệnh có đa điều kiện trong thao tác của nó. Thay vì nhiều điều kiện, chuyển các nhánh điều kiện liên quan vào lớp *Strategy* riêng của chúng.

6.20 Mẫu thiết kế COMMAND

6.20.1 Mở đầu

Mẫu thiết kế *Command* là mẫu thiết kế hành vi và trợ giúp phân tách người triệu gọi khỏi người nhận được báo cáo.

Để hiểu mẫu thiết kế *Command*, chúng ta sẽ tạo một ví dụ mà thực hiện các kiểu công việc khác nhau. Một công việc có thể là bắt cứ điều gì trong hệ thống, chẳng hạn, gửi thư điện tử, nhắn tin SMS, ghi lưu logging hay thực hiện các chức năng IO.

Mẫu *Command* sẽ giúp phân tách người triệu gọi và người nhận và trợ giúp để thực hiện bắt cứ kiểu công việc nào mà không cần biết về cài đặt của nó. Giả sử chúng ta làm cho ví dụ này thú vị hơn bằng cách tạo ra các luồng mà sẽ giúp thực hiện các công việc một cách đồng thời. Như các công việc này là độc lập với nhau, vì vậy dãy thực hiện các công việc này thực tế là không quan trọng, Chúng ta sẽ tạo một bể luồng để giới hạn số luồng thực hiện các công việc. Đối tượng *Command* sẽ đóng gói các công việc và sẽ truyền nó cho luồng từ bể này mà sẽ thực thi công việc đó.

Trước khi cài đặt Ví dụ này, chúng ta sẽ tìm hiểu thêm về mẫu thiết kế *Command*.

6.20.2 Mẫu thiết kế Command là gì

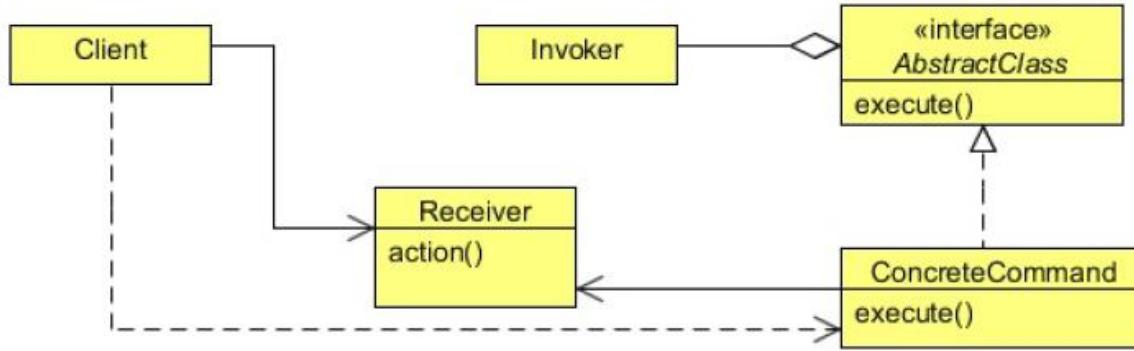
Mục đích của mẫu thiết kế *Command* là đóng gói yêu cầu như một đối tượng, do đó cho phép người phát triển tham số hóa *client* với các yêu cầu khác nhau, hàng đợi hoặc các yêu cầu ghi lại và hỗ trợ các thao tác mà *undo* được.

Nói chung, ứng dụng lập trình hướng đối tượng bao gồm một tập các đối tượng tương tác, mỗi cái đưa ra chức năng chuyên biệt và hạn chế. Để phản hồi tương tác của người sử dụng, ứng dụng tiến hành một kiểu xử lý nào đó. Để làm việc đó, ứng dụng sử dụng dịch vụ của các đối tượng khác nhau để xử lý yêu cầu.

Theo thuật ngữ cài đặt, một ứng dụng có thể phụ thuộc vào đối tượng dành riêng mà triệu gọi các phương thức trên các đối tượng này bằng việc truyền dữ liệu được yêu cầu như các đối số. Đối tượng dành riêng này có thể được tham chiếu đến như người triệu gọi mà nó triệu gọi các thao tác trên các đối tượng khác nhau. Người triệu gọi *Invoker* có thể được coi như một phần của ứng dụng *client*. Tập các đối tượng mà thực tế chứa cài đặt để đưa ra các dịch vụ đáp ứng cho việc xử lý yêu cầu được tham chiếu như các đối tượng nhận *Receiver*.

Sử dụng mẫu *Command*, *Invoker* mà đưa ra các yêu cầu thay mặt *client* và tập các đối tượng *Receiver* cho thuê dịch vụ cần là tách bạch nhau. Mẫu *Command* để xuất tạo một trùu tượng cho việc xử lý được tiến hành hoặc hành động được thực hiện để đáp ứng yêu cầu của *client*. Trùu tượng này cần được thiết kế để khai báo giao diện chung được cài đặt bởi các *concrete implementer* khác nhau được tham chiếu như các đối tượng *Command*. Mỗi đối tượng *Command* thể hiện một kiểu yêu cầu của *client* và một xử lý tương ứng.

Một đối tượng *Command* chỉ chịu trách nhiệm đưa ra một chức năng đáp ứng để xử lý một yêu cầu mà nó thể hiện, nhưng nó không chứa cài đặt thực tế của chức năng đó. Các đối tượng *Command* sử dụng các đối tượng *Receiver* để xuất chức năng đó.



Command

- Khai báo giao diện cho việc thực hiện một thao tác.

ConcreteCommand

- Định nghĩa gắn kết giữa một đối tượng *Receiver* và một hành động
- Cài đặt *Execute* bằng việc triệu gọi thao tác tương ứng trên *Receiver*

Client

- Tạo một đối tượng *ConcreteCommand* và thiết lập *Receiver* của nó.

Invoker

- Yêu cầu một *command* để xử lý một yêu cầu.

Receiver

- Biết thực hiện thao tác liên kết nào để xử lý được yêu cầu đó. Lớp bất kỳ có thể được dùng như lớp *Receiver*.

6.20.3 Cài đặt mẫu Command

Chúng ta sẽ cài đặt ví dụ sử dụng đối tượng *Command*. Một đối tượng *Command* sẽ được tham chiếu bằng giao diện chung và sẽ chứa một phương thức mà sẽ được sử dụng để thực hiện yêu cầu. Các lớp *command* cụ thể sẽ ghi đè phương thức này và sẽ cung cấp cài đặt chuyên biệt của riêng họ để thực hiện yêu cầu đó.

```
package com.javacodegeeks.patterns.commandpattern;

public interface Job {
    public void run();
}
```

Giao diện Job là giao diện *Command*, chứa một phương thức đơn giản là *run*, mà được thực hiện bởi một luồng. Phương thức *execute* của *command* của chúng ta là phương thức *run* mà sẽ được sử dụng để thực thi bởi một luồng để hoàn thành công việc.

Ở đây cần có một kiểu job khác mà cần được thực hiện. Sau đây là các lớp cụ thể khác nhau mà khởi tạo của chúng sẽ được thực thi bởi các đối tượng *command* khác nhau.

```
package com.javacodegeeks.patterns.commandpattern;

public class Email {
    public void sendEmail() {
        System.out.println("Sending email.....");
    }
}
```

```
package com.javacodegeeks.patterns.commandpattern;

public class FileIO {
    public void execute() {
        System.out.println("Executing File IO operations...");
    }
}
```

```
package com.javacodegeeks.patterns.commandpattern;

public class Logging {
    public void log() {
        System.out.println("Logging...");
    }
}
```

```
package com.javacodegeeks.patterns.commandpattern;

public class Sms {
    public void sendSms() {
        System.out.println("Sending SMS...");
    }
}
```

Sau đây là các lớp *Command* khác nhau mà đóng gói các lớp trên và cài đặt giao diện *Job*.

```
package com.javacodegeeks.patterns.commandpattern;

public class EmailJob implements Job{

    private Email email;

    public void setEmail(Email email) {
        this.email = email;
    }

    @Override
    public void run() {
        System.out.println("Job ID: "+Thread.currentThread().getId()+" executing ←
                           email jobs.");
        if(email!=null){
            email.sendEmail();
        }

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

package com.javacodegeeks.patterns.commandpattern;

public class FileIOJob implements Job{

    private FileIO fileIO;

    public void setFileIO(FileIO fileIO) {
        this.fileIO = fileIO;
    }

    @Override
    public void run() {
        System.out.println("Job ID: "+Thread.currentThread().getId()+" executing ←
                           fileIO jobs.");
        if(fileIO!=null){
            fileIO.execute();
        }

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

```
package com.javacodegeeks.patterns.commandpattern;

public class LoggingJob implements Job{

    private Logging logging;

    public void setLogging(Logging logging) {
        this.logging = logging;
    }

    @Override
    public void run() {
        System.out.println("Job ID: "+Thread.currentThread().getId()+" executing ←
                           logging jobs.");
        if(logging!=null){
            logging.log();
        }

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

}
```

```
package com.javacodegeeks.patterns.commandpattern;
```

```
public class SmsJob implements Job{

    private Sms sms;

    public void setSms(Sms sms) {
        this.sms = sms;
    }

    @Override
    public void run() {
        System.out.println("Job ID: "+Thread.currentThread().getId()+" executing ←
                           sms jobs.");
        if(sms!=null){
            sms.sendSms();
        }

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

}
```

Các lớp trên giữ tham chiếu đến các lớp tương ứng của chúng, mà sẽ được sử dụng để hoàn thành công việc. Các lớp ghi đè phương thức *run* và thực hiện công việc được yêu cầu. Chẳng hạn, lớp *SmsJob* được sử dụng để gửi *sms*, phương thức *run* của nó gọi phương thức *sendSms* của đối tượng *Sms* để hoàn thành công việc.

Bạn có thể đặt các đối tượng khác nhau cái nọ cạnh cái kia như cùng là đối tượng *Command*.

Dưới đây là lớp *ThreadPool* được sử dụng để tạo bể các luồng và cho phép một luồng được đẩy vào và thực thi công việc từ hàng đợi công việc.

