ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

KHOA KHOA HỌC MÁY TÍNH

BÁO CÁO SEMINAR GIỮA KỲ

Môn:

NGUYÊN LÝ VÀ PHƯƠNG PHÁP LẬP TRÌNH

Đề tài:

TRÌNH BÀY VỀ SÁCH CLEAN CODE

Giảng viên hướng dẫn: Trần Nguyên Phong

Sinh viên thực hiện: Nguyễn Đức Anh – 15520021

Nguyễn Tuyết Nhi – 15520585

Cao Quốc Đạt – 15580097

Nguyễn Trần Hoàng–15520259

TP Hồ Chí Minh, ngày 17 tháng 10 năm 2017

MỞ ĐẦU

Đối với mỗi người lập trình viên mà nói viết code là một phần không thể thiếu trong cuộc sống của họ. Chính vì thế việc trao dồi kỹ năng lập trình là một việc vô cùng quan trọng. Thế nhưng phải làm sao để phát triển kỹ năng lập trình, viết code. Có rất nhiều cuốn sách, giáo trình giúp ta làm điều ấy. Và cuốn sách sau đây mà nhóm giới thiệu cũng là một cuốn sách giúp nâng cao kỹ năng lập trình, giúp cho người đọc nắm được các kỹ thuật để viết code được trực quan hơn, dễ hiểu hơn. Cuốn sách mà nhóm muốn giới thiệu là "Clean Code" của Robert Cecil Martin, một cuốn sách gối đầu dành cho các lập trình viên.

Báo cáo trình bày nội dung cơ bản của cuốn sách Clean Code. Qua đó tóm tắt lại những ý chính trong cuốn sách và truyền đạt lại một cách ngắn gọn dễ hiểu. Giúp người đọc nắm nhanh các kĩ thuật chính của Clean Code từ đó có những định hướng luyện tập nâng cao kĩ thuật viết code.

Mục lục

1. Clean Code

1.1 Lý Do

Điều đầu tiên lý do mà bạn cần đọc cuốn sách này chính là vì:

1. Bạn là 1 lập trình viên.
2. Bạn muốn trở thành 1 lập trình viên tốt hơn.

Nếu bạn đã và đang bước trên con đường trở thành một lập trình viên thì đây sẽ là một cuốn sách gối đầu của bạn. Clean Code sẽ giúp bạn những kinh nghiệm để có thể viết được code một cách "sạch" nhất có thể.

1.2 Về tác giả

Robert Cecil Martin (Uncle Bob) là một kĩ sư phần mềm từ năm 1970 và là chuyên gia tư vấn phần mềm quốc tế từ năm 1990. Ông từng là chủ tịch đầu tiên của Agile Alliance. Ông là tác giả của 1 số đầu sách Agile Programming, Extreme Programming, UML, Object-Oriented Programming, C++ Programming và Clean Code.

Ông có rất nhiều bài viết, video ở trang 'Code Casts' (một trang web dành cho các chuyên gia phần mềm) và đã xuất bản hàng chục bài viết trong các tạp chí thương mại khác nhau và là diễn giả thường xuyên tại các hội nghị và hội chợ thương mại quốc tế.

Ông cũng là người sáng lập, giám đốc điều hành, và chủ tịch của Uncle Bob Consulting, LLC và Object Mentor Incorporated.

1.3 Bad Code

Thế mã bẩn là gì ? Bạn có thể hiểu đơn giản là code khó đọc và khó hiểu.

