CHƯƠNG 2

***

NHỮNG CÁI TÊN RÕ NGHĨA

Viết bởi Tim Ottinger

Giới thiệu

Những cái tên có ở khắp mọi nơi trong phần mềm. Chúng ta đặt tên cho các biến, các hàm, các đối số, các lớp và các gói của chúng ta. Chúng ta đặt tên cho những file mã nguồn và thư mục chứa chúng. Chúng ta đặt tên cho những file **.jar*, file **.war,...* Chúng ta đặt tên và đặt tên. Vì chúng ta đặt tên rất nhiều, nên chúng ta cần làm tốt điều đó. Sau đây là một số quy tắc đơn giản để tạo nên những cái tên tốt.

Dùng những tên thể hiện được mục đích

Điều này rất dễ. Nhưng chúng tôi muốn nhấn mạnh rằng chúng tôi nghiêm túc trong việc này. Chọn một cái tên "xịn" mất khá nhiều thời gian, nhưng lại tiết kiệm (thời gian) hơn sau đó. Vì vậy, hãy quan tâm đến cái tên mà bạn chọn và chỉ thay đổi chúng khi bạn sáng tạo ra tên "xịn" hơn. Những người đọc code của bạn (kể cả bạn) sẽ *sung sướng* hơn khi bạn làm điều đó.

Tên của biến, hàm, hoặc lớp phải trả lời tất cả những câu hỏi về nó. Nó phải cho bạn biết lý do nó tồn tại, nó làm được những gì, và dùng nó ra sao. Nếu có một comment đi kèm theo tên, thì tên đó không thể hiện được mục đích của nó.

```
int d;  // elapsed time in days
```

Tên **`d`** không tiết lộ điều gì cả. Nó không gợi lên cảm giác gì về thời gian, cũng không liên quan gì đến ngày. Chúng ta nên chọn một tên thể hiện được những gì đang được cân đo, và cả đơn vị đo của chúng:

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

Việc chọn tên thể hiện được mục đích có thể làm cho việc hiểu và thay đổi code dễ dàng hơn nhiều. Hãy đoán xem mục đích của đoạn code dưới đây là gì?

```
public List<int[]> getThem() {
    List<int[]> list1 = new ArrayList<int[]>();
    for (int[] x : theList)
        if (x[0] == 4)
            list1.add(x);
    return list1;
}
```

Tại sao lại nói khó mà biết được đoạn code này đang làm gì? Không có biểu thức phức tạp, khoảng cách và cách thụt đầu dòng hợp lý, chỉ có 3 biến và 2 hằng số được đề cập. Thậm chí không có các lớp (class) và phương thức đa hình nào, nó chỉ có một danh sách mảng (hoặc thứ gì đó trông giống vậy).

Vấn đề không nằm ở sự đơn giản của code mà nằm ở ý nghĩa của code, do bối cảnh không rõ ràng. Đoạn code trên bắt chúng tôi phải tìm câu trả lời cho các câu hỏi sau:

1. `theList` chứa cái gì?
2. Ý nghĩa của chỉ số 0 trong phần tử của `theList`?
3. Số `4` có ý nghĩa gì?
4. Danh sách được `return` thì dùng kiểu gì?

Câu trả lời không có trong code, nhưng sẽ có ngay sau đây. Giả sử chúng tôi đang làm game *dò mìn*. Chúng tôi thấy rằng giao diện trò chơi là một danh sách các ô vuông (cell) được gọi là `theList`. Vậy nên, hãy đổi tên nó thành `gameBoard`.

Mỗi ô trên màn hình được biểu diễn bằng một sanh sách đơn giản. Chúng tôi cũng thấy rằng chỉ số của số 0 là vị trí biểu diễn giá trị trạng thái (status value), và giá trị 4 nghĩa là trạng thái *được gắn cờ (flagged).* Chỉ bằng cách đưa ra các khái niệm này, chúng tôi có thể cải thiện mã nguồn một cách đáng kể:

```
public List<int[]> getFlaggedCells() {
    List<int[]> flaggedCells = new ArrayList<int[]>();
    for (int[] cell : gameBoard)
        if (cell[STATUS_VALUE] == FLAGGED)
            flaggedCells.add(cell);
    return flaggedCells;
}
```

Cần lưu ý rằng mức độ đơn giản của code vẫn không thay đổi, nó vẫn chính xác về toán tử, hằng số, và các lệnh lồng nhau,…Nhưng đã trở nên rõ ràng hơn rất nhiều.