```
package com.javacodegeeks.patterns.commandpattern;

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public class ThreadPool {

    private final BlockingQueue<Job> jobQueue;
    private final Thread[] jobThreads;
    private volatile boolean shutdown;

    public ThreadPool(int n)
    {
        jobQueue = new LinkedBlockingQueue<>();
        jobThreads = new Thread[n];

        for (int i = 0; i < n; i++) {
            jobThreads[i] = new Worker("Pool Thread " + i);
            jobThreads[i].start();
        }
    }

    public void addJob(Job r)
    {
        try {
            jobQueue.put(r);
        }
```

```

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    public void shutdownPool()
    {
        while (!jobQueue.isEmpty()) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        shutdown = true;
        for (Thread workerThread : jobThreads) {
            workerThread.interrupt();
        }
    }

    private class Worker extends Thread
    {
        public Worker(String name)
        {
            super(name);
        }

        public void run()
        {
            while (!shutdown) {
                try {
                    Job r = jobQueue.take();
                    r.run();
                } catch (InterruptedException e) {
                }
            }
        }
    }
}

```

Lớp trên được sử dụng để tạo n luồng (các luồng làm việc). Mỗi *worker thread* sẽ chờ một công việc trong hàng đợi và sau đó thực thi công việc và sẽ trở lại trạng thái chờ đợi. Lớp này chứa hàng đợi công việc; khi một công việc mới sẽ được bổ sung vào hàng đợi, một *worker* từ bể sẽ thực thi công việc này.

Chúng ta sẽ bổ sung phương thức *shutdownPool* mà sẽ được sử dụng để tắt *Pool* bằng cách ngắt mọi luồng làm việc chỉ khi hàng đợi công việc là rỗng. Phương thức *addJob* được sử dụng để bổ sung công việc vào hàng đợi.

Bây giờ chúng ta kiểm tra code.

```
package com.javacodegeeks.patterns.commandpattern;

public class TestCommandPattern {
    public static void main(String[] args)
    {
        init();
    }

    private static void init()
    {
        ThreadPool pool = new ThreadPool(10);

        Email email = null;
        EmailJob emailJob = new EmailJob();

        Sms sms = null;
        SmsJob smsJob = new SmsJob();

        FileIO fileIO = null;;
        FileIOJob fileIOJob = new FileIOJob();

        Logging logging = null;
        LoggingJob logJob = new LoggingJob();

        for (int i = 0; i < 5; i++) {
            email = new Email();
            emailJob.setEmail(email);

            sms = new Sms();
            smsJob.setSms(sms);

            fileIO = new FileIO();
            fileIOJob.setFileIO(fileIO);

            logging = new Logging();
            logJob.setLogging(logging);

            pool.addJob(emailJob);
            pool.addJob(smsJob);
            pool.addJob(fileIOJob);
            pool.addJob(logJob);
        }
        pool.shutdownPool();
    }
}
```

Code trên cho kết quả đầu ra sau:

```
Job ID: 9 executing email jobs.  
Sending email.....  
Job ID: 12 executing logging jobs.  
Job ID: 17 executing email jobs.  
Sending email.....  
Job ID: 13 executing email jobs.  
Sending email.....  
Job ID: 10 executing sms jobs.  
Sending SMS...  
Job ID: 11 executing fileIO jobs.  
Executing File IO operations...  
Job ID: 18 executing sms jobs.  
Sending SMS...  
Logging...  
Job ID: 16 executing logging jobs.  
Logging...  
Job ID: 15 executing fileIO jobs.  
Executing File IO operations...  
Job ID: 14 executing sms jobs.  
Sending SMS...  
Job ID: 12 executing fileIO jobs.  
Executing File IO operations...  
Job ID: 10 executing logging jobs.  
Logging...  
Job ID: 18 executing email jobs.
```

```
Sending email.....  
Job ID: 16 executing sms jobs.  
Sending SMS...  
Job ID: 14 executing fileIO jobs.  
Executing File IO operations...  
Job ID: 9 executing logging jobs.  
Logging...  
Job ID: 17 executing email jobs.  
Sending email.....  
Job ID: 13 executing sms jobs.  
Sending SMS...  
Job ID: 15 executing fileIO jobs.  
Executing File IO operations...  
Job ID: 11 executing logging jobs.  
Logging...
```

Lưu ý rằng đây ra có thể khác nhau do thứ tự thực hiện các luồng.

Trong lớp trên, chúng ta đã tạo một bể luồng gồm 10 luồng. Sau đó, chúng ta thiết lập các đối tượng *command* khác nhau với các công việc khác nhau và bổ sung các công việc này vào hàng đợi sử dụng phương thức *addJob* của lớp *ThreadPool*. Như chúng ta đã thấy, một công việc được chèn vào hàng đợi, một luồng thực thi công việc đó và xóa nó khỏi hàng đợi.

Chúng ta đã đặt kiểu khác nhau của công việc, nhưng sử dụng mẫu thiết kế *Command*, chúng ta tách bạch công việc khỏi luồng triệu gọi. Một luồng sẽ thực thi bất cứ kiểu đối tượng nào mà cài đặt giao diện *Job*. Các đối tượng *command* khác nhau đóng gói đối tượng khác nhau và thực thi thao tác được yêu cầu trên các đối tượng đó.

Đầu ra chỉ ra rằng các luồng khác nhau thực thi công việc khác nhau. Bằng cách quan sát id công việc ở đầu ra, bạn có thể thấy rõ là mỗi luồng đơn thực hiện nhiều hơn một công việc. Điều này xảy ra vì sau khi thực hiện xong một công việc luồng được gửi trở lại bể.

Ưu điểm của mẫu thiết kế *Command* là bạn có thể bổ sung nhiều kiểu công việc khác mà không thay đổi các lớp đã tồn tại. Nó dẫn đến tính linh hoạt và dễ bảo trì hơn và cũng giảm cơ hội lỗi trong code.

6.20.4 Khi nào sử dụng mẫu thiết kế Command

Sử dụng mẫu *Command* khi bạn muốn:

- Tham số hóa các đối tượng bởi hành động thực hiện.
- Đặc tả, hàng đợi, và thực thi các yêu cầu tại các thời điểm khác nhau. Đối tượng *Command* có thể có thời gian sống độc lập với yêu cầu gốc. Nếu *receiver* của yêu cầu có thể được thể hiện theo cách độc lập với không gian địa chỉ, thì bạn có thể truyền đối tượng *command* cho yêu cầu đến tiến trình khác nhau và thực hiện yêu cầu ở đó.
- Hỗ trợ *undo*. Thao tác *Execute* của *Command* có thể được lưu giữ trạng thái cho việc đảo lại các tác động trong bản thân *command*. Giao diện *Command* cần được bổ sung thao tác *Un-execute* mà đảo lại các tác động của lời gọi *Execute* trước. Các *command* đã thực hiện là được lưu giữ trong danh sách lịch sử. *Undo* mức không hạn chế và *redo* là đạt được bởi viếng thăm danh sách đó ngược chiều hay xuôi chiều khi gọi *Un-execute* hay *Execute* tương ứng.
- Hỗ trợ ghi lại các thay đổi sao cho chúng có thể được áp dụng lại trong trường hợp hệ thống bị sập. Bằng cách làm gia tăng giao diện *Command* với việc tải và lưu giữ các thao tác, bạn có thể giữ log thay đổi một cách bền vững. Khôi phục từ lỗi sập bao gồm tải lại các *logged command* từ đĩa và *re-executing* chúng với thao tác *Execute*.
- Cấu trúc một hệ thống quanh các thao tác bậc cao được xây dựng trên các thao tác nguyên thủy. Một cấu trúc như vậy là phổ cập trong các hệ thống thông tin mà hỗ trợ các giao dịch. Một giao dịch đóng gói một tập các thay đổi đến dữ liệu. Mẫu *Command* đưa ra cách để mô hình các giao dịch. *Commands* có giao diện chung, cho phép bạn triệu gọi mọi giao dịch theo cách như nhau. Mẫu này cũng làm cho nó dễ mở rộng hệ thống với các giao dịch mới.

6.21 Mẫu thiết kế INTERPRETER

6.21.1 Mở đầu

Mẫu thiết kế *Interpreter* là mẫu có trọng trách cao (heavy-duty). Nó gộp tất cả những gì đi kèm với ngôn ngữ lập trình của riêng bạn lại hoặc kiểm soát cái đã có bằng cách tạo bộ thông dịch *interpreter* cho ngôn ngữ đó. Để sử dụng mẫu này, bạn cần biết một chút về văn phạm hình thức đi kèm với một ngôn ngữ. Như bạn có thể hình dung, nó là một trong số các mẫu mà người phát triển thực tế không sử dụng hàng ngày, vì việc tạo một ngôn ngữ của riêng bạn không phải là việc nhiều người làm.

Chẳng hạn, định nghĩa biểu thức trong ngôn ngữ mới của bạn có thể trông giống như đoạn sau trong ngôn ngữ hình thức:

```
expression ::=<command> | <repetition> | <sequence>
```

Mỗi biểu thức trong ngôn ngữ của bạn, khi đó, có thể được tạo nên từ các biểu thức lệnh command, phép lặp repetition hoặc các biểu thức tuần tự. Mỗi mục có thể được thể hiện như một đối tượng với phương thức *interpret* để dịch ngôn ngữ mới của bạn vào một cái gì đó bạn có thể chạy trên Java.

Để minh họa sử dụng mẫu thiết kế *Interpreter* giả sử chúng ta tạo một ví dụ để tính một biểu thức toán đơn giản, nhưng trước đó, chúng ta sẽ tìm hiểu thêm về mẫu *Interpreter* trong mục sau.

6.21.2 Mẫu thiết kế Interpreter là gì

Cho một ngôn ngữ, định nghĩa một thể hiện cho một văn phạm của nó với một bộ thông dịch mà sử dụng thể hiện đó để dịch các câu trong ngôn ngữ đó.

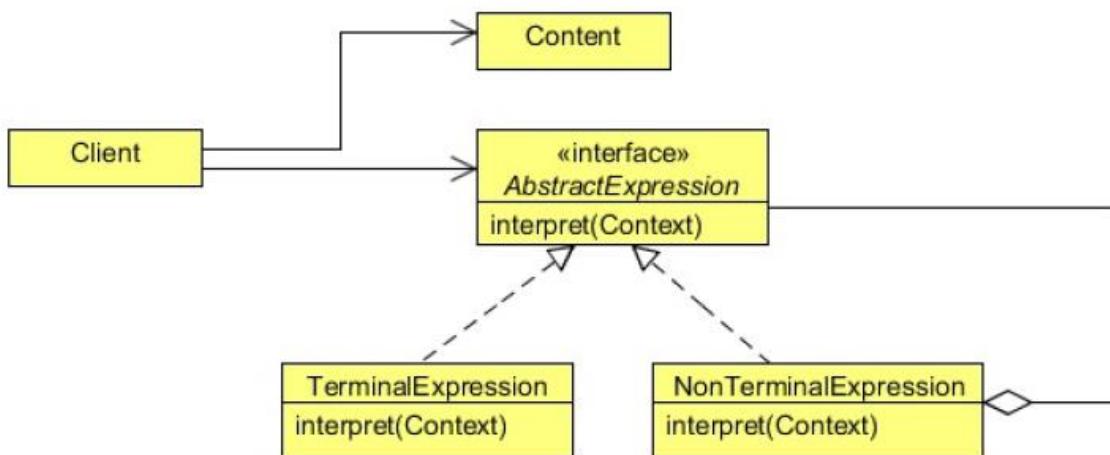
Nó chung, ngôn ngữ là được sinh ra từ tập các qui tắc văn phạm. Các câu khác nhau có thể được tạo nên bởi việc tuân theo các qui tắc văn phạm này. Đôi khi một ứng dụng có thể cần để xử lý các xuất hiện lặp của các yêu cầu tương tự mà là sự kết hợp của tập các qui tắc văn phạm. Các yêu cầu này là khác nhau nhưng là tương tự theo nghĩa chúng được tạo nên từ cùng một tập các qui tắc.

Một ví dụ đơn giản của điều trên có thể là tập các biểu thức số học khác nhau được đưa vào chương trình tính toán. Mặc dù mỗi biểu thức như vậy là khác nhau, chúng đều được tạo nên sử dụng các qui tắc cơ bản mà hợp lại là bộ văn phạm của ngôn ngữ các biểu thức số học.

Trong các trường hợp như vậy, thay vì xử lý mỗi sự kết hợp khác nhau của các qui tắc như một trường hợp riêng biệt, ứng dụng sẽ có ích hơn nếu có khả năng thông dịch sự kết hợp tổng quan của các qui tắc. Mẫu *Interpreter* có thể được sử dụng để thiết kế khả năng như vậy trong một ứng dụng sao cho các ứng dụng và người sử dụng khác có thể đặc tả các thao tác sử dụng ngôn ngữ đơn giản được định nghĩa bởi tập các qui tắc văn phạm.

Một phân cấp lớp có thể được thiết kế để biểu diễn tập các qui tắc văn phạm với mỗi lớp trong phân cấp đó biểu diễn một qui tắc văn phạm riêng biệt. Một module *Interpreter* có thể được thiết kế để thông dịch các câu được tạo nên sử dụng phân cấp lớp được thiết kế bên trên và tiến hành các thao tác cần thiết.

Bởi vì lớp khác nhau biểu diễn một qui tắc văn phạm, nên số các lớp tăng lên cùng số qui tắc văn phạm. Một ngôn ngữ với các qui tắc văn phạm phức tạp và mở rộng đòi hỏi số lớp rất lớn. Mẫu *Interpreter* làm việc tốt nhất khi văn phạm là đơn giản. Có văn phạm đơn giản sẽ tránh sự cần thiết phải có nhiều lớp tương ứng với một tập phức tạp các qui tắc kéo theo mà là khó quản trị và bảo trì.



AbstractExpression

- Khai báo một thao tác *Interpret* trừu tượng mà là chung cho mọi đỉnh trên cây cú pháp trừu tượng.

TerminalExpression

- Cài đặt một thao tác *Interpret* tương ứng với ký hiệu tận cùng trong văn phạm.
- Một khởi tạo được đòi hỏi cho mỗi ký hiệu tận cùng trong một câu.

NonterminalExpression

- Một lớp như vậy được yêu cầu cho mỗi qui tắc $R ::= R_1 R_2 \dots R_n$ trong văn phạm.
- Bảo trì các biến khởi tạo của kiểu *AbstractExpression* cho mỗi ký hiệu R_1 cho đến R_n .
- Cài đặt một thao tác *Interpret* cho mỗi ký hiệu chưa tận trong văn phạm. *Interpret* thông thường gọi đệ qui đến chính nó trên các biến biểu diễn R_1 đến R_n .

Context

- Chứa thông tin mà là tổng thể cho bộ thông dịch *Interpreter*.

Client

- Xây dựng một cây cú pháp trừu tượng (hoặc là cho trước) biểu diễn một câu cụ thể trong ngôn ngữ mà văn phạm định nghĩa. Cây cú pháp trừu tượng được lắp ghép từ các khởi tạo của các lớp *NonterminalExpression* và *TerminalExpression*.
- Triệu gọi thao tác *Interpret*.

6.21.3 Cài đặt thiết kế mẫu Interpreter

```
package com.javacodegeeks.patterns.interpreterpattern;

public interface Expression {
    public int interpret();
}
```

Giao diện trên được sử dụng bởi mọi biểu thức cụ thể khác nhau và ghi đè phương thức *Interpret* để định nghĩa thao tác chuyên biệt của chúng trên biểu thức đó.

Sau đây là các lớp biểu thức chuyên biệt của thao tác.

```
package com.javacodegeeks.patterns.interpreterpattern;

public class Add implements Expression{

    private final Expression leftExpression;
    private final Expression rightExpression;

    public Add(Expression leftExpression, Expression rightExpression ) {
        this.leftExpression = leftExpression;
        this.rightExpression = rightExpression;
    }
    @Override
    public int interpret() {
        return leftExpression.interpret() + rightExpression.interpret();
    }
}
```

```
package com.javacodegeeks.patterns.interpreterpattern;

public class Product implements Expression{

    private final Expression leftExpression;
    private final Expression rightExpression;

    public Product(Expression leftExpression, Expression rightExpression ) {
        this.leftExpression = leftExpression;
        this.rightExpression = rightExpression;
    }
    @Override
    public int interpret() {
        return leftExpression.interpret() * rightExpression.interpret();
    }
}
```

```
package com.javacodegeeks.patterns.interpreterpattern;

public class Subtract implements Expression{

    private final Expression leftExpression;
    private final Expression rightExpression;

    public Subtract(Expression leftExpression, Expression rightExpression ) {
        this.leftExpression = leftExpression;
        this.rightExpression = rightExpression;
    }
    @Override
    public int interpret() {

        return leftExpression.interpret() - rightExpression.interpret();
    }
}
```

```
package com.javacodegeeks.patterns.interpreterpattern;

public class Number implements Expression{

    private final int n;

    public Number(int n){
        this.n = n;
    }
    @Override
    public int interpret() {
        return n;
    }
}
```

Dưới đây là lớp tiện ích tùy chọn mà chứa các phương thức tiện ích khác nhau được dùng để tính toán biểu thức.

```
package com.javacodegeeks.patterns.interpreterpattern;

public class ExpressionUtils {

    public static boolean isOperator(String s) {
        if (s.equals("+") || s.equals("-") || s.equals("*"))
            return true;
        else
            return false;
    }

    public static Expression getOperator(String s, Expression left, Expression right) {
        switch (s) {
        case "+":
            return new Add(left, right);
        case "-":
            return new Subtract(left, right);
        case "*":
            return new Product(left, right);
        }
        return null;
    }

}
```

Bây giờ chúng ta kiểm tra ví dụ trên.