Bây giờ hãy tưởng tượng ra là bạn phải viết chương trình hay phần mềm nào đó. Và rồi khi bạn hay một ai đó cần phải đọc lại code của chương trình, cần phải sửa code để sửa lỗi, hay để xây dựng thêm tính năng mới, và điều đó sẽ vô cùng kinh khủng nếu code bạn viết "không được đẹp". Bạn hay người đọc sẽ phải khổ sở để có thể hiểu lại những dòng code mà bạn đã viết, hay phải mất hàng tiếng trời thậm chí cả hàng tá ngày chỉ để sửa một lỗi mà đã có thể sửa được trong

vài giây. Và rồi mọi thứ trở lên rối ren hơn khi phải thêm xóa sửa trong đống code đó, và chúng sẽ dần trở thành một đống rác đúng nghĩa,…

Vậy thì lý do gì mà chúng ta lại viết ra bad code tồi tệ như thế. Lý do lớn đó chính là thời gian. Bạn cần phải hoàn thành thật nhanh cho kịp deadline, cho kịp tiến độ để giao cho sếp, và bạn không quan tâm cũng như không còn thời gian để suy nghĩ hay tối ưu lại code. Hay đơn giản chỉ vì bạn lười, bạn không muốn mất quá nhiều thời gian cho nó, bạn chỉ muốn code xong thật nhanh và rồi không thèm để ý gì tới nó nữa.

Dưới đây là một hình ảnh thể hiện năng suất công việc của một dự án nếu code xấu:

Trục đứng thể hiện hiệu suất công việc, trục ngang thể hiện thời gian. Có thể khi bắt đầu một dự án, hiệu suất công việc được tiến hành rất nhanh chóng, nhưng theo thời gian sau vài lần thay đổi code để sửa chữa hoặc nâng cấp. Lúc này các dòng code bắt đầu trở nên rối rắm và khó sửa chữa, khi ấy hiệu suất công việc giảm dần tiệm cận về 0. Lúc này công việc của chúng ta sẽ bị trì trệ, thậm trí sẽ không thể cứu vãn và đi vào kết thúc…

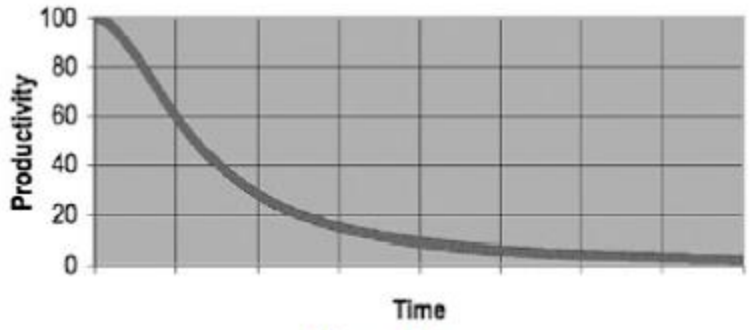

Figure 1-1
Productivity vs. time

Như bạn đã thấy bad code là những thứ vô cùng tệ hại, mà chúng ta là những lập trình viên sẽ không mong muốn gì nó. Vậy thì chúng ta phải làm sao? Chúng ta cần phải viết good code hay code sạch.

1.4 Clean Code (Code Sạch)

- Clean Code là gì?

Dưới đây là một số ý kiến của những nhà lập trình vô cùng tài giỏi:

Bjarne Stroustrup (*Cha đẻ C++*):

"Tôi thích code của tôi thanh lịch và hiệu quả"

"Code sạch là code làm tốt một việc."

- ⇨ Code sạch có nghĩa là phải đẹp, đơn giản, hiệu quả và chỉ nên thực hiện mọi việc từng thứ một.

Grady Booch (*Object Oriented Analysis and Design with Applications*):

"Đơn giản và trực tiếp."

"Code sạch có thể đọc như văn xuôi một cách trôi chảy".

"Không làm lu mờ đi ý định thiết kế, nhưng vẫn đầy đủ các khái niệm trừu tượng hóa rõ ràng (nhanh nhạy quyết đoán và cần thiết) và đơn giản dễ hiểu."

- ⇨ Code sạch viết ra cần phải dễ đọc, dễ hiểu, trực quan, không gây hiểu nhầm.

"Big" Dave Thomas (*Sáng lập OTI, Bố già của Eclipse*): *"Có thể đọc và cải tiến bởi một developer khác không nhất thiết phải là tác giả ban đầu."*

=>Dễ dàng đọc và cải tiến code.

