

chukimmuo / Clean-Code---Tieng-Viet Public

forked from quoctinnguyen8/Clean-Code---Tieng-Viet

[Code](#) [Pull requests](#) [Actions](#) [Projects](#) [Security](#) [Insights](#)

[master](#) ▾

...

Clean-Code---Tieng-Viet / Markdown / Chaper10.md



chukimmuo Add Chaper10 - ver3.md

[History](#)

1 contributor

598 lines (500 sloc) | 38.7 KB

...

Lớp

bởi Jeff Langr



Cho đến nay trong cuốn sách này, chúng ta đã tập trung vào cách viết tốt các dòng và khối mã. Đi sâu vào thành phần thích hợp của các chức năng và cách chúng tương tác với nhau. Nhưng sau tất cả, sự chú ý đến tính biểu đạt của các câu lệnh và các chức năng mà chúng bao gồm, vẫn chưa có mã sạch cho đến khi quan tâm đến các cấp tổ chức mã cao hơn. Hãy nói về các mã lớp sạch.

Tổ chức lớp

Theo quy ước Java tiêu chuẩn, một lớp phải bắt đầu bằng một danh sách các biến. Các **public static constants**, nếu có, nên xuất hiện trước. Sau đó là các biến **private static**, tiếp theo là các biến **private**. Ít khi có lý do chính đáng để có một biến là **public**.

Các hàm **public** nên tuân theo danh sách các biến. Chúng tôi muốn đặt các tiện ích **private** được gọi bởi một chức năng **public** ngay sau chức năng đó. Điều này tuân theo quy tắc nhìn xuống và giúp chương trình đọc giống như một bài văn.

Đóng gói

Chúng ta muốn giữ các biến và hàm của mình ở chế độ riêng tư, nhưng không nên quá cứng nhắc về nó. Đôi khi cần một biến hoặc một hàm **protected** để có thể sử dụng kiểm tra. Quy tắc kiểm tra nên được ưu tiên. Nếu một thử nghiệm trong cùng một gói cần gọi một hàm hoặc truy cập một biến, chúng tôi sẽ đặt nó là **protected** hoặc đặt nó trong phạm vi gói. Tuy nhiên, trước tiên, chúng ta sẽ tìm cách duy trì quyền riêng tư. Nói lỏng sự đóng gói luôn là phương sách cuối cùng.

Lớp nên nhỏ!

Quy tắc đầu tiên của các lớp là chúng phải nhỏ. Quy tắc thứ hai của các lớp là chúng phải nhỏ hơn thế. Đúng vậy, tôi lặp lại từ ngữ chính xác từ chương **Hàm**. Cũng như các hàm, nhỏ hơn là quy tắc chính khi thiết kế các lớp. Đối với các chức năng, câu hỏi trước mắt luôn là "**Nhỏ như thế nào?**"

Với các chức năng, chúng ta đo kích thước bằng cách đếm số dòng. Với các lớp, chúng tôi sử dụng một thước đo khác. Chúng ta tính đến trách nhiệm.

Listing 10-1 phác thảo một lớp, **SuperDashboard**, hiển thị khoảng 70 phương thức **public**. Hầu hết các nhà phát triển sẽ đồng ý rằng nó có kích thước hơi quá lớn. Một số nhà phát triển có thể coi **SuperDashboard** là "**lớp học của Chúa**".

Listing 10-1

Quá nhiều trách nhiệm

```
public class SuperDashboard extends JFrame implements MetaDataUser {  
    public String getCustomizerLanguagePath();  
    public void setSystemConfigPath(String systemConfigPath);  
    public String getSystemConfigDocument();  
    public void setSystemConfigDocument(String systemConfigDocument);  
    public boolean getGuruState();  
    public boolean getNoviceState();  
    public boolean getOpenSourceState();  
    public void showObject(MetaObject object);  
    public void showProgress(String s);  
    public boolean isMetadataDirty();  
    public void setIsMetadataDirty(boolean isMetadataDirty);  
    public Component getLastFocusedComponent();  
    public void setLastFocused(Component lastFocused);  
    public void setMouseSelectState(boolean isMouseSelected);  
    public boolean isMouseSelected();  
    public LanguageManager getLanguageManager();  
    public Project getProject();  
    public Project getFirstProject();  
    public Project getLastProject();  
    public String getNewProjectName();
```

```
public void setComponentSizes(Dimension dim);
public String getCurrentDir();
public void setCurrentDir(String newDir);
public void updateStatus(int dotPos, int markPos);
public Class[] get DataBaseClasses();
public MetadataFeeder getMetadataFeeder();
public void addProject(Project project);
public boolean setCurrentProject(Project project);
public boolean removeProject(Project project);
public MetaProjectHeader getProgramMetadata();
public void resetDashboard();
public Project loadProject(String fileName, String projectName);
public void setCanSaveMetadata(boolean canSave);
public MetaObject getSelectedObject();
public void deselectObjects();
public void setProject(Project project);
public void editorAction(String actionPerformed, ActionEvent event);
public void setMode(int mode);
public FileManager getFileManager();
public void setFileManager(FileManager fileManager);
public ConfigManager getConfigManager();
public void setConfigManager(ConfigManager configManager);
public ClassLoader getClassLoader();
public void setClassLoader(ClassLoader classLoader);
public Properties getProps();
public String getUserHome();
public String getBaseDir();
public int getMajorVersionNumber();
public int getMinorVersionNumber();
public int getBuildNumber();
public MetaObject pasting(
    MetaObject target, MetaObject pasted, MetaProject project);
public void processMenuItems(MetaObject metaObject);
public void processMenuSeparators(MetaObject metaObject);
public void processTabPages(MetaObject metaObject);
public void processPlacement(MetaObject object);
public void processCreateLayout(MetaObject object);
public void updateDisplayLayer(MetaObject object, int layerIndex);
public void propertyEditedRepaint(MetaObject object);
public void processDeleteObject(MetaObject object);
public boolean getAttachedToDesigner();
public void processProjectChangedState(boolean hasProjectChanged);
public void processObjectNameChanged(MetaObject object);
public void runProject();
public void setAllowDragging(boolean allowDragging);
public boolean allowDragging();
public boolean isCustomizing();
public void setTitle(String title);
public IdeMenuBar getIdeMenuBar();
```

```
public void showHelper(MetaObject metaObject, String propertyName);
// ... many non-public methods follow ...
}
```