```
package com.javacodegeeks.patterns.interpreterpattern;

import java.util.Stack;

public class TestInterpreterPattern {

    public static void main(String[] args) {

        String tokenString = "7 3 - 2 1 + *";
        Stack<Expression> stack = new Stack<>();
        String[] tokenArray = tokenString.split(" ");
        for (String s : tokenArray) {

            if (ExpressionUtils.isOperator(s)) {
                Expression rightExpression = stack.pop();
                Expression leftExpression = stack.pop();
                Expression operator = ExpressionUtils.getOperator(s, ←
                    leftExpression, rightExpression);
                int result = operator.interpret();
                stack.push(new Number(result));
            } else {
                Expression i = new Number(Integer.parseInt(s));
                stack.push(i);
            }
        }
        System.out.println("(" +tokenString+ "): "+stack.pop().interpret());
    }
}
```

Code trên sẽ cho ra kết quả sau.

```
( 7 3 -2 1 + * ):12
```

Lưu ý rằng chúng ta đã sử dụng biểu thức hậu tố để tính toán.

Nếu bạn không biết về hậu tố, dưới đây là tóm tắt thông tin về nó. Có ba cách ký hiệu cho biểu thức toán học: trung tố, hậu tố và tiền tố.

- Ký hiệu trung tố là ký hiệu số học và logic chung, mà ở đó phép toán được viết ở giữa các toán hạng mà nó thao tác như $3 + 4$.
- Hậu tố hay ký hiệu Balan viết ngược là cách ký hiệu toán học mà ở đó mỗi phép toán được viết sau các toán hạng của nó như $3 4 +$.
- Tiền tố hay ký hiệu Balan là ký hiệu toán học mà ở đó mỗi phép toán được viết trước các toán hạng của nó như $+ 3 4$.

Ký hiệu trung tố thường được sử dụng trong biểu thức toán học. Hai ký hiệu khác được sử dụng như cú pháp cho biểu thức toán học cho bởi Interpreter cho các ngôn ngữ lập trình.

Trong lớp trên, chúng ta khai báo hậu tố của một biểu thức trong biến *tokenString*. Sau đó ta tách *tokenString* và gán nó cho một mảng, *tokenArray*. Khi lặp các token từng cái một, trước hết chúng ta kiểm tra xem *token* là phép toán hay toán hạng. Nếu *token* là toán hạng, ta truyền nó cho ngăn xếp, nhưng nếu nó là phép toán ta kéo hai toán hạng trên cùng ra khỏi ngăn xếp. Phương thức *getOperation* từ *ExpressionUtils* trả về lớp biểu thức tương ứng tương ứng với phép toán được truyền cho nó.

Sau đó, chúng ta thông dịch nhận kết quả và đẩy lại ngăn xếp. Sau khi lặp xong *tokenList* chúng ta nhận được kết quả cuối cùng.

6.21.4 Khi nào sử dụng mẫu thiết kế Interpreter

Sử dụng mẫu *Interpreter* khi có một ngôn ngữ để thông dịch, và bạn có thể biểu diễn các câu trong ngôn ngữ đó như các cây cú pháp trừu tượng. Mẫu *Interpreter* làm việc tốt nhất khi

- Văn phạm là đơn giản. Đối với văn phạm phức tạp, phân cấp lớp cho văn phạm trở nên lớn và khó quản trị. Các công cụ như bộ sinh bộ phân tích là lựa chọn tốt hơn trong trường hợp này. Chúng có thể thông dịch các biểu thức mà không cần xây dựng cây cú pháp trừu tượng mà có thể tiết kiệm không gian và thời gian.
- Tính hiệu quả không là vấn đề quan trọng ở đây. Hầu hết các bộ thông dịch hiệu quả thường không được cài đặt bởi thông dịch cây cú pháp trực tiếp, nhưng trước hết chúng sang dạng khác. Chẳng hạn, các biểu thức chính qui thường được biến đổi vào máy trạng thái. Nhưng ngay cả khi đó, bộ biến đổi có thể được cài đặt bởi mẫu *Interpreter*, như vậy mẫu này vẫn được áp dụng.

6.22 Mẫu thiết kế DECORATOR

6.22.1 Mở đầu

Để hiểu được mẫu thiết kế *Decorator* (người trang trí), giả sử giúp công ty Pizza thực hiện tinh toán thêm cho bề mặt. Một người sử dụng có thể yêu cầu bổ sung một lớp bề mặt thêm trên pizza và công việc của chúng ta là bổ sung bề mặt và tăng giá của nó khi sử dụng hệ thống.

Đây là một cái gì đó giống như bổ sung thêm trách nhiệm cho pizza của chúng ta trong thời gian thực và mẫu thiết kế *Decorator* là phù hợp với kiểu yêu cầu này. Nhưng trước hết chúng ta sẽ tìm hiểu thêm về mẫu hành vi này.

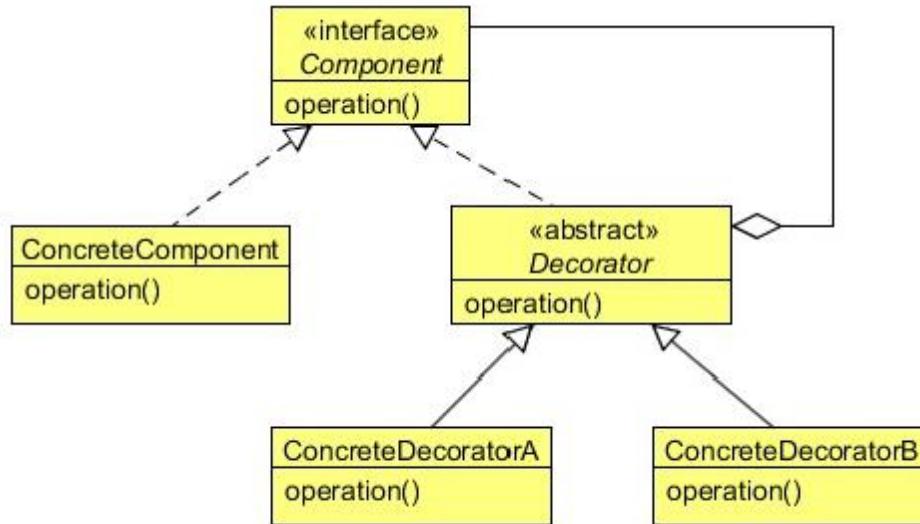
6.22.2 Mẫu thiết kế *Decorator* là gì

Mục đích của mẫu thiết kế *Decorator* là dính trách nhiệm thêm vào một đối tượng một cách động. *Decorator* cung cấp một lựa chọn linh hoạt cho lớp con để mở rộng chức năng.

Mẫu *Decorator* được sử dụng để mở rộng chức năng của đối tượng một cách động mà không thay đổi lớp gốc hoặc sử dụng giao diện. Điều này được thực hiện bằng cách tạo một đối tượng bao bọc được tham chiếu như *Decorator* bao quanh đối tượng thực tế. .

Một đối tượng *Decorator* được thiết kế để có cùng một giao diện như đối tượng bên dưới. Nó cho phép đối tượng client tương tác với đối tượng *Decorator* theo cách chính xác như nhau như có thể với chính đối tượng bên dưới. Đối tượng *Decorator* chứa một tham chiếu đến đối tượng thực tế. Đối tượng *Decorator* nhận một yêu cầu từ client. Đến lượt nó chuyển tiếp các lời gọi này đến đối tượng bên dưới. Đối tượng *Decorator* bổ sung một số chức năng trước hoặc sau khi chuyển tiếp một số yêu cầu cho đối tượng bên dưới. Điều này đảm bảo rằng chức năng bổ sung có thể được thêm vào bên ngoài cho đối tượng cho trước trong thời gian chạy mà không thay đổi cấu trúc của nó.

Decorator ngăn cản sự gia tăng của các lớp con dẫn đến độ phức tạp và hỗn độn ít hơn. Để dàng bổ sung bất cứ tổ hợp khả năng nào. Một khả năng như vậy có thể bổ sung hai lần. Nó trở nên có thể có các đối tượng *decorator* khác nhau đồng thời cho cùng một đối tượng. Một client có thể chọn các khả năng nào mà muốn bằng cách gửi thông điệp đến *decorator* phù hợp.



Component

- Định nghĩa giao diện cho các đối tượng mà có trách nhiệm bổ sung vào chúng một cách động.

ConcreteComponent

- Định nghĩa một đối tượng mà gắn vào nó các trách nhiệm bổ sung có thể đính kèm

Decorator

- Duy trì một tham chiếu đến một đối tượng `Component` và định nghĩa giao diện hợp với giao diện của `Component`.

ConcreteDecorator

- Bổ sung trách nhiệm cho `Component`.

6.22.3 Cài đặt mẫu thiết kế Decorator

Để đơn giản chúng ta tạo giao diện Pizza mà chỉ có hai phương thức.

```
package com.javacodegeeks.patterns.decoratorpattern;

public interface Pizza {
    public String getDesc();
    public double getPrice();
}
```

Phương thức `getDesc` được sử dụng để nhận được mô tả pizza, trong khi phương thức `getPrice` được dùng để nhận giá.

Dưới đây là hai lớp Pizza cụ thể.