Chúng ta có thể đi xa hơn bằng cách viết một lớp đơn giản cho các ô thay vì sử dụng các mảng kiểu `int`. Nó có thể bao gồm một hàm thể hiện được mục đích (gọi nó là *isFlagged – được gắn cờ* chẳng hạn) để giấu đi những con số ma thuật *(Từ gốc: magic number – Một khái niệm về các hằng số, tìm hiểu thêm tại https://en.wikipedia.org/wiki/Magic_number_(programming)).*

```
public List<Cell> getFlaggedCells() {
    List<Cell> flaggedCells = new ArrayList<Cell>();
    for (Cell cell : gameBoard)
        if (cell.isFlagged())
            flaggedCells.add(cell);
    return flaggedCells;
}
```

Với những thay đổi đơn giản này, không quá khó để hiểu những gì mà đoạn code đang trình bày. Đây chính là sức mạnh của việc chọn tên tốt.

Tránh sai lệch thông tin

Các lập trình viên phải tránh để lại những dấu hiệu làm code trở nên khó hiểu. Chúng ta nên tránh dùng những từ mang nghĩa khác với nghĩa cố định của nó. Ví dụ, các tên biến như `hp`, `aix` và `sco` là những tên biến vô cùng tồi tệ, chúng là tên của các nền tảng Unix hoặc các biến thể. Ngay cả khi bạn đang code về cạnh huyền (hypotenuse) và `hp` trông giống như một tên viết tắt tốt, rất có thể đó là một cái tên tồi.

Không nên quy kết rằng một nhóm các tài khoản là một `accountList` nếu nó không thật sự là một danh sách (`List`). Từ *danh sách* có nghĩa là một thứ gì đó cụ thể cho các lập trình viên. Nếu các tài khoản không thực sự tạo thành danh sách, nó có thể dẫn đến một kết quả sai lầm. Vậy nên, `accountGroup` hoặc `bunchOfAccounts`, hoặc đơn giản chỉ là `accounts` sẽ tốt hơn.

Cẩn thận với những cái tên gần giống nhau. Mất bao lâu để bạn phân biệt được sự khác nhau giữa `XYZControllerForEfficientHandlingOfStrings` và `XYZControllerForEfficientStorageOfStrings` trong cùng một module, hay đâu đó xa hơn một chút? Những cái tên gần giống nhau như thế này thật sự, thật sự rất khủng khiếp cho lập trình viên.

Một kiểu khủng bố tinh thần khác về những cái tên không rõ ràng là ký tự L viết thường và O viết hoa. Vấn đề? Tất nhiên là nhìn chúng gần như hoàn toàn giống hằng số không và một, kiểu như:

```
int a = l;
if ( O == l ) a = O1;
else l = 01;
```

Bạn nghĩ chúng tôi *xạo*? Chúng tôi đã từng khảo sát, và kiểu code như vậy thực sự rất nhiều. Trong một số trường hợp, tác giả của code đề xuất sử dụng phông chữ khác nhau để tách biệt chúng. Một giải pháp khác có thể được sử dụng là truyền đạt bằng lời nói hoặc để lại tài liệu cho các lập trình viên sau này có thể hiểu nó. Vấn đề được giải quyết mà không cần phải đổi tên để tạo ra một sản phẩm khác.

Tạo nên những sự khác biệt có nghĩa

Các lập trình viên tạo ra vấn đề cho chính họ khi viết code chỉ để đáp ứng cho trình biên dịch hoặc thông dịch. Ví dụ, vì bạn không thể sử dụng cùng một tên để chỉ hai thứ khác nhau trong cùng một khối lệnh hoặc cùng một phạm vi, bạn có thể bị "dụ dỗ" thay đổi tên một cách tùy tiện. Đôi khi điều đó làm bạn cố tình viết sai chính tả, và người nào đó quyết định sửa lỗi chính tả đó, khiến trình biên dịch không có khả năng hiểu nó (cụ thể – tạo ra một biến tên *klass* chỉ vì tên *class* đã được dùng cho thứ gì đó).

Mặc dù trình biên dịch có thể làm việc với những tên này, nhưng điều đó không có nghĩa là bạn được phép dùng nó. Nếu tên khác nhau, thì chúng cũng có ý nghĩa khác nhau.