Nhưng nếu **SuperDashboard** chỉ chứa các phương thức được hiển thị trong Listing 10-2 thì sao?

Listing 10-2

Đủ nhỏ?

```
public class SuperDashboard extends JFrame implements MetaDataUser {
    public Component getLastFocusedComponent();
    public void setLastFocused(Component lastFocused);
    public int getMajorVersionNumber();
    public int getMinorVersionNumber();
    public int getBuildNumber();
}
```

Năm phương pháp không phải là quá nhiều, phải không? Trong trường hợp này, đó là bởi vì mặc dù có số lượng các phương pháp nhỏ, nhưng **SuperDashboard** lại có quá nhiều **trách nhiệm**.

Tên của một lớp phải mô tả những trách nhiệm mà nó thực hiện. Trên thực tế, đặt tên có lẽ là cách đầu tiên giúp xác định quy mô lớp. Nếu chúng ta không thể tìm ra một tên ngắn gọn cho một lớp, thì có thể nó quá lớn. Tên lớp càng mơ hồ thì càng có nhiều khả năng nó có quá nhiều trách nhiệm. Ví dụ: tên lớp bao gồm các từ như **Processor** hoặc **Manager** hoặc **Super** thường gợi ý về sự kết hợp của nhiều trách nhiệm.

Chúng ta cũng có thể viết mô tả ngắn gọn về lớp học trong khoảng 25 từ, không sử dụng các từ “nếu”, “và”, “hoặc” hoặc “nhưng”. Chúng ta sẽ mô tả **SuperDashboard** như thế nào? “**SuperDashboard** cung cấp quyền truy cập vào thành phần giữ tiêu điểm **và** nó cũng cho phép chúng tôi theo dõi phiên bản **cùng** số lượng bản dựng”. Chữ “**và**” đầu tiên là dấu hiệu rằng **SuperDashboard** có quá nhiều trách nhiệm.

Nguyên tắc trách nhiệm duy nhất

Nguyên tắc Trách nhiệm Đơn lẻ (SRP) nêu rõ rằng một lớp hoặc mô-đun nên có một và **chỉ một lý do để thay đổi**. Nguyên tắc này cung cấp cho chúng ta cả định nghĩa về trách nhiệm và hướng dẫn về quy mô lớp học. Các lớp chỉ nên có một trách nhiệm — chỉ một lý do để thay đổi.

Lớp **SuperDashboard** có vẻ nhỏ trong Listing 10-2, nhưng có hai lý do để thay đổi. **Đầu tiên**, nó theo dõi thông tin về phiên bản đường như cần được cập nhật mỗi khi phần mềm được xuất xưởng. **Thứ hai**, nó quản lý các thành phần Java Swing (nó là một dẫn xuất của **JFrame**, đại diện Swing của một cửa sổ GUI cấp cao nhất). Không nghi ngờ gì nữa, chúng ta sẽ muốn cập nhật số phiên bản nếu chúng tôi thay đổi bất kỳ mã Swing nào, tuy nhiên có trường hợp khác đó là: Chúng ta có thể thay đổi thông tin phiên bản dựa trên những thay đổi với mã khác trong hệ thống.

Cố gắng xác định trách nhiệm (lý do để thay đổi) thường giúp chúng ta nhận ra và tạo ra những hàm tốt hơn trong mã của mình. Chúng ta có thể dễ dàng trích xuất cả ba phương thức **SuperDashboard** xử lý thông tin phiên bản vào một lớp riêng biệt có tên **Version**. (Xem Listing 10-3.) Lớp **Version** là một cấu trúc có tiềm năng cao để sử dụng lại trong các ứng dụng khác!