```
package com.javacodegeeks.patterns.decoratorpattern;

public class SimplyVegPizza implements Pizza{

    @Override
    public String getDesc() {
        return "SimplyVegPizza (230)";
    }

    @Override
    public double getPrice() {
        return 230;
    }

}

package com.javacodegeeks.patterns.decoratorpattern;

public class SimplyNonVegPizza implements Pizza{

    @Override
    public String getDesc() {
        return "SimplyNonVegPizza (350)";
    }

    @Override
    public double getPrice() {
        return 350;
    }

}
```

Decorator đóng gói đối tượng mà chức năng cần thiết cần được tăng cường cho nó, như vậy cần cài đặt cùng giao diện. Dưới đây là *abstract decorator* mà sẽ được mở rộng bởi mọi *decorator* cụ thể.

```
package com.javacodegeeks.patterns.decoratorpattern;

public abstract class PizzaDecorator implements Pizza {

    @Override
    public String getDesc() {
        return "Toppings";
    }

}
```

Sau đây là các lớp *Decorator* cụ thể:

```
package com.javacodegeeks.patterns.decoratorpattern;

public class Broccoli extends PizzaDecorator{

    private final Pizza pizza;

    public Broccoli(Pizza pizza) {
        this.pizza = pizza;
    }

    @Override
    public String getDesc() {
        return pizza.getDesc()+" , Broccoli (9.25)";
    }
}
```

```
    @Override
    public double getPrice() {
        return pizza.getPrice()+9.25;
    }

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class Cheese extends PizzaDecorator{

    private final Pizza pizza;

    public Cheese(Pizza pizza) {
        this.pizza = pizza;
    }

    @Override
    public String getDesc() {
        return pizza.getDesc()+" , Cheese (20.72)";
    }

    @Override
    public double getPrice() {
        return pizza.getPrice()+20.72;
    }

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class Chicken extends PizzaDecorator{

    private final Pizza pizza;

    public Chicken(Pizza pizza) {
        this.pizza = pizza;
    }

    @Override
    public String getDesc() {
        return pizza.getDesc()+" Chicken (12.75)";
    }

    @Override
    public double getPrice() {
        return pizza.getPrice()+12.75;
    }
}

package com.javacodegeeks.patterns.decoratorpattern;

public class FetaCheese extends PizzaDecorator{

    private final Pizza pizza;

    public FetaCheese(Pizza pizza) {
        this.pizza = pizza;
    }

    @Override
    public String getDesc() {
        return pizza.getDesc()+" Feta Cheese (25.88)";
    }

    @Override
    public double getPrice() {
        return pizza.getPrice()+25.88;
    }
}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class GreenOlives extends PizzaDecorator{

    private final Pizza pizza;

    public GreenOlives(Pizza pizza) {
        this.pizza = pizza;
    }

    @Override
    public String getDesc() {
        return pizza.getDesc()+" , Green Olives (5.47)";
    }

    @Override
    public double getPrice() {
        return pizza.getPrice()+5.47;
    }

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class Ham extends PizzaDecorator{

    private final Pizza pizza;

    public Ham(Pizza pizza) {
        this.pizza = pizza;
    }

    @Override
    public String getDesc() {
        return pizza.getDesc()+" , Ham (18.12)";
    }

    @Override
    public double getPrice() {
        return pizza.getPrice()+18.12;
    }

}
```

```
package com.javacodegeeks.patterns.decoratorpattern;

public class Meat extends PizzaDecorator{

    private final Pizza pizza;

    public Meat(Pizza pizza){
        this.pizza = pizza;
    }

    @Override
    public String getDesc() {
        return pizza.getDesc()+" , Meat (14.25)";
    }

    @Override
    public double getPrice() {
        return pizza.getPrice()+14.25;
    }

}

package com.javacodegeeks.patterns.decoratorpattern;

public class RedOnions extends PizzaDecorator{

    private final Pizza pizza;

    public RedOnions(Pizza pizza){
        this.pizza = pizza;
    }

    @Override
    public String getDesc() {
        return pizza.getDesc()+" , Red Onions (3.75)";
    }

    @Override
    public double getPrice() {
        return pizza.getPrice()+3.75;
    }

}

package com.javacodegeeks.patterns.decoratorpattern;

public class RomaTomatoes extends PizzaDecorator{

    private final Pizza pizza;

    public RomaTomatoes(Pizza pizza){
        this.pizza = pizza;
    }

    @Override
    public String getDesc() {
        return pizza.getDesc()+" , Roma Tomatoes (5.20)";
    }

}
```

```
@Override  
public double getPrice() {  
    return pizza.getPrice() + 5.20;  
}  
  
}  
  
package com.javacodegeeks.patterns.decoratorpattern;  
  
public class Spinach extends PizzaDecorator{  
  
    private final Pizza pizza;  
  
    public Spinach(Pizza pizza) {  
        this.pizza = pizza;  
    }  
  
    @Override  
    public String getDesc() {  
        return pizza.getDesc() + ", Spinach (7.92)";  
    }  
  
    @Override  
    public double getPrice() {  
        return pizza.getPrice() + 7.92;  
    }  
}
```

Chúng ta cần trang trí đối tượng pizza của chúng ta với các lớp bề mặt trên. Các lớp trên chưa tham chiếu đến đối tượng pizza mà cần được trang trí.. Đối tượng decorator bổ sung chức năng của nó cho decorator sau khi gọi hàm của decorator.

```
package com.javacodegeeks.patterns.decoratorpattern;

import java.text.DecimalFormat;

public class TestDecoratorPattern {

    public static void main(String[] args) {

        DecimalFormat dformat = new DecimalFormat("#.##");
        Pizza pizza = new SimplyVegPizza();

        pizza = new RomaTomatoes(pizza);
        pizza = new GreenOlives(pizza);
        pizza = new Spinach(pizza);

        System.out.println("Desc: "+pizza.getDesc());
        System.out.println("Price: "+dformat.format(pizza.getPrice()));

        pizza = new SimplyNonVegPizza();

        pizza = new Meat(pizza);
        pizza = new Cheese(pizza);
        pizza = new Cheese(pizza);
        pizza = new Ham(pizza);

        System.out.println("Desc: "+pizza.getDesc());
        System.out.println("Price: "+dformat.format(pizza.getPrice()));
    }
}
```

Code trên cho kết quả đầu ra như sau:

```
Desc: SimplyVegPizza (230), Roma Tomatoes (5.20), Green Olives (5.47), Spinach (7.92)
Price: 248.59
Desc: SimplyNonVegPizza (350), Meat (14.25), Cheese (20.72), Cheese (20.72), Ham (18.12)
Price: 423.81
```

Trong lớp trên, trước hết chúng ta tạo đối tượng *SimplyVegPizza* và sau đó trang trí nó với *RomaTomatoes*, *GreenOlives* và *Spinach*. Mô tả desc trong đầu ra chỉ rõ các lớp bù mặt này được bổ sung cho *SimplyVegPizza* và giá của nó là tổng tất cả.

Chúng ta làm như vậy cho *SimplyNonVegPizza* và bổ sung lớp khác lên nó. Lưu ý rằng bạn có thể trang trí cùng một vật nhiều hơn một lần cho một đối tượng. Trong ví dụ trên, chúng ta đã bổ sung *cheese* hai lần, nó cũng cộng hai lần vào giá, mà có thể thấy ở đầu ra.

Mẫu thiết kế *Decorator* trông có vẻ tốt khi bạn cần bổ sung thêm chức năng cho đối tượng với việc thay đổi nó trong thời gian chạy. Nhưng nó cho kết quả trong nhiều đối tượng nhỏ. Một thiết kế mà sử dụng *Decorator* thường cho kết quả trong hệ thống tích hợp từ nhiều đối tượng nhỏ mà trông giống như vậy. Các đối tượng khác nhau chỉ ở cách chúng được kết nối với nhau, không ở trong lớp của chúng hoặc ở giá trị của các biến của chúng. Mặc dù các hệ thống này là dễ tùy biến bởi những người hiểu chúng, nhưng chúng có thể vẫn là khó để học và bắt lỗi.

6.22.4 Khi nào sử dụng mẫu thiết kế Decorator

Sử dụng mẫu *Decorator* trong các trường hợp sau:

- Bổ sung trách nhiệm cho các đối tượng riêng lẻ một cách động và trong suốt, mà là không làm ảnh hưởng đến các đối tượng khác.
- Đối với các trách nhiệm mà có thể gỡ bỏ.
- Khi mở rộng bởi các lớp con là không thể. Đôi khi số lớn các mở rộng độc lập là có thể và có thể tạo ra sự bùng nổ các lớp con để hỗ trợ mỗi tổ hợp. Hoặc định nghĩa một lớp có thể được che giấu hoặc ngược lại là không sẵn sàng cho tách các lớp con.

6.23 Mẫu thiết kế ITERATOR

6.23.1 Mở đầu

Các đối tượng hợp lại như danh sách, cần cho bạn cách truy cập đến các phần tử của nó mà không cần mở cấu trúc bên trong. Hơn nữa, bạn có thể muốn duyệt danh sách theo các cách khác nhau, phụ thuộc dựa trên cái gì bạn muốn thực hiện. Nhưng bạn có thể không muốn làm phỏng lên giao diện *List* với thao tác cho các bộ duyệt khác nhau, ngay cả nếu bạn có thể thấy trước cái bạn cần. Bạn có thể cũng có nhiều hơn một bộ duyệt chưa quyết định trên cùng một danh sách.

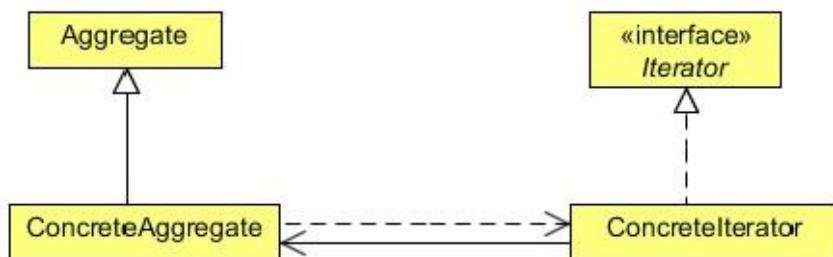
Mẫu Iterator cho phép bạn làm mọi việc đó. Ý tưởng chính trong mẫu này là nhận trách nhiệm để truy cập và duyệt một đối tượng như danh sách *list* và đặt chúng vào trong một đối tượng *Iterator*. Lớp *Iterator* định nghĩa một giao diện để truy cập các phần tử của danh sách. Một đối tượng *Iterator* là có trách nhiệm giữ theo dõi phần tử hiện thời; như vậy, nó biết các phần tử nào đã được viếng thăm.

6.23.2 Mẫu Iterator là gì

Mục đích của mẫu thiết kế *Iterator* là cung cấp cách truy cập các phần tử của một đối tượng hợp lại một cách tuần tự mà không mở biểu diễn bên trong của nó.

Mẫu *Iterator* cho phép đối tượng client truy cập đến nội dung của vật chứa theo cách tuần tự, mà không biết gì về thể hiện bên trong của nội dung của nó. Thuật ngữ vật chứa *container*, được sử dụng ở trên, có thể đơn giản được định nghĩa như một họ các dữ liệu hoặc các đối tượng. Các đối tượng bên trong vật chứa đến lượt nó có thể là họ các họ.

Mẫu *Iterator* cho phép đối tượng client duyệt qua họ này của các đối tượng (hay vật chứa) mà không mở vật chứa xem dữ liệu bên trong được lưu như thế nào. Để làm được việc đó, mẫu *Iterator* đề xuất đối tượng *Container* cần được thiết kế để cung cấp giao diện *public* ở dạng một đối tượng *Iterator* cho các đối tượng client khác nhau truy cập đến nội dung của nó. Một đối tượng *Iterator* chứa các phương thức *public* cho phép một đối tượng client định hướng qua danh sách các đối tượng bên trong *Container*.



Iterator

- Định nghĩa giao diện cho việc truy vấn thăm viếng các phần tử.

ConcreteIterator

- Cài đặt giao diện *Iterator*.
- Theo dõi vị trí hiện thời trong viếng thăm hợp phần tử

Aggregate

- Định nghĩa giao diện tạo đối tượng *Iterator*

ConcreteAggregate

- Cài đặt giao diện tạo Iterator để trả về khởi tạo của *ConcreteIterator*

6.23.3 Cài đặt mẫu thiết kế Iterator

Chúng ta sẽ cài đặt mẫu thiết kế *Iterator* sử dụng lớp *Shape*. Chúng ta sẽ lưu giữ và duyệt các đối tượng *Shape* sử dụng *Iterator*.