Michael Feathers (*Working Effectively with Legacy Code*): "*Mã sạch là mã được viết ra bởi một người có tâm.*"

Ron Jeffries (*Extreme Programming Installed* và *Extreme Programming Adventures in C#*):

"Giảm thiểu sự trùng lặp, rõ ràng và xây dựng những mô hình trừu tượng hóa đơn giản."

- ⇨ Xây dựng các hàm trừu tượng hóa một số thao tác được thực hiện nhiều lần, nhằm giảm thiểu sự trùng lặp, và dễ dàng sửa chữa hay thay đổi.

Ward Cunningham (*Wiki, Fit, eXtreme Programming,*): *"Bạn có biết bạn vẫn đang làm việc trên code sạch khi mỗi đoạn code mà bạn đọc lại đúng như những gì bạn mong đợi."*

=> Khi code sạch, bạn sẽ đọc nó đúng như những gì bạn mong đợi, đơn giản, dễ hiểu và thuyết phục.

Làm thế nào để viết Code sạch?

Rất dễ để nhận biết thế nào là code sạch hay code bẩn khi nhìn vào nó. Bởi vì code sạch sẽ rất dễ để đọc và hiểu. Nhưng điều đó không đồng nghĩa với việc bạn có thể viết được code sạch.

(Viết code sạch giống như vẽ một bức tranh, hầu hết chúng ta đều có thể nhận ra được tranh vẽ đẹp hay vẽ xấu, nhưng điều đó không đồng nghĩa chúng ta có thể vẽ được).

Để viết được mã sạch bạn phải có code-sense của cleanliness (*cảm giác mã về sự sạch sẽ*), và để được như vậy bạn có thể luyện tập thông qua quá trình áp dụng hàng nghìn kỹ thuật nhỏ vào việc viết code. Khi có được kinh nghiệm bạn có thể nhận biết được đâu là code sạch đâu là code bẩn, và có thể biến code bẩn thành code sạch, biết nên thay đổi code ra sao là phù hợp nhất, đẹp đẽ nhất.

2. Meaning Name (Đặt tên có ý nghĩa)

2.1 Use Intention-Revealing Names (Sử dụng tên có ý nghĩa)

Một cách dễ hiểu là ta cần chọn tên có ý nghĩa. Cái tên được đặt sẽ nói cho bạn biết nó được sử dụng để làm gì, sử dụng ra sao. Phải chọn tên để dễ hiểu và dễ dàng cho việc thay đổi code sau này. Đừng coi thường việc đặt tên biến, nó là cả một nghệ thuật đấy nếu không chịu khó đầu tư tên biến ngay từ đầu, bạn sẽ mất hàng tá thời gian để đọc hiểu code sau này.

Ví dụ:

1. ind d // không có ý nghĩa
   int dayofmonth //số ngày trong tháng
2.

```
public List<int[]> getThem() {              // getThem là lấy gì
  List<int[]> list1 = new ArrayList<int[]>();//List1 làm gì
    for (int[] x : theList)
      if (x[0] == 4)                        // tại sao lại so sánh với 4
        list1.add(x);
  return list1;
}
```

```
public List<Cell> getFlaggedCells() {      //Rõ ràng mục đích
    List<Cell> flaggedCells = new ArrayList<Cell>(); //tạo list lưu flag
   for (Cell cell : gameBoard)
     if (cell.isFlagged())                    //Kiểm tra flag
       flaggedCells.add(cell);
 return flaggedCells;
}
```

2.2 Avoid Disinformation (Tránh gây hiểu lầm)

Tránh đặt tên những từ dễ gây hiểu lầm ý nghĩa của nó.

Ví dụ:

- hp, aix, and sco là những tên được sử dụng trong Unix platform hay biến.
- Cẩn thận với các tên gần giống nhau như dưới vì rất khó để phân biệt:

  XYZControllerForEfficientHandlingOfStringsvà XYZControllerForEfficientStorage OfStrings?

- Các từ l và o rất dễ nhầm lẫn với số 1 và 0.