Listing 10-3

Một lớp trách nhiệm duy nhất

```
public class Version {
    public int getMajorVersionNumber();
    public int getMinorVersionNumber();
    public int getBuildNumber();
}
```

SRP là một trong những khái niệm quan trọng nhất trong thiết kế OO. Đây cũng là một trong những khái niệm đơn giản để hiểu và tuân thủ. Tuy nhiên, kỳ lạ là SRP thường là nguyên tắc thiết kế lớp bị vi phạm nhiều nhất. Chúng tôi thường xuyên gặp phải các lớp làm quá nhiều thứ. Tại sao?

Làm cho phần mềm chạy và làm cho phần mềm sạch là hai việc khác nhau. Hầu hết chúng ta đều nghĩ ưu tiên phần mềm chạy được trước, vì vậy chúng ta tập trung vào việc làm cho mã của mình chạy đúng ngữ cảnh nhiều hơn là tính tổ chức và sự sạch sẽ. Điều này hoàn toàn phù hợp. Tuy nhiên duy trì sự tách biệt cũng quan trọng trong việc lập trình như việc chạy được các chương trình vậy.

Vấn đề là có quá nhiều người nghĩ rằng đã hoàn thành công việc khi chương trình chạy đúng. Chúng ta không để tâm đến việc tổ chức mã và sự sạch sẽ. Và ngay lập tức chuyển sang vấn đề tiếp theo thay vì xem lại và **chia nhỏ các lớp** được nhét quá nhiều thành các **lớp nhỏ hơn với một trách nhiệm duy nhất**.

Đồng thời, nhiều nhà phát triển lo ngại rằng một số lượng lớn các lớp nhỏ, khiến việc hiểu bức tranh lớn trở nên khó khăn hơn. Họ lo ngại rằng phải di chuyển từ lớp này sang lớp khác để tìm ra cách hoàn thành một phần công việc lớn hơn.

Tuy nhiên, việc di chuyển giữa các lớp của một hệ thống có nhiều lớp nhỏ không nhiều hơn hệ thống có một vài lớp lớn. Có nhiều thứ cần học trong hệ thống có một vài lớp học lớn. Vậy câu hỏi đặt ra là: Bạn có muốn các công cụ của mình được tổ chức thành các hộp công cụ với nhiều ngăn kéo nhỏ, mỗi ngăn chứa các thành phần được xác định rõ ràng và được dán nhãn rõ ràng? Hay bạn muốn có một vài ngăn kéo mà bạn chỉ việc ném mọi thứ vào?

Mỗi hệ thống lớn sẽ chứa một lượng lớn logic và độ phức tạp. Mục tiêu chính trong việc quản lý sự phức tạp đó là tổ chức nó để một nhà phát triển biết nơi cần tìm, để tìm mọi thứ và chỉ cần hiểu sự phức tạp bị ảnh hưởng trực tiếp tại bất kỳ thời điểm nào. Ngược lại, một hệ thống với các lớp đa dụng, nhiều chức năng luôn cần trao đổi chúng ta bằng cách yêu cầu chúng ta để ý qua rất nhiều thứ mà chúng ta không cần biết ngay khi đó.

Nhắc lại các điểm trọng: Chúng ta muốn hệ thống của mình bao gồm nhiều lớp nhỏ, không phải một vài lớp lớn. Mỗi lớp nhỏ chứa đựng **một** khả năng đáp ứng duy nhất, có một lý do duy nhất để thay đổi và cộng tác với một vài lớp khác để đạt được các hành vi hệ thống mong muốn.

Sự gắn kết

Các lớp nên có một số lượng nhỏ các biến **instance**. Mỗi phương thức của một lớp nên thao tác với một hoặc nhiều biến đó. Nói chung, một phương thức càng thao tác với nhiều biến thì phương thức đó càng gắn kết với lớp của nó. Một lớp trong đó mỗi biến được sử dụng bởi mỗi phương thức là tối đa gắn kết.

Nói chung, không nên và cũng không thể tạo các lớp gắn kết tối đa như vậy; mặt khác, chúng ta muốn sự gắn kết cao. Khi tính liên kết cao, có nghĩa là các phương thức và biến của lớp là đồng phụ thuộc và gắn kết với nhau như một tổng thể logic.

Xem xét việc triển khai một Stack trong Listing 10-4. Đây là một lớp rất gắn kết. Trong ba phương thức, chỉ có **size()** không sử dụng được cả hai biến.

Listing 10-4

Stack.java A cohesive class.

```
public class Stack {
    private int topOfStack = 0;
    List<Integer> elements = new LinkedList<Integer>();

    public int size() {
        return topOfStack;
    }

    public void push(int element) {
```

```
        topOfStack++;
        elements.add(element);
    }

    public int pop() throws PoppedWhenEmpty {
        if (topOfStack == 0)
            throw new PoppedWhenEmpty();
        int element = elements.get(--topOfStack);
        elements.remove(topOfStack);
        return element;
    }
}
```

Chiến lược giữ các hàm nhỏ và giữ cho danh sách tham số ngắn đôi khi có thể dẫn đến sự gia tăng của các biến `instance` được sử dụng bởi một tập hợp con các phương thức. Khi điều này xảy ra, hầu như luôn luôn có nghĩa là có ít nhất một lớp mang nhiệm vụ khác đang hình thành. Bạn nên cố gắng tách các biến và phương thức thành hai hoặc nhiều lớp để các lớp mới gắn kết hơn.