Những tên dạng chuỗi số (`a1, a2,… aN`) đi ngược lại nguyên tắc đặt tên có mục đích. Mặc dù những tên như vậy không phải là không đúng, nhưng chúng không có thông tin. Chúng không cung cấp manh mối nào về ý định của tác giả. Ví dụ:

```
public static void copyChars(char a1[], char a2[]) {
    for (int i = 0; i < a1.length; i++) {
        a2[i] = a1[i];
    }
}
```

Hàm này dễ đọc hơn nhiều khi *nguyên nhân* và *mục đích* của nó được đặt tên cho các đối số.

Những từ gây nhiễu tạo nên sự khác biệt, nhưng là sự khác biệt vô dụng. Hãy tưởng tượng rằng bạn có một lớp `Product`, nếu bạn có một `ProductInfo` hoặc `ProductData` khác, thì bạn đã thành công trong việc tạo ra các tên khác nhau nhưng về mặt ngữ nghĩa thì chúng là một. `Info` và `Data` là các từ gây nhiễu, giống như `a`, `an` và `the`.

Lưu ý rằng không có gì sai khi sử dụng các tiền tố như `a` và `the` để tạo ra những khác biệt hữu ích. Ví dụ, bạn có thể sử dụng `a` cho tất cả các biến cục bộ và tất cả các đối số của hàm. `a` và `the` sẽ trở thành vô dụng khi bạn quyết định tạo một biến `theZork` vì trước đó bạn đã có một biến mang tên `Zork`.

Những từ gây nhiễu là không cần thiết. Từ `variable` sẽ không bao giờ xuất hiện trong tên biến, từ `table` cũng không nên dùng trong tên bảng. `NameString` sao lại tốt hơn `Name`? `Name` có bao giờ là một số đâu mà lại? Nếu `Name` là một số, nó đã phá vỡ nguyên tắc *Tránh sai lệch thông tin.* Hãy tưởng tượng bạn đang tìm kiếm một lớp có tên `Customer`, và một lớp khác có tên `CustomerObject`. Chúng khác nhau kiểu gì? Cái nào chứa lịch sử thanh toán của khách hàng? Còn cái nào chứa thông tin của khách?

Có một ứng dụng minh họa cho các lỗi trên, chúng tôi đã thay đổi một chút về tên để bảo vệ tác giả. Đây là những thứ chúng tôi thấy trong mã nguồn:

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

Tôi thắc mắc không biết các lập trình viên trong dự án này phải `getActiveAccount` như thế nào!

Trong trường hợp không có quy ước cụ thể, biến `moneyAmount` không thể phân biệt được với `money`; `customerInfo` không thể phân biệt được với `customer`; `accountData` không thể phân biệt được với `account` và `theMessage` với `message` được xem là một. Hãy phân biệt tên theo cách cung cấp cho người đọc những khác biệt rõ ràng.

Dùng những tên phát âm được

Con người rất giỏi về từ ngữ. Một phần quan trọng trong bộ não của chúng ta được dành riêng cho các khái niệm về từ. Và các từ, theo định nghĩa, có thể phát âm được. Thật lãng phí khi không sử dụng được bộ não mà chúng ta đã tiến hóa nhằm thích nghi với ngôn ngữ nói. Vậy nên, hãy làm cho những cái tên phát âm được đi nào.

Nếu bạn không thể phát âm nó, thì bạn không thể thảo luận một cách bình thường: "Hey, ở đây chúng ta có *bee cee arr three cee enn tee*, và *pee ess zee kyew int*, thấy chứ?" – Vâng, tôi thấy một thằng thiểu năng. Vấn đề này rất quan trọng vì lập trình cũng là một hoạt động xã hội, chúng ta cần trao đổi với mọi người.