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

2.3 Make Meaningful Distinctions (Làm cho sự riêng biệt trở nên rõ ràng)

Các lập trình viên thường tự tạo cho mình một vấn đề khi mà học viết code chỉ để cho trình biên dịch có thể chạy được. Ví dụ: Họ cần đặt tên cho 2 biến khác nhau và họ chỉ đơn giản tùy ý chọn đại các tên biến như a1, a2… điều này sẽ gây khó khăn để mà nhận ra sự khác biệt giữa chúng.

Vì vậy, nếu bạn phải đặt một tên biến, bạn nên lựa chọn tên dựa theo sự khác biệt của 2 biến đó:

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

Hàm trên sẽ dễ đọc hơn khi ta thay biến a1, a2 thành source(nguồn) và destination(đích).

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

Có phải dễ hiểu hơn không nào, copy chuỗi từ biến nguồn sang biến đích thay vì các biến không rõ nghĩa như a1, a2.

Use Pronounceable Names (Sử dụng tên đọc được)

Những cái tên khó đọc hay khó phát âm sẽ gây khó khăn cho việc đọc code, ghi nhớ hay thảo luận.

```
class DtaRcrd102 {

  private Date genymdhms;

  private Date modymdhms;

  private final String pszqint = "102";

  /* ... */

};
```

Ví dụ trên các biến genymdhms, modymdhms, pszqint tối nghĩa và khó phát âm mặc dù mục đích của người viết cũng có (genymdhms viết tắt của generation date, year, month, day, hour, minute, and second).

```
class Customer {

  private Date generationTimestamp;
```

```
  private Date modificationTimestamp;

  private final String recordId = "102";

  /* ... */

};
```

Và khi code được đặt tên có phát âm rõ ràng thì người đọc nó sẽ dễ dàng đọc hiểu được nội dung hơn.

2.4 Use Searchable Names (Sử dụng tên tìm kiếm được)

Các tên biến và hằng khó mà phát hiện ra vị trí của nó khi cần tìm kiếm. Vì vậy ta nên đặt tên cho chúng nó để dễ dàng tìm kiếm khi cần.

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

Các biến và hằng đã được đặt tên theo ý nghĩa của chúng, khi ta cần tìm lại các biến này để thay đổi, ta chỉ cần nhớ tới tên của chúng.

```
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;
}
```

2.5 Member Prefixes (Thành phần tiền tố)

Hãy tránh phải đặt tên với các tiền tố ví dụ (m_,s_,..). sử dụng chúng sẽ làm các đoạn code của bạn trở lên rối rắm. Và việc bỏ đi các tiền tố thì tên biến cũng sẽ có ý nghĩa hơn.

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

Sau khi bỏ tiền tố m_ và thay đổi tên biến rõ ràng hơn

```
public class Part {
  String description;
  void setDescription(String description) {
    this.description = description;
  }
}
```

2.6 Hungarian Notation (Ký hiệu người hungary)

Loại bỏ kiểu dữ liệu ghi thêm vào tên biến, điều này nên tránh vì nó khá dư thừa

```
PhoneNumber phoneString;

//Bỏ String đi không ảnh hưởng gì.

PhoneNumber phone;
```

2.7 Avoid Mental Mapping (Tránh ánh xạ tinh thần)

Các vòng lặp có thể sử dụng các biến i, j, k nếu phạm vi nhỏ và không có tên nào trùng với nó.

```
for (I = 0; i < 10; i++)
for (j = 0; j < 10; j++)
```

2.7.1 *Class Names*

Tên Classes và Objects không nên là một động từ, nó nên là danh từ hoặc cụm danh từ như là Customer, WikiPage, Account, và AddressParser. Và tránh những từ như *Manager, Processor, Data*, or *Info*.

Ví dụ nữa như tên Address, thì các tên sau là dư thừa AccountAddress, ClientAddress, MACAddress. Tốt hơn hết nên để Address vì tên class đã thể hiện ý nghĩa của nó rồi Account.Address, Client.Address, hay MACAddress đơn giản hơn chỉ cần ghi MAC là đủ.

2.7.2 *Method Names (Tên Phương Thức)*

Tên của phương thức