Duy trì kết quả gắn kết trong nhiều lớp nhỏ

Chỉ cần chia các chức năng lớn thành các chức năng nhỏ hơn cũng gây ra sự gia tăng các lớp. Hãy xem xét một hàm lớn với nhiều biến được khai báo bên trong nó. Giả sử bạn muốn tách một phần nhỏ của hàm đó thành một hàm khác riêng biệt. Tuy nhiên, mã bạn muốn chia sẽ sử dụng 4 trong số các biến được khai báo trong hàm. Bạn có phải chuyển tất cả 4 biến đó vào hàm mới dưới dạng đối số không?

Không vấn đề! Nếu chúng ta đã chuyển 4 biến đó thành các biến `instance` của lớp, thì có thể chia tách mã mà không cần truyền theo bất kỳ biến nào. Sẽ rất dễ dàng để chia các hàm lớn thành nhiều hàm nhỏ.

Thật không may, điều này cũng có nghĩa là các lớp của chúng ta mất tính liên kết vì chúng tích lũy ngày càng nhiều biến `instance` chỉ tồn tại để cho phép một vài hàm sử dụng chúng. Nhưng đợi đã! Nếu có một vài hàm muốn dụng một số biến nhất định, điều đó khiến chúng trở thành một lớp theo đúng nghĩa của chúng không? Tất nhiên là thế. **Khi các lớp mất tính liên kết, hãy chia tách chúng!**

Vì vậy, việc chia một hàm lớn thành nhiều hàm nhỏ hơn thường cho chúng ta cơ hội để tách ra một số lớp nhỏ hơn. Điều này mang lại cho chương trình của chúng ta một cấu trúc minh bạch hơn.

Để minh chứng, hãy sử dụng một ví dụ lâu đời được lấy từ cuốn sách tuyệt vời của Knuth **Lập trình cho văn bản**. Listing 10-5 hiển thị bản dịch sang Java của chương trình **PrintPrimes** của Knuth. Công bằng mà nói với Knuth, đây không phải là chương trình như anh ấy đã viết mà là nó được xuất ra bởi công cụ WEB của anh ấy. Tôi đang sử dụng nó vì nó là một nơi khởi đầu tuyệt vời để chia một hàm lớn thành nhiều hàm và lớp nhỏ hơn.

Listing 10-5

PrintPrimes.java

```
package literatePrimes;

public class PrintPrimes {
    public static void main(String[] args) {
        final int M = 1000;
        final int RR = 50;
        final int CC = 4;
        final int WW = 10;
        final int ORDMAX = 30;
        int P[] = new int[M + 1];
        int PAGENUMBER;
        int PAGEOFFSET;
        int ROWOFFSET;
        int C;
        int J;
        int K;
        boolean JPRIME;
        int ORD;
        int SQUARE;
        int N;
        int MULT[] = new int[ORDMAX + 1];
        J = 1;
        K = 1;
        P[1] = 2;
        ORD = 2;
        SQUARE = 9;
        while (K < M) {
            do {
                J = J + 2;
                if (J == SQUARE) {
                    ORD = ORD + 1;
                    SQUARE = P[ORD] * P[ORD];
                    MULT[ORD - 1] = J;
                }
                N = 2;
                JPRIME = true;
                while (N < ORD && JPRIME) {
                    while (MULT[N] < J) MULT[N] = MULT[N] + P[N] + P[N];
                    if (MULT[N] == J)
```

```
JPRIME = false;
N = N + 1;
}
} while (!JPRIME);
K = K + 1;
P[K] = J;
}
{
while (PAGEOFFSET <= M) {
    PAGENUMBER = 1;
    PAGEOFFSET = 1;
    System.out.println("The First " + M + " Prime Numbers --- Page " + P
    System.out.println("");
    for (ROWOFFSET = PAGEOFFSET; ROWOFFSET < PAGEOFFSET + RR; ROWOFFSET+
        for (C = 0; C < CC; C++)
            if (ROWOFFSET + C * RR <= M)
                System.out.format("%10d", P[ROWOFFSET + C * RR]);
            System.out.println("");
        }
    System.out.println("\f");
    PAGENUMBER = PAGENUMBER + 1;
    PAGEOFFSET = PAGEOFFSET + RR * CC;
}
}
}
```

Chương trình này, được viết dưới dạng một hàm duy nhất, là một mớ hỗn độn. Nó có một cấu trúc sâu, rất nhiều biến số và một cấu trúc liên kết chặt chẽ. Chí ít thì một chức năng lớn nên được chia thành một vài chức năng nhỏ hơn.