```
package com.javacodegeeks.patterns.iteratorpattern;

public class Shape {

    private int id;
    private String name;

    public Shape(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString(){
        return "ID: "+id+" Shape: "+name;
    }
}
```

Lớp *Shape* đơn giản có *Id* và *Name* như các thuộc tính của nó

```
package com.javacodegeeks.patterns.iteratorpattern;

public class ShapeStorage {

    private Shape []shapes = new Shape[5];

    private int index;

    public void addShape(String name) {
        int i = index++;
        shapes[i] = new Shape(i, name);
    }

    public Shape[] getShapes() {
        return shapes;
    }
}
```

Lớp trên được sử dụng để lưu giữ các đối tượng *Shape*. Lớp này chứa mảng kiểu *Shape*, để đơn giản chúng ta khởi tạo mảng gồm 5 phần tử. Phương thức *addShape* được sử dụng để bổ sung đối tượng *Shape* vào mảng và tăng chỉ số lên một. Phương thức *getShapes* trả về mảng kiểu *Shape*.

```
package com.javacodegeeks.patterns.iteratorpattern;

import java.util.Iterator;

public class ShapeIterator implements Iterator<Shape>{

    private Shape [] shapes;
    int pos;

    public ShapeIterator(Shape []shapes) {
        this.shapes = shapes;
    }

    @Override
    public boolean hasNext() {
        if(pos >= shapes.length || shapes[pos] == null)
            return false;
        return true;
    }

    @Override
    public Shape next() {
        return shapes[pos++];
    }

    @Override
    public void remove() {
        if(pos <=0 )
            throw new IllegalStateException("Illegal position");
        if(shapes[pos-1] !=null){
            for (int i= pos-1; i<(shapes.length-1);i++){
                shapes[i] = shapes[i+1];
            }
            shapes[shapes.length-1] = null;
        }
    }
}
```

Lớp trên là *Iterator* cho lớp *Shape*. Lớp này cài đặt giao diện *Iterator* và định nghĩa mọi phương thức của giao diện *Iterator*.

Phương thức *hasNext* trả về giá trị *Bool*, còn hay không phần tử còn lại. Phương thức *next* trả về phần tử tiếp theo từ trong họ đó và phương thức *remove* xóa bỏ phần tử hiện thời khỏi họ.

```
package com.javacodegeeks.patterns.iteratorpattern;

public class TestIteratorPattern {

    public static void main(String[] args) {
        ShapeStorage storage = new ShapeStorage();
        storage.addShape("Polygon");
        storage.addShape("Hexagon");
        storage.addShape("Circle");
        storage.addShape("Rectangle");
        storage.addShape("Square");

        ShapeIterator iterator = new ShapeIterator(storage.getShapes());
        while(iterator.hasNext()) {
            System.out.println(iterator.next());
        }
        System.out.println("Apply removing while iterating...");
        iterator = new ShapeIterator(storage.getShapes());
        while(iterator.hasNext()) {
            System.out.println(iterator.next());
            iterator.remove();
        }
    }
}
```

Code trên sẽ cho kết quả đầu ra như sau:

```
ID: 0 Shape: Polygon
ID: 1 Shape: Hexagon
ID: 2 Shape: Circle
ID: 3 Shape: Rectangle
ID: 4 Shape: Square
Apply removing while iterating...
ID: 0 Shape: Polygon
ID: 2 Shape: Circle
ID: 4 Shape: Square
```

Trong lớp trên, chúng ta đã tạo đối tượng *ShapeStorage* và lưu giữ các đối tượng *Shapes* ở đó. Tiếp theo, chúng ta tạo đối tượng *ShapeIterator* và gán nó cho các *shape*. Chúng ta lặp hai lần, lần đầu không gọi phương thức *remove* và sau đó có gọi *remove*.

Đầu ra chỉ ra cho bạn tác động của *remove*. Tại vòng lặp đầu tiên, *iterator* in ra mọi *shape* nhưng khi phương thức *remove* được gọi, trước hết nó in ra *shape* hiện thời và sau đó *remove* *shape* tiếp theo. Sau đó, chúng ta gọi phương thức *remove* trên nó mà loại bỏ phần tử hiện thời và tiếp theo *iterator* trả đến đối tượng *shape* tiếp theo đang sẵn sàng.

Đó là tại sao, đầu ra thay đổi thành “Áp dụng removing trong khi iterating...” chỉ đưa ra các đối tượng 0, 2 và 4.

6.23.4 Iterator bên trong và bên ngoài

Một *Iterator* có thể được thiết kế là *iterator* bên trong hoặc *iterator* bên ngoài.

1. Iterator bên trong

- Một họ bản thân nó để xuất các phương thức mà cho phép client viếng thăm các đối tượng khác nhau bên trong họ đó. Chẳng hạn, lớp *Java.util.ResultSet* chứa dữ liệu và cũng để xuất các phương thức như *next* để điều hướng đọc theo danh sách các phần tử.
- Có thể chỉ có một *iterator* trên một họ tại mỗi thời điểm.
- Một họ cần phải duy trì hoặc lưu trạng thái của *iterator*.

2. Iterator bên ngoài

- Chức năng lặp bên ngoài là tách bạch khỏi họ các phần tử và giữ bên trong một đối tượng khác được tham chiếu là *iterator*. Thông thường, bản thân họ trả về đối tượng *iterator* phù hợp cho client phụ thuộc vào đầu vào của client. Chẳng hạn, lớp *Java.util.Vector* có *iterator* được định nghĩa ở dạng một đối tượng riêng kiểu *Enumeration*. Đối tượng này được trả về cho đối tượng client để phản hồi cho lời gọi của phương thức *element ()*.
- Có thể có nhiều bộ duyệt *iterator* trên cùng một họ tại một thời điểm.
- Chi phí cho việc lưu giữ trạng thái của *iterator* là không liên quan đến họ đó. Nó đi cùng đối tượng *Iterator* dành riêng.

6.23.5 Khi nào dùng mẫu thiết kế Iterator

Sử dụng mẫu *Iterator*:

- Để truy cập đến nội dung của đối tượng hợp thành mà không mở biểu diễn bên trong của nó.
- Để hỗ trợ duyệt đa chiều các đối tượng hợp thành.
- Cung cấp giao diện đồng nhất để duyệt các cấu trúc hợp thành khác nhau (như là, hỗ trợ lặp đa hình).

6.24 Mẫu thiết kế VISITOR

6.24.1 Mở đầu

Để hiểu mẫu thiết kế *Visitor*, chúng ta xem lại mẫu thiết kế *Composite*. Mẫu *Composite* cho phép bạn tích hợp các đối tượng vào một cấu trúc cây để thể hiện phân cấp một phần – tổng thể.

Trong ví dụ mẫu *Composite*, chúng ta đã tạo cấu trúc *html* hợp thành từ các kiểu đối tượng khác nhau. Bây giờ giả sử chúng ta cần bổ sung một lớp *css* vào *tag html*. Một cách làm điều này là bằng cách bổ sung một lớp khi thêm *start tag* sử dụng phương thức *setStartTag*. Nhưng đặt code cứng này sẽ tạo nên tính không mềm dẻo trong code của chúng ta.

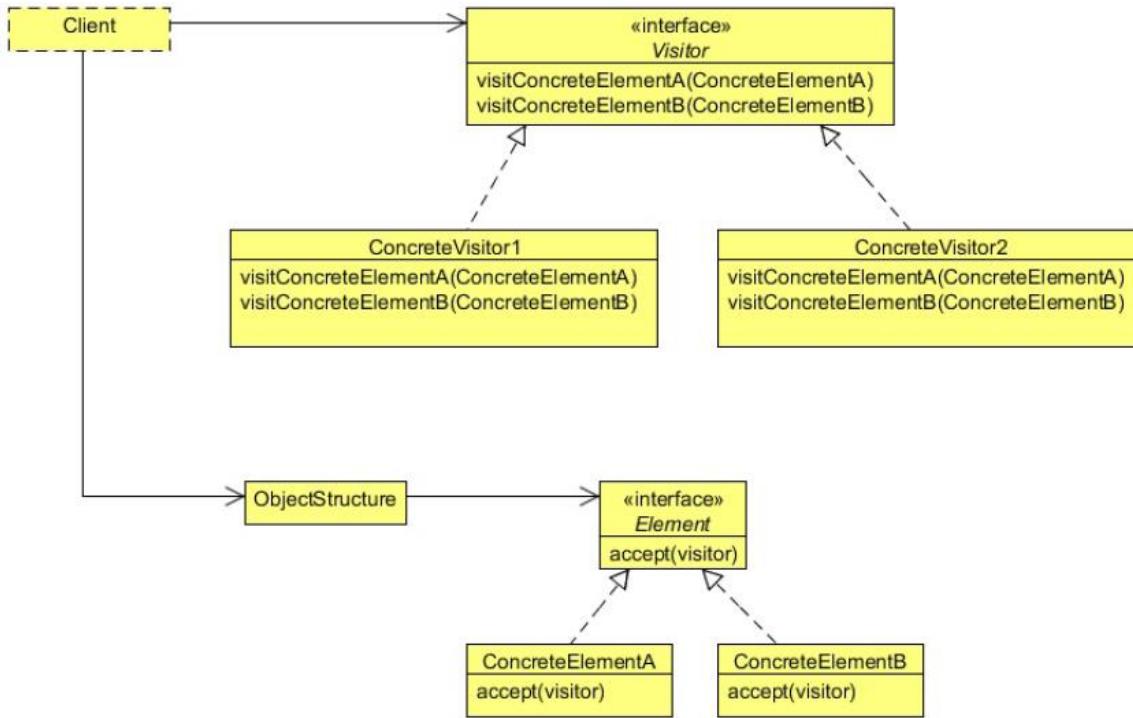
Một cách khác làm điều này là bổ sung phương thức mới kiểu *addClass* trong lớp trừu tượng *HtmlTag*. Mọi con của lớp đó sẽ ghi đè phương thức này và sẽ cung cấp lớp *css*. Một nhược điểm chính của cách tiếp cận này là, nếu có nhiều lớp con (sẽ ở trong trang *html* lớn), nó sẽ trở nên rất đắt và buộc phải cài đặt phương thức này trong mọi lớp con của nó. Và giả sử, sau này chúng ta cần bổ sung phần tử kiểu khác trong *tag*, chúng ta cũng cần phải làm như vậy.

Mẫu *Visitor* cung cấp cho bạn cách bổ sung các thao tác mới trên các đối tượng mà không cần thay đổi các lớp của các phần tử, đặc biệt khi các thao tác thay đổi khá thường xuyên.

6.24.2 Mẫu thiết kế Visitor là gì

Mục đích của mẫu thiết kế *Visitor* là biểu diễn một thao tác được thực hiện trên các phần tử của một cấu trúc đối tượng. *Visitor* cho phép bạn định nghĩa một thao tác mới mà không thay đổi các lớp của các phần tử mà trên đó nó thao tác.

Mẫu *Visitor* là hữu ích khi thiết kế một thao tác đọc theo họ không đồng nhất các đối tượng của một phân cấp lớp. Mẫu *Visitor* cho phép thao tác đó được định nghĩa mà không thay đổi lớp của bất cứ đối tượng nào trong họ đó. Để thực hiện điều đó, mẫu *Visitor* đề xuất định nghĩa thao tác trong một lớp riêng được tham chiếu là lớp *visitor*. Nó tách bạch thao tác đó khỏi họ đối tượng mà nó thao tác trên đó. Với mỗi thao tác được định nghĩa mới, một lớp *visitor* mới được tạo. Vì thao tác này được thực hiện đọc theo tập các đối tượng, *visitor* đó cần có cách truy cập các thành viên *public* của các đối tượng đó. Yêu cầu này có thể được đáp ứng bằng cách cài đặt hai ý tưởng thiết kế sau.



Visitor

- Khai báo một thao tác *Visit* cho mỗi lớp *ConcreteElement* trong cấu trúc đối tượng. Tên và đối số của thao tác này định danh lớp mà gửi yêu cầu *visit* cho *visitor*. Điều này cho phép *visitor* xác định lớp *concrete* của phần tử cần được viếng thăm. Sau đó *visitor* này có thể truy cập trực tiếp đến phần tử thông qua một giao diện cụ thể của nó.

ConcreteVisitor

- Cài đặt mỗi thao tác được định nghĩa trong *Visitor*. Mỗi thao tác cài đặt một đoạn của một thuật toán được định nghĩa cho lớp tương ứng của đối tượng trong cấu trúc đó. *ConcreteVisitor* cung cấp ngữ cảnh cho thuật toán đó và lưu giữ trạng thái cục bộ của nó. Trạng thái này thường tích góp các kết quả đọc theo quá trình viếng thăm cấu trúc đó.

Element

- Định nghĩa thao tác *Accept* mà nhận đối số là *visitor*

ConcreteElement

Cài đặt thao tác *Accept* mà nhận đối số là *visitor*

ObjectStructure

- Có thể liệt kê các phần tử của nó
- Có thể cung cấp giao diện mức độ cao để cho phép *visitor* viếng thăm các phần tử của nó.
- Có thể hoặc là một hợp thành hoặc một họ như một danh sách hoặc một tập hợp.

6.24.2 Mẫu thiết kế Visitor là gì

6.24.3 Cài đặt mẫu thiết kế Visitor

Để cài đặt mẫu thiết kế *Visitor*, chúng ta sử dụng cùng *Composite Pattern code* và sẽ đưa vào một số giao diện, lớp và phương thức mới.

Cài đặt mẫu *Visitor* yêu cầu hai giao diện quan trọng, giao diện *Element* mà chứa một phương thức *accept* với đối số kiểu *Visitor*. Giao diện này sẽ được cài đặt bởi tất cả các lớp mà cần để cho phép các *visitor* viếng thăm chúng. Trong trường hợp của chúng ta, *HtmlTag* sẽ cài đặt giao diện *Element*, như *HtmlTag* là lớp cha trùu tượng của mọi lớp *html* cụ thể, các lớp cụ thể này sẽ kế thừa và sẽ ghi đè phương thức *accept* của giao diện *Element*.

Giao diện khác là giao diện *Visitor*, giao diện này sẽ chứa các phương thức *visit* với các đối số của một lớp mà cài đặt giao diện *Element*. Lưu ý rằng, chúng ta sẽ yêu cầu hai phương thức mới trong lớp *HtmlTag* của chúng ta, *getStartTag* và *getEndTag*, ngược lại với ví dụ chỉ ra trong Bài mẫu thiết kế *Composite*.

```
package com.javacodegeeks.patterns.visitorpattern;

public interface Element {
    public void accept(Visitor visitor);
}

package com.javacodegeeks.patterns.visitorpattern;

public interface Visitor {
    public void visit(HtmlElement element);
    public void visit(HtmlParentElement parentElement);
}
```

Code dưới đây lấy từ Ví dụ Bài mẫu thiết kế *Composite* với một ít thay đổi:

```
package com.javacodegeeks.patterns.visitorpattern;

import java.util.List;

public abstract class HtmlTag implements Element{

    public abstract String getTagName();
    public abstract void setStartTag(String tag);
    public abstract String getStartTag();
    public abstract String getEndTag();
    public abstract void setEndTag(String tag);
    public void setTagBody(String tagBody){
        throw new UnsupportedOperationException("Current operation is not support ←
            for this object");
    }
    public void addChildTag(HtmlTag htmlTag){
        throw new UnsupportedOperationException("Current operation is not support ←
            for this object");
    }
    public void removeChildTag(HtmlTag htmlTag){
        throw new UnsupportedOperationException("Current operation is not support ←
            for this object");
    }
    public List<HtmlTag>getChildren(){
        throw new UnsupportedOperationException("Current operation is not support ←
            for this object");
    }
    public abstract void generateHtml();
}
```

Lớp trùu tượng *HtmlTag* cài đặt giao diện *Element*. Các lớp cụ thể dưới đây sẽ ghi đè phương thức *accept* của giao diện *Element* và sẽ gọi phương thức *visit* và sẽ truyền thao tác này như một đối số. Điều này cho phép phương thức *visitor* nhận mọi thành viên *public* của đối tượng này, bổ sung các thao tác mới trên nó.