Tôi có biết một công ty dùng tên `genymdhms` (generation date, year, month, day, hour, minute, and second – phát sinh ngày, tháng, năm, giờ, phút, giây), họ đi xung quanh tôi và "gen why emm dee aich emm ess" (cách phát âm theo tiếng Anh). Tôi có thói quen phát âm như những gì tôi viết, vì vậy tôi bắt đầu nói "gen-yah-muddahims". Sau này nó được gọi bởi một loạt các nhà thiết kế và phân tích, và nghe vẫn có vẻ ngớ ngẩn. Chúng tôi đã từng troll nhau như thế, nó rất thú vị. Nhưng dẫu thế nào đi nữa, chúng tôi đã chấp nhận những cái tên xấu xí. Những lập trình viên mới của công ty tìm hiểu ý nghĩa của các biến, và sau đó họ nói về những từ ngớ ngẩn, thay vì dùng các thuật ngữ tiếng Anh cho thích hợp. Hãy so sánh:

```
class DtaRcrd102 {
    private Date genymdhms;
    private Date modymdhms;
    private final String pszqint = "102";
    /* ... */
};
```

và

```
class Customer {
    private Date generationTimestamp;
    private Date modificationTimestamp;
    private final String recordId = "102";
    /* ... */
};
```

Cuộc trò chuyện giờ đây đã thông minh hơn: "Hey, Mikey, take a look at this record! The generation timestamp is set to tomorrow's date! How can that be?"

Dùng những tên tìm kiếm được

Các tên một chữ cái và các hằng số luôn có vấn đề, đó là không dễ để tìm chúng trong hàng ngàn dòng code.

Người ta có thể dễ dàng tìm kiếm `MAX_CLASSES_PER_STUDENT`, nhưng số 7 thì lại rắc rối hơn. Các công cụ tìm kiếm có thể mở các tệp, các hằng, hoặc các biểu thức chứa số 7 này, nhưng được sử dụng với các mục đích khác nhau. Thậm chí còn tồi tệ hơn khi hằng số là một số có giá trị lớn và ai đó vô tình thay đổi giá trị của nó, từ đó tạo ra một lỗi mà các lập trình viên không tìm ra được.

Tương tự như vậy, tên `e` là một sự lựa chọn tồi tệ cho bất kỳ biến nào mà một lập trình viên cần tìm kiếm. Nó là chữ cái phổ biến nhất trong tiếng anh và có khả năng xuất hiện trong mọi đoạn code của chương trình. Về vấn đề này, tên dài thì tốt hơn tên ngắn, và những cái tên tìm kiếm được sẽ tốt hơn một hằng số trơ trọi trong code.

Sở thích cá nhân của tôi là chỉ đặt tên ngắn cho những biến cục bộ bên trong những phương thức ngắn. *Độ dài của tên phải tương ứng với phạm vi hoạt động của nó*. Nếu một biến hoặc hằng số được nhìn thấy và sử dụng ở nhiều vị trí trong phần thân của mã nguồn, bắt buộc phải đặt cho nó một tên dễ tìm kiếm. Ví dụ:

```
for (int j=0; j<34; j++) {
    s += (t[j]*4)/5;
}
```

và

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
    int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
    int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
    sum += realTaskWeeks;
}
```

Lưu ý rằng `sum` ở ví dụ trên, dù không phải là một tên đầy đủ nhưng có thể tìm kiếm được. Biến và hằng được cố tình đặt tên dài, nhưng hãy so sánh việc tìm kiếm `WORK_DAYS_PER_WEEK` dễ hơn bao nhiêu lần so với số 5, đó là chưa kể cần phải lọc lại danh sách tìm kiếm và tìm ra những trường hợp có nghĩa.

Tránh việc mã hóa

Các cách mã hóa hiện tại là đủ với chúng tôi. Mã hóa các kiểu dữ liệu hoặc phạm vi thông tin vào tên chỉ đơn giản là thêm một gánh nặng cho việc giải mã. Điều đó không hợp lý khi bắt nhân viên phải học thêm một "ngôn ngữ" mã hóa khác khác ngoài các ngôn ngữ mà họ dùng để làm việc với code. Đó là một gánh nặng không cần thiết, các tên mã hóa ít khi được phát âm và dễ bị đánh máy sai.

Ký pháp Hungary

Ngày trước, khi chúng tôi làm việc với những ngôn ngữ mà độ dài tên là một thách thức, chúng tôi đã loại bỏ sự cần thiết này. Fortran bắt buộc mã hóa bằng những chữ cái đầu tiên, phiên bản BASIC ban đầu của nó chỉ cho phép đặt tên tối đa 6 ký tự. Ký pháp Hungary (KH) đã giúp ích cho việc đặt tên rất nhiều.

KH thực sự được coi là quan trọng khi Windows C API xuất hiện, khi mọi thứ là một số nguyên, một con trỏ kiểu `void` hoặc là các chuỗi,.... Trong những ngày đó, trình biên dịch không thể kiểm tra được các lỗi về kiểu dữ liệu, vì vậy các lập trình viên cần một cái phao cứu sinh trong việc nhớ các kiểu dữ liệu này.

Trong các ngôn ngữ hiện đại, chúng ta có nhiều kiểu dữ liệu mặc định hơn, và các trình biên dịch có thể phân biệt được chúng. Hơn nữa, mọi người có xu hướng làm cho các lớp, các hàm trở nên nhỏ hơn để dễ dàng thấy nguồn gốc dữ liệu của biến mà họ đang sử dụng.

Các lập trình viên Java thì không cần mã hóa. Các kiểu dữ liệu mặc định là đủ mạnh mẽ, và các công cụ sửa lỗi đã được nâng cấp để chúng có thể phát hiện các vấn đề về dữ liệu trước khi được biên dịch. Vậy nên, hiện nay KH và các dạng mã hóa khác chỉ đơn giản là một loại chướng ngại vật. Chúng làm cho việc đổi tên biến, tên hàm, tên lớp (hoặc kiểu dữ liệu của chúng) trở nên khó khăn hơn. Chúng làm cho code khó đọc, và tạo ra một hệ thống mã hóa có khả năng đánh lừa người đọc:

```
PhoneNumber phoneString;
     // name not changed when type changed!
```

Các thành phần tiền tố

Bạn cũng không cần phải thêm các tiền tố như `m_` vào biến thành viên (member variable) nữa. Các lớp và các hàm phải đủ nhỏ để bạn không cần chúng. Và bạn nên sử dùng các công cụ chỉnh sửa giúp làm nổi bật các biến này, làm cho chúng trở nên khác biệt với phần còn lại.

```
public class Part {
    private String m_dsc; // The textual description
    void setName(String name) {
        m_dsc = name;
    }
}
/*...*/
public class Part {
    String description;
    void setDescription(String description) {
        this.description = description;
    }
}
```

Bên cạnh đó, mọi người cũng nhanh chóng bỏ qua các tiền tố (hoặc hậu tố) để xem phần có ý nghĩa của tên. Càng đọc code, chúng ta càng ít thấy các tiền tố. Cuối cùng, các tiền tố trở nên vô hình, và bị xem là một dấu hiệu của những dòng code lạc hậu.

Giao diện và thực tiễn

Có một số trường hợp đặc biệt cần mã hóa. Ví dụ: bạn đang xây dựng một ABSTRACT FACTORY. Factory sẽ là giao diện và sẽ được thực hiện bởi một lớp cụ thể. Bạn sẽ đặt tên cho chúng là gì? `IShapeFactory` và `ShapeFactory` ? Tôi thích dùng những cách đặt tên đơn giản. Trước đây, `I` rất phổ biến trong các tài liệu, nó làm chúng tôi phân tâm và đưa ra quá nhiều thông tin. Tôi không muốn người dùng biết rằng tôi đang tạo cho họ một giao diện, tôi chỉ muốn họ biết rằng đó là `ShapeFactory`. Vì vậy, nếu phải lựa chọn việc mã hóa hay thể hiện đầy đủ, tôi sẽ chọn cách thứ nhất. Gọi nó là `ShapeFactoryImp`, hoặc thậm chí là `CShapeFactory` là cách hoàn hảo để che giấu thông tin.

Tránh "hiếp râm não" người khác

Những lập trình viên khác sẽ không cần phải điên đầu ngồi dịch các tên mà bạn đặt thành những tên mà họ biết. Vấn đề này thường phát sinh khi bạn chọn một thuật ngữ không chính xác.

Đây là vấn đề với các tên biến đơn. Chắc chắn một vong lặp có thể sử dụng các biến được đặt tên là `i`, `j` hoặc `k` (không bao giờ là `l` – dĩ nhiên rồi) nếu phạm vi của nó là rất nhỏ và không có tên khác xung đột với nó. Điều này là do việc đặt tên có một chữ cái trong vòng lặp đã trở thành truyền thống. Tuy nhiên, trong hầu hết trường hợp, tên một chữ cái không phải là sự lựa chọn tốt. Nó chỉ là một tên đầu gấu, bắt người đọc phải điên đầu tìm hiểu ý nghĩa, vai trò của nó. Không có lý do nào tồi tệ hơn cho cho việc sử dụng tên `c` chỉ vì `a` và `b` đã được dùng trước đó.

Nói chung, lập trình viên là những người khá thông minh. Và những người thông minh đôi khi muốn thể hiện điều đó bằng cách hack não người khác. Sau tất cả, nếu bạn đủ khả năng nhớ `r` là *the lower-cased version of the url with the host and scheme removed,* thì rõ ràng là – bạn cực kỳ thông minh luôn.

Sự khác biệt giữa lập trình viên thông minh và lập trình viên chuyên nghiệp là họ – những người chuyên nghiệp hiểu rằng sự rõ ràng là trên hết. Các chuyên gia dùng khả năng của họ để tạo nên những dòng code mà người khác có thể hiểu được.

Tên lớp

Tên lớp và các đối tượng nên sử dụng danh từ hoặc cụm danh từ, như `Customer`, `WikiPage`, `Account`, và `AddressParser`. Tránh những từ như `Manager`, `Processor`, `Data`, hoặc `Info` trong tên của một lớp. Tên lớp không nên dùng động từ.

Tên các phương thức

Tên các phương thức nên có động từ hoặc cụm động từ như `postPayment`, `deletePage`, hoặc `save`. Các phương thức truy cập, chỉnh sửa thuộc tính phải được đặt tên cùng với `get`, `set` và `is` theo tiêu chuẩn của Javabean.