Từ Listing 10-6 đến Listing 10-8 cho thấy kết quả của việc tách mã trong Listing 10-5 thành các lớp và hàm nhỏ hơn, đồng thời chọn tên có ý nghĩa cho các lớp, chức năng và biến đó.

Listing 10-6

PrimePrinter.java (refactored)

```
package literatePrimes;

public class PrimePrinter {
    public static void main(String[] args) {
        final int NUMBER_OF_PRIMES = 1000;
        int[] primes = PrimeGenerator.generate(NUMBER_OF_PRIMES);
        final int ROWS_PER_PAGE = 50;
        final int COLUMNS_PER_PAGE = 4;
        RowColumnPagePrinter tablePrinter =
            new RowColumnPagePrinter(ROWS_PER_PAGE, COLUMNS_PER_PAGE,
```

```

        "The First " + NUMBER_OF_PRIMES + " Prime Numbers");
    tablePrinter.print(primes);
}
}

```

Listing 10-7

RowColumnPagePrinter.java

```

package literatePrimes;

import java.io.PrintStream;

public class RowColumnPagePrinter {
    private int rowsPerPage;
    private int columnsPerPage;
    private int numbersPerPage;
    private String pageHeader;
    private PrintStream printStream;

    public RowColumnPagePrinter(int rowsPerPage, int columnsPerPage,
                               String pageHeader) {
        this.rowsPerPage = rowsPerPage;
        this.columnsPerPage = columnsPerPage;
        this.pageHeader = pageHeader;
        numbersPerPage = rowsPerPage * columnsPerPage;
        printStream = System.out;
    }

    public void print(int data[]) {
        int pageNumber = 1;
        for (int firstIndexOnPage = 0;
             firstIndexOnPage < data.length;
             firstIndexOnPage += numbersPerPage) {
            int lastIndexOnPage =
                Math.min(firstIndexOnPage + numbersPerPage - 1, data.length - 1)
            printPageHeader(pageHeader, pageNumber);
            printPage(firstIndexOnPage, lastIndexOnPage, data);
            printStream.println("\f");
            pageNumber++;
        }
    }

    private void printPage(int firstIndexOnPage, int lastIndexOnPage,
                          int[] data) {
        int firstIndexOfLastRowOnPage =
            firstIndexOnPage + rowsPerPage - 1;
        for (int firstIndexInRow = firstIndexOnPage;
             firstIndexInRow <= firstIndexOfLastRowOnPage;

```

```

        firstIndexInRow++ );
    printRow(firstIndexInRow, lastIndexOnPage, data);
    printStream.println("");
}
}

private void printRow(int firstIndexInRow, int lastIndexOnPage,
                      int[] data) {
    for (int column = 0; column < columnsPerPage; column++) {
        int index = firstIndexInRow + column * rowsPerPage;
        if (index <= lastIndexOnPage)
            printStream.format("%10d", data[index]);
    }
}

private void printPageHeader(String pageHeader, int pageNumber) {
    printStream.println(pageHeader + " --- Page " + pageNumber);
    printStream.println("");
}

public void setOutput(PrintStream printStream) {
    this.printStream = printStream;
}
}

```

Listing 10-8

PrimeGenerator.java

```

package literatePrimes;

import java.util.ArrayList;

public class PrimeGenerator {
    private static int[] primes;
    private static ArrayList<Integer> multiplesOfPrimeFactors;

    protected static int[] generate(int n) {
        primes = new int[n];
        multiplesOfPrimeFactors = new ArrayList<Integer>();
        set2AsFirstPrime();
        checkOddNumbersForSubsequentPrimes();
        return primes;
    }

    private static void set2AsFirstPrime() {
        primes[0] = 2;
        multiplesOfPrimeFactors.add(2);
    }
}

```

```
}

private static void checkOddNumbersForSubsequentPrimes() {
    int primeIndex = 1;
    for (int candidate = 3;
        primeIndex < primes.length;
        candidate += 2) {
        if (isPrime(candidate))
            primes[primeIndex++] = candidate;
    }
}

private static boolean isPrime(int candidate) {
    if (isLeastRelevantMultipleOfNextLargerPrimeFactor(candidate)) {
        multiplesOfPrimeFactors.add(candidate);
        return false;
    }
    return isNotMultipleOfAnyPreviousPrimeFactor(candidate);
}

private static boolean isLeastRelevantMultipleOfNextLargerPrimeFactor(int candid
    int nextLargerPrimeFactor = primes[multiplesOfPrimeFactors.size()];
    int leastRelevantMultiple = nextLargerPrimeFactor * nextLargerPrimeFactor;
    return candidate == leastRelevantMultiple;
}

private static boolean isNotMultipleOfAnyPreviousPrimeFactor(int candidate) {
    for (int n = 1; n < multiplesOfPrimeFactors.size(); n++) {
        if (isMultipleOfNthPrimeFactor(candidate, n)) return false;
    }
    return true;
}

private static boolean isMultipleOfNthPrimeFactor(int candidate, int n) {
    return candidate == smallestOddNthMultipleNotLessThanCandidate(candidate, n)
}

private static int
smallestOddNthMultipleNotLessThanCandidate(int candidate, int n) {
    int multiple = multiplesOfPrimeFactors.get(n);
    while (multiple < candidate)
        multiple += 2 * primes[n];
    multiplesOfPrimeFactors.set(n, multiple);
    return multiple;
}
}
```