```
package com.javacodegeeks.patterns.visitorpattern;

import java.util.ArrayList;
import java.util.List;

public class HtmlParentElement extends HtmlTag {

    private String tagName;
    private String startTag;
    private String endTag;
    private List<HtmlTag> childrenTag;

    public HtmlParentElement(String tagName) {
        this.tagName = tagName;
        this.startTag = "";
        this.endTag = "";
        this.childrenTag = new ArrayList<>();
    }

    @Override
    public String getTagName() {
        return tagName;
    }

    @Override
    public void setStartTag(String tag) {
        this.startTag = tag;
    }

    @Override
    public void setEndTag(String tag) {
        this.endTag = tag;
    }

    @Override
    public String getStartTag() {
        return startTag;
    }

    @Override
    public String getEndTag() {
        return endTag;
    }

    @Override
    public void addChildTag(HtmlTag htmlTag){
        childrenTag.add(htmlTag);
    }

    @Override
    public void removeChildTag(HtmlTag htmlTag){
        childrenTag.remove(htmlTag);
    }

    @Override
    public List<HtmlTag>getChildren(){
        return childrenTag;
    }
}
```

```
@Override
public void generateHtml() {
    System.out.println(startTag);
    for(HtmlTag tag : childrenTag){
        tag.generateHtml();
    }
    System.out.println(endTag);
}

@Override
public void accept(Visitor visitor) {
    visitor.visit(this);
}

}

package com.javacodegeeks.patterns.visitorpattern;

public class HtmlElement extends HtmlTag{

    private String tagName;
    private String startTag;
    private String endTag;
    private String tagBody;

    public HtmlElement(String tagName) {
        this.tagName = tagName;
        this.tagBody = "";
        this.startTag = "";
        this.endTag = "";
    }

    @Override
    public String getTagName() {
        return tagName;
    }

    @Override
    public void setStartTag(String tag) {
        this.startTag = tag;
    }

    @Override
    public void setEndTag(String tag) {
        this.endTag = tag;
    }

    @Override
    public String getStartTag() {
        return startTag;
    }

    @Override
    public String getEndTag() {
        return endTag;
    }

    @Override
    public void setTagBody(String tagBody) {
        this.tagBody = tagBody;
    }
}
```

```
}

@Override
public void generateHtml() {
    System.out.println(startTag+" "+tagBody+" "+endTag);
}

@Override
public void accept(Visitor visitor) {
    visitor.visit(this);
}

}
```

Bây giờ, các lớp *concrete visitor*: chúng ta tạo hai lớp *concrete*, một sẽ bổ sung lớp *visitor* là *css* cho mọi html tag và cái khác là thay đổi độ rộng *tag* sử dụng thuộc tính *style* của *html tag*.

```
package com.javacodegeeks.patterns.visitorpattern;

public class CssClassVisitor implements Visitor{

    @Override
    public void visit(HtmlElement element) {
        element.setStartTag(element.getStartTag().replace(">", " class='visitor'>") );
    }

    @Override
    public void visit(HtmlParentElement parentElement) {
        parentElement.setStartTag(parentElement.getStartTag().replace(">", " class ='visitor'>"));
    }

}
```

```
package com.javacodegeeks.patterns.visitorpattern;

public class StyleVisitor implements Visitor {

    @Override
    public void visit(HtmlElement element) {
        element.setStartTag(element.getStartTag().replace(">", " style='width:46px <br>"));
    }

    @Override
    public void visit(HtmlParentElement parentElement) {
        parentElement.setStartTag(parentElement.getStartTag().replace(">", " style ='width:58px;'>"));
    }

}
```

Bây giờ chúng ta kiểm tra ví dụ trên.

```
package com.javacodegeeks.patterns.visitorpattern;

public class TestVisitorPattern {
```

```
public static void main(String[] args) {  
    System.out.println("Before visitor..... \\n");  
  
    HtmlTag parentTag = new HtmlParentElement("<html>");  
    parentTag.setStartTag("<html>");  
    parentTag.setEndTag("</html>");  
  
    HtmlTag p1 = new HtmlParentElement("<body>");  
    p1.setStartTag("<body>");  
    p1.setEndTag("</body>");  
  
    parentTag.addChildTag(p1);  
  
    HtmlTag child1 = new HtmlElement("<P>");  
    child1.setStartTag("<P>");  
    child1.setEndTag("</P>");  
    child1.setTagBody("Testing html tag library");  
    p1.addChildTag(child1);  
  
    child1 = new HtmlElement("<P>");  
    child1.setStartTag("<P>");  
    child1.setEndTag("</P>");  
    child1.setTagBody("Paragraph 2");  
    p1.addChildTag(child1);  
  
    parentTag.generateHtml();  
  
    System.out.println("\\nAfter visitor..... \\n");  
  
    Visitor cssClass = new CssClassVisitor();  
    Visitor style = new StyleVisitor();  
  
    parentTag = new HtmlParentElement("<html>");  
    parentTag.setStartTag("<html>");  
    parentTag.setEndTag("</html>");  
    parentTag.accept(style);  
    parentTag.accept(cssClass);  
  
    p1 = new HtmlParentElement("<body>");  
    p1.setStartTag("<body>");  
    p1.setEndTag("</body>");  
    p1.accept(style);  
    p1.accept(cssClass);  
  
    parentTag.addChildTag(p1);  
  
    child1 = new HtmlElement("<P>");  
    child1.setStartTag("<P>");  
    child1.setEndTag("</P>");  
    child1.setTagBody("Testing html tag library");  
    child1.accept(style);  
    child1.accept(cssClass);  
  
    p1.addChildTag(child1);  
  
    child1 = new HtmlElement("<P>");  
    child1.setStartTag("<P>");  
    child1.setEndTag("</P>");  
    child1.setTagBody("Paragraph 2");  
    child1.accept(style);  
    child1.accept(cssClass);  
}
```

```
        p1.addChildTag(child1);

        parentTag.generateHtml();
    }

}
```

Code trên sẽ in đầu ra như sau:

```
Before visitor.....
<html>
<body>
<P>Testing html tag library</P>
<P>Paragraph 2</P>
</body>
</html>

After visitor.....
<html style='width:58px;' class='visitor'>
<body style='width:58px;' class='visitor'>
<p style='width:46px;' class='visitor'>Testing html tag library</P>
<p style='width:46px;' class='visitor'>Paragraph 2</P>
</body>
</html>
```

Đầu ra sau “*Before Visitor ...*” là giống với kết quả trong Ví dụ ở Bài mẫu *Composite*. Sau đó, chúng ta tạo hai *concrete visitor* và bổ sung chúng cho các đối tượng *concrete html* sử dụng phương thức *accept*. Đầu ra “*After visitor ...*” chỉ ra cho bạn rằng ở đó lớp *css* và các phần tử *style* đã được bổ sung cho các *html tag*.

Lưu ý rằng, ưu điểm của mẫu *Visitor* là ở chỗ chúng ta có thể thêm các thao tác mới cho các đối tượng mà không thay đổi các lớp này. Chẳng hạn, chúng ta có thể bổ sung một số chức năng của *Javascript* như *onclick* hoặc một số *angular js ng tags* mà không thay đổi các lớp đó.

6.24.4 Khi nào sử dụng mẫu thiết kế *Visitor*

Sử dụng mẫu thiết kế *Visitor* khi:

- Một cấu trúc đối tượng chứa nhiều lớp các đối tượng có sự khác nhau về giao diện và bạn muốn thực hiện các thao tác trên các đối tượng này mà phụ thuộc các lớp cụ thể của chúng.
- Nhiều thao tác không liên quan và khác nhau cần được thực hiện trên các đối tượng trong cấu trúc các đối tượng, và chúng ta muốn tránh làm ảnh hưởng các lớp của họ với các thao tác này. *Visitor* cho phép bạn giữ các thao tác liên quan cùng nhau bởi định nghĩa chúng trong một lớp. Khi cấu trúc các đối tượng được chia sẻ bởi nhiều ứng dụng, sử dụng *Visitor* để đặt các thao tác trong các ứng dụng mà cần đến chúng.
- Các lớp định nghĩa cấu trúc đối tượng rất ít khi thay đổi, nhưng bạn thường xuyên muốn định nghĩa các thao tác mới trên cấu trúc đó. Thay đổi các lớp cấu trúc đối tượng đòi hỏi định nghĩa lại giao diện giao diện cho mọi *visitor*, mà là phải trả giá. Nếu các lớp cấu trúc đối tượng thay đổi thường xuyên, thì có thể tốt hơn là định nghĩa các thao tác trong các lớp đó.

TÀI LIỆU THAM KHẢO

1. Barbara Liskov with John Guttag. *Program Development in Java: Abstraction, Specification and Object-Oriented Design*. Addison Wesley, Pearson Education, 2002.
2. Rohit Joshi. *Java Design Patterns: Reusable solutions to common problems*. Exelixis Media, P.C, 2015.
3. Schneider. S. The B Method: an introduction. Macmillan Education UK, 10/2001.