```
string name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())...
```

Khi các hàm khởi tạo bị nạp chồng, sử dụng các phương thức tĩnh có tên thể hiện được đối số sẽ tốt hơn. Ví dụ:

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

sẽ tốt hơn câu lệnh

```
Complex fulcrumPoint = new Complex(23.0);
```

Xem xét việc thực thi chúng bằng các hàm khởi tạo private tương ứng.

Đừng thể hiện rằng bạn cute

Nếu tên quá hóm hỉnh, chúng sẽ chỉ được nhớ bởi tác giả và những người bạn. Liệu có ai biết chức năng của hàm `HolyHandGrenade` không? Nó rất thú vị, nhưng trong trường hợp này, `DeleteItems` sẽ là tên tốt hơn. Chọn sự rõ ràng thay vì giải trí.

Sự cute thường xuất hiện dưới dạng phong tục hoặc tiếng lóng. Ví dụ: đừng dùng `whack()` thay thế cho `kill()`, đừng mang những câu đùa trong văn hóa nước mình vào code, như `eatMyShorts()` có nghĩa là `abort()`.

Say what you mean. Mean what you say.

Chọn một từ cho mỗi khái niệm

Chọn một từ cho một khái niệm và gắn bó với nó. Ví dụ, rất khó hiểu khi `fetch`, `retrieve` và `get` là các phương thức có cùng chức năng, nhưng lại đặt tên khác nhau ở các lớp khác nhau. Làm thế nào để nhớ phương thức nào đi với lớp nào? Buồn thay, bạn phải nhớ tên công ty, nhóm hoặc cá nhân nào đã viết ra các thư viện hoặc các lớp, để nhớ cụm từ nào được dùng cho các phương thức. Nếu không, bạn sẽ mất thời gian để tìm hiểu chúng trong các đoạn code trước đó.

Các công cụ chỉnh sửa hiện đại như Eclipse và IntelliJ cung cấp các định nghĩa theo ngữ cảnh, chẳng hạn như danh sách các hàm bạn có thể gọi trên một đối tượng nhất định. Nhưng lưu ý rằng, danh sách thường không cung cấp cho bạn các ghi chú bạn đã viết xung quanh tên hàm và danh sách tham

số. Bạn may mắn nếu nó cung cấp tên tham số từ các khai báo hàm. Tên hàm phải đứng một mình, và chúng phải nhất quán để bạn có thể chọn đúng phương pháp mà không cần phải tìm hiểu thêm.

Tương tự như vậy, rất khó hiểu khi `controller`, `manager` và `driver` lại xuất hiện trong cùng một mã nguồn. Có sự khác biệt nào giữa `DeviceManager` và `ProtocolController`? Tại sao cả hai đều không phải là `controller` hay `manager`? Hay cả hai đều cùng là `driver`? Tên dẫn bạn đến hai đối tượng có kiểu khác nhau, cũng như có các lớp khác nhau.

Một từ phù hợp chính là một ân huệ cho những lập trình viên phải dùng code của bạn.

Đừng chơi chữ

Tránh dùng cùng một từ cho hai mục đích. Sử dụng cùng một thuật ngữ cho hai ý tưởng khác nhau về cơ bản là một cách chơi chữ.

Nếu bạn tuân theo nguyên tắc *Chọn một từ cho mỗi khái niệm,* bạn có thể kết thúc nhiều lớp với một…Ví dụ, phương thức `add`. Miễn là danh sách tham số và giá trị trả về của các phương thức `add` này tương đương về ý nghĩa, tất cả đều tốt.