Điều đầu tiên bạn có thể nhận thấy là chương trình dài hơn rất nhiều. Nó dài từ hơn một trang đến gần ba trang. Có một số lý do cho sự tăng trưởng này. Đầu tiên, chương trình được tái cấu trúc sử dụng các tên biến mô tả dài hơn. Thứ hai, chương trình được tái cấu trúc sử dụng khai báo hàm và lớp như một cách để thêm chú thích vào mã. Thứ ba, chúng tôi đã sử dụng kỹ thuật định dạng và khoảng trắng để giữ cho chương trình có thể đọc được.

Lưu ý rằng chương trình đã được chia thành ba trách nhiệm chính như thế nào. Chương trình chính được chứa trong lớp **PrimePrinter**. Trách nhiệm của nó là xử lý môi trường thực thi. Nó sẽ thay đổi nếu phương thức gọi thay đổi. Ví dụ: nếu chương trình này được chuyển đổi thành dịch vụ SOAP, thì đây là lớp sẽ bị ảnh hưởng.

RowColumnPagePrinter biết tất cả về cách định dạng danh sách các số thành các trang với một số hàng và cột nhất định. Nếu định dạng của đầu ra cần thay đổi, thì đây là lớp sẽ bị ảnh hưởng.

Lớp **PrimeGenerator** cho biết cách tạo một danh sách các số nguyên tố. Lưu ý rằng nó không có nghĩa là được khởi tạo như một đối tượng. Lớp chỉ là một phạm vi hữu ích trong đó các biến của nó có thể được khai báo và giữ ẩn. Lớp này sẽ thay đổi nếu thuật toán tính toán các số nguyên tố thay đổi.

Đây không phải là một bài viết lại! Chúng tôi đã không bắt đầu lại từ đầu và viết lại chương trình. Thật vậy, nếu bạn xem xét kỹ hai chương trình khác nhau, bạn sẽ thấy rằng chúng sử dụng cùng một thuật toán và cơ chế để hoàn thành công việc của mình.

Thay đổi được thực hiện bằng cách viết một bộ thử nghiệm xác minh hành vi chính xác của chương trình đầu tiên. Sau đó, vô số thay đổi nhỏ được thực hiện, mỗi lần một thay đổi. Sau mỗi thay đổi, chương trình được thực thi để đảm bảo rằng hành vi không thay đổi. Hết bước này đến bước khác, chương trình đầu tiên được dọn dẹp và chuyển thành chương trình thứ hai.

Tổ chức để thay đổi

Đối với hầu hết các hệ thống, sự thay đổi là liên tục. Mọi thay đổi đều khiến chúng ta có nguy cơ khiến phần còn lại của hệ thống không còn hoạt động như dự kiến. Trong một hệ thống sạch, chúng ta tổ chức các lớp của mình để giảm nguy cơ phải thay đổi.

Lớp **Sql** trong Liệt kê 10-9 được sử dụng để tạo các chuỗi SQL được định dạng đúng với siêu dữ liệu thích hợp. Đây là một công việc đang được tiến hành và do đó, chưa hỗ trợ chức năng SQL như câu lệnh **update**. Khi đến lúc, lớp **Sql** hỗ trợ chức năng **update**, chúng ta sẽ phải "mở" lớp này để thực hiện sửa đổi. Vấn đề với việc mở một lớp học là nó dẫn đến rủi ro. Bất kỳ sửa đổi nào đối với lớp đều có khả năng phá vỡ mã trong lớp. Nó phải được kiểm tra lại hoàn toàn.

Listing 10-9

A class that must be opened for change

```
public class Sql {
    public Sql(String table, Column[] columns);

    public String create();

    public String insert(Object[] fields);

    public String selectAll();

    public String findByKey(String keyColumn, String keyValue);

    public String select(Column column, String pattern);

    public String select(Criteria criteria);

    public String preparedInsert();

    private String columnList(Column[] columns);

    private String valuesList(Object[] fields, final Column[] columns);

    private String selectWithCriteria(String criteria);

    private String placeholderList(Column[] columns);
}
```

Lớp **Sql** phải thay đổi khi chúng ta thêm một kiểu câu lệnh mới. Nó cũng phải thay đổi khi chúng ta thay đổi chi tiết của một loại câu lệnh đơn lẻ — ví dụ: nếu chúng tôi cần sửa đổi chức năng **select** để hỗ trợ các lựa chọn con. Hai lý do để thay đổi này có nghĩa là lớp **Sql** vi phạm SRP.

Chúng ta có thể phát hiện vi phạm SRP này từ quan điểm tổ chức đơn giản. Sơ lược qua, phương thức của **Sql** cho thấy rằng có các phương thức **private**, chẳng hạn như **selectWithCriteria**, dường như chỉ liên quan đến các câu lệnh **select**.

Hành vi của phương thức **private** chỉ áp dụng cho một tập con nhỏ của một lớp có thể là một phương pháp heuristic hữu ích để phát hiện các khu vực tiềm năng để cải thiện. Tuy nhiên, động lực chính để thực hiện hành động phải là thay đổi hệ thống. Nếu lớp **Sql** được coi là hoàn chỉnh về mặt logic, thì chúng ta không cần lo lắng về việc tách các trách nhiệm. Nếu chúng ta không cần chức năng **update** trong tương lai gần, thì chúng ta nên để nguyên **Sql**. Nhưng ngay khi chúng ta nhận thấy cần phải "mở" lớp, chúng ta nên xem xét việc sửa chữa thiết kế của mình.

Điều gì sẽ xảy ra nếu chúng ta xem xét một giải pháp như vậy trong Listing 10-10? Mỗi phương thức **public interface** được định nghĩa trong **Sql** trước đó từ Listing 10-9 được cấu trúc lại thành chuyển hóa của lớp **Sql**. Lưu ý rằng các phương thức **private**, chẳng hạn như **valueList**, di chuyển trực tiếp đến nơi cần thiết. Hành vi phổ biến **private** được tách biệt với một cặp lớp tiện ích, **Where** và **ColumnList**.

Listing 10-10

A set of closed classes

```
abstract public class Sql {
    public Sql(String table, Column[] columns);

    abstract public String generate();
}

public class CreateSql extends Sql {
    public CreateSql(String table, Column[] columns);

    @Override
    public String generate();
}

public class SelectSql extends Sql {
    public SelectSql(String table, Column[] columns);

    @Override
    public String generate();
}

public class InsertSql extends Sql {
    public InsertSql(String table, Column[] columns, Object[] fields);

    @Override
    public String generate();

    private String valuesList(Object[] fields, final Column[] columns);
}

public class SelectWithCriteriaSql extends Sql {
```

```
public SelectWithCriteriaSql(  
    String table, Column[] columns, Criteria criteria);  
  
    @Override  
    public String generate();  
}  
  
public class SelectWithMatchSql extends Sql {  
    public SelectWithMatchSql(  
        String table, Column[] columns, Column column, String pattern);  
  
    @Override  
    public String generate();  
}  
  
public class FindByKeySql extends Sql {  
  
    public FindByKeySql(  
        String table, Column[] columns, String keyColumn, String keyValue);  
  
    @Override  
    public String generate();  
}  
  
public class PreparedInsertSql extends Sql {  
    public PreparedInsertSql(String table, Column[] columns);  
  
    @Override  
    public String generate() {  
        private String placeholderList(Column[] columns);  
    }  
}  
  
public class Where {  
    public Where(String criteria);  
  
    public String generate();  
}  
  
public class ColumnList {  
    public ColumnList(Column[] columns);  
  
    public String generate();  
}
```



Mã trong mỗi lớp trở nên cực kỳ đơn giản. Thời gian để hiểu bất kỳ lớp nào giảm xuống gần như là 0. Rủi ro mà một chức năng này có thể gây ảnh hưởng đến một chức năng khác trở nên rất nhỏ. Từ quan điểm kiểm tra, việc chứng minh tất cả các bit logic trong giải pháp này trở thành một nhiệm vụ dễ dàng hơn, vì các lớp đều được tách biệt với nhau.

Quan trọng không kém, khi đã đến lúc thêm các câu lệnh **update**, không có lớp nào trong số các lớp hiện có cần thay đổi! Chúng tôi viết mã logic để xây dựng các câu lệnh **update** trong một lớp con mới của **Sql** có tên là **UpdateSql**. Không có mã nào khác trong hệ thống sẽ bị hỏng do thay đổi này.

Logic **Sql** được cấu trúc lại đại diện cho mọi điều tốt nhất. Nó hỗ trợ SRP. Nó cũng hỗ trợ một nguyên tắc thiết kế lớp OO quan trọng khác được gọi là **Nguyên tắc Đóng/Mở**, hoặc OCP: Các lớp nên **mở** để mở rộng nhưng **đóng** để sửa đổi. Lớp **Sql** được cấu trúc lại của chúng tôi được mở để cho phép chức năng mới thông qua lớp con, nhưng chúng tôi có thể thực hiện thay đổi này trong khi vẫn đóng mọi lớp khác. Chúng tôi chỉ cần tạo lớp **UpdateSql** kế thừa **Sql**.

Chúng tôi muốn cấu trúc hệ thống của mình để chúng tôi sửa đổi ít nhất có thể khi chúng tôi cập nhật các tính năng mới hoặc thay đổi. Trong một hệ thống lý tưởng, chúng tôi kết hợp các tính năng mới bằng cách mở rộng hệ thống, không phải bằng cách thực hiện các sửa đổi đối với mã hiện có.