Tuy nhiên, người ta có thể quyết định dùng từ `add` khi người đó không thực sự tạo nên một hàm có cùng ý nghĩa với cách hoạt động của hàm `add`. Giả sử chúng tôi có nhiều lớp, trong đó `add` sẽ tạo một giá trị mới bằng cách cộng hoặc ghép hai giá trị hiện tại. Bây giờ, giả sử chúng tôi đang viết một lớp mới và có một phương thức thêm tham số của nó vào mảng. Chúng tôi có nên gọi nó là `add` không? Có vẻ phù hợp đấy, nhưng trong trường hợp này, ý nghĩa của chúng là khác nhau, vậy nên chúng tôi dùng một cái tên khác như `insert` hay `append` để thay thế. Nếu được dùng cho phương thức mới, `add` chính xác là một kiểu chơi chữ.

Mục tiêu của chúng tôi, với tư cách là tác giả, là làm cho code của chúng tôi dễ hiểu nhất có thể. Chúng tôi muốn code của chúng tôi là một bài viết ngắn gọn, chứ không phải là một bài nghiên cứu […].

Dùng thuật ngữ

Hãy nhớ rằng những người đọc code của bạn là những lập trình viên, vậy nên hãy sử dụng các thuật ngữ khoa học, các thuật toán, tên mẫu (pattern),…cho việc đặt tên. Sẽ không khôn ngoan khi bạn đặt tên của vấn đề theo cách mà khách hàng định nghĩa. Chúng tôi không muốn đồng nghiệp của chúng tôi phải tìm khách hàng để hỏi ý nghĩa của tên, trong khi họ đã biết khái niệm đó – nhưng là dưới dạng một cái tên khác.

Tên `AccountVisitor` có ý nghĩa rất nhiều đối với một lập trình viên quen thuộc với mô hình VISITOR (VISITOR pattern). Có lập trình viên nào không biết `JobQueue`? Có rất nhiều thứ liên quan đến kỹ thuật mà lập trình viên phải đặt tên. Chọn những tên thuật ngữ thường là cách tốt nhất.

Thêm ngữ cảnh thích hợp

Chỉ có một vài cái tên có nghĩa trong mọi trường hợp – số còn lại thì không. Vậy nên, bạn cần đặt tên phù hợp với ngữ cảnh, bằng cách đặt chúng vào các lớp, các hàm hoặc các không gian tên (namespace). Khi mọi thứ thất bại, tiền tố nên được cân nhắc như là giải pháp cuối cùng.

Hãy tưởng tượng bạn có các biến có tên là `firstName`, `lastName`, `street`, `houseNumber`, `city`, `state` và `zipcode`. Khi kết hợp với nhau, chúng rõ ràng tạo thành một địa chỉ. Nhưng nếu bạn chỉ thấy biến `state` được sử dụng một mình trong một phương thức thì sao? Bạn có thể suy luận ra đó là một phần của địa chỉ không?

Bạn có thể thêm ngữ cảnh bằng cách sử dụng tiền tố: `addrFirstName`, `addrLastName`, `addrState`,... Ít nhất người đọc sẽ hiểu rằng những biến này là một phần của một cấu trúc lớn hơn. Tất nhiên, một giải pháp tốt hơn là tạo một lớp có tên là `Address`. Khi đó, ngay cả trình biên dịch cũng biết rằng các biến đó thuộc về một khái niệm lớn hơn.

Hãy xem xét các phương thức trong *Listing 2-1*. Các biến có cần một ngữ cảnh có ý nghĩa hơn không? Tên hàm chỉ cung cấp một phần của ngữ cảnh, thuật toán cung cấp phần còn lại. Khi bạn đọc qua hàm, bạn thấy rằng ba biến, `number`, `verb` và `pluralModifier`, là một phần của thông báo "giả định thống kê". Thật không may, bối cảnh này phải suy ra mới có được. Khi bạn lần đầu xem xét phương thức, ý nghĩa của các biến là không rõ ràng.

Listing 2-1

Biến với bối cảnh không rõ ràng.

```
private void printGuessStatistics(char candidate, int count) {
    String number;
    String verb;
    String pluralModifier;
    if (count == 0) {
        number = "no";
        verb = "are";
        pluralModifier = "s";
    } else if (count == 1) {
        number = "1";
        verb = "is";
        pluralModifier = "";
    } else {
        number = Integer.toString(count);
        verb = "are";
        pluralModifier = "s";
    }
```

```
    String guessMessage = String.format("There %s %s %s%s", verb,
                                number, candidate, pluralModifier);
    print(guessMessage);
}
```

Hàm này hơi dài và các biến được sử dụng xuyên suốt. Để tách hàm thành các phần nhỏ hơn, chúng ta cần tạo một lớp `GuessStatisticsMessage` và tạo ra ba biến của lớp này. Điều này cung cấp một bối cảnh rõ ràng cho ba biến. Chúng là một phần của `GuessStatisticsMessage`. Việc cải thiện bối cảnh cũng cho phép thuật toán được rõ ràng hơn bằng cách chia nhỏ nó thành nhiều chức năng nhỏ hơn. (Xem *Listing 2-2*.)

Listing 2-2

Biến có ngữ cảnh

```
public class GuessStatisticsMessage {
    private String number;
    private String verb;
    private String pluralModifier;
    public String make(char candidate, int count) {
        createPluralDependentMessageParts(count);
        return String.format("There %s %s %s%s",verb, number, candidate,
                                        pluralModifier );
    }
    private void createPluralDependentMessageParts(int count) {
        if (count == 0) {
            thereAreNoLetters();
        } else if (count == 1) {
            thereIsOneLetter();
        } else {
            thereAreManyLetters(count);
        }
    }
    private void thereAreManyLetters(int count) {
        number = Integer.toString(count);
        verb = "are";
        pluralModifier = "s";
    }
    private void thereIsOneLetter() {
        number = "1";
        verb = "is";
```

```
        pluralModifier = "";
    }
    private void thereAreNoLetters() {
        number = "no";
        verb = "are";
        pluralModifier = "s";
    }
}
```

Tên ngắn thường tốt hơn tên dài, miễn là chúng rõ ràng. Thêm đủ ngữ cảnh cho tên sẽ tốt hơn khi cần thiết.

Tên `accountAddress` và `customerAddress` là những tên đẹp cho trường hợp đặc biệt của lớp `Address` nhưng có thể là tên tồi cho các lớp khác. `Address` là một tên đẹp cho lớp. Nếu tôi cần phân biệt giữa địa chỉ MAC, địa chỉ cổng (port) và địa chỉ web thì tôi có thể xem xét `MAC`, `PostalAddress` và `URL`. Kết quả là tên chính xác hơn. Đó là tâm điểm của việc đặt tên.

Lời kết

Điều khó khăn nhất của việc lựa chọn tên đẹp là nó đòi hỏi kỹ năng mô tả tốt và nền tảng văn hóa lớn. Đây là vấn đề về học hỏi hơn là vấn đề kỹ thuật, kinh doanh hoặc quản lý. Kết quả là nhiều người trong lĩnh vực này không học cách làm điều đó.

Mọi người cũng sợ đổi tên mọi thứ vì lo rằng người khác sẽ phản đối. Chúng tôi không chia sẻ nỗi sợ đó cho bạn. Chúng tôi thật sự biết ơn những ai đã đổi tên khác cho biến, hàm,…(theo hướng tốt hơn). Hầu hết thời gian chúng tôi không thật sự nhớ tên lớp và những phương thức của nó. Chúng tôi có các công cụ giúp chúng tôi trong việc đó để chúng tôi có thể tập trung vào việc code có dễ đọc hay không. Bạn có thể sẽ gây ngạc nhiên cho ai đó khi bạn đổi tên, giống như bạn có thể làm với bất kỳ cải tiến nào khác. Đừng để những cái tên tồi phá hủy sự nghiệp coder của mình.

Thực hiện theo một số quy tắc trên và xem liệu bạn có cải thiện được khả năng đọc code của mình hay không. Nếu bạn đang bảo trì code của người khác, hãy sử dụng các công cụ tái cấu trúc để giải quyết vấn đề này. Mất một ít thời gian nhưng có thể làm bạn nhẹ nhõm trong vài tháng.