Tách biệt khỏi sự thay đổi

Nhu cầu sẽ thay đổi, do đó mã sẽ thay đổi. Chúng tôi đã học được trong OO 101 rằng có các lớp cụ thể, chứa các chi tiết triển khai (mã) và các lớp trừu tượng, chỉ đại diện cho các khái niệm. Một lớp khách hàng phụ thuộc vào các chi tiết cụ thể sẽ gặp rủi ro khi các chi tiết đó thay đổi. Chúng tôi có thể giới thiệu các **interfaces** và các lớp **abstract** để giúp cô lập tác động của những chi tiết đó.

Sự phụ thuộc vào các chi tiết cụ thể tạo ra thách thức cho việc kiểm tra hệ thống của chúng tôi. Nếu đang xây dựng một lớp **Portfolio** và nó phụ thuộc vào API **TokyoStockExchange** bên ngoài để lấy giá trị của danh mục đầu tư, thì các trường hợp thử nghiệm sẽ bị ảnh hưởng bởi sự biến động của việc tra cứu như vậy. Thật khó để viết một bài kiểm tra khi cứ năm phút lại nhận được một câu trả lời khác nhau!

Thay vì thiết kế **Portfolio** để nó phụ thuộc trực tiếp vào **TokyoStockExchange**, chúng tôi tạo một **interface**, **StockExchange**, khai báo một phương pháp duy nhất:

```
public interface StockExchange {
    Money currentPrice(String symbol);
}
```

Chúng tôi thiết kế **TokyoStockExchange** để thực hiện **interface** này. Chúng tôi cũng đảm bảo rằng hàm tạo của **Portfolio** lấy tham chiếu **StockExchange** làm đối số:

```
public Portfolio {
    private StockExchange exchange;

    public Portfolio(StockExchange exchange) {
        this.exchange = exchange;
    }
    // ...
}
```

Bây giờ thử nghiệm tạo ra một triển khai có thể kiểm tra được của **interface StockExchange** mô phỏng **TokyoStockExchange**. Việc triển khai thử nghiệm này sẽ cố định giá trị hiện tại cho bất kỳ ký hiệu nào sử dụng trong thử nghiệm. Nếu thử nghiệm cho thấy việc mua 5 cổ phiếu của Microsoft cho danh mục đầu tư của mình, chúng tôi lập mã việc triển khai thử nghiệm để luôn trả lại 100\$ cho mỗi cổ phiếu của Microsoft. Việc triển khai thử nghiệm **interface StockExchange** giảm xuống một bảng tra cứu đơn giản. Sau đó, chúng tôi có thể viết một bài kiểm tra giá trị dự kiến là 500\$ cho danh mục đầu tư tổng thể.

```
public class PortfolioTest {
    private FixedStockExchangeStub exchange;
    private Portfolio portfolio;

    @Before
    protected void setUp() throws Exception {
        exchange = new FixedStockExchangeStub();
        exchange.fix("MSFT", 100);
        portfolio = new Portfolio(exchange);
    }

    @Test
    public void GivenFiveMSFTTotalShouldBe500() throws Exception {
        portfolio.add(5, "MSFT");
        Assert.assertEquals(500, portfolio.value());
    }
}
```

Nếu một hệ thống được chia nhỏ đủ để được kiểm tra theo cách này, nó cũng sẽ linh hoạt hơn và thúc đẩy nhiều lần tái sử dụng hơn. Việc thiếu kết nối có nghĩa là các yếu tố trong hệ thống được tách biệt tốt hơn với nhau và khỏi sự thay đổi. Sự cô lập này giúp bạn dễ dàng hiểu rõ hơn từng phần tử của hệ thống.

Bằng cách giảm thiểu sự ghép nối theo cách này, các lớp của chúng ta tuân thủ một nguyên tắc thiết kế lớp khác được gọi là **Nguyên tắc Đảo ngược Phụ thuộc** (DIP). Về bản chất, DIP nói rằng các lớp của chúng ta nên phụ thuộc vào các yếu tố trừu tượng, không phụ thuộc vào các chi tiết cụ thể.

Thay vì phụ thuộc vào chi tiết triển khai của lớp **TokyoStockExchange**, lớp **Portfolio** của chúng tôi hiện phụ thuộc vào interface **StockExchange**. Interface **StockExchange** đại diện cho khái niệm **abstract** về việc yêu cầu giá hiện tại. Sự trừu tượng này cô lập tất cả các chi tiết cụ thể của việc có được một mức giá như vậy, bao gồm cả việc lấy mức giá đó từ đâu.

Tham khảo

[RDD]: Object Design: Roles, Responsibilities, and Collaborations, Rebecca Wirfs-Brock et al., Addison-Wesley, 2002.

[PPP]: Agile Software Development: Principles, Patterns, and Practices, Robert C. Martin, Prentice Hall, 2002.

[Knuth92]: Literate Programming, Donald E. Knuth, Center for the Study of language and Information, Leland Stanford Junior University, 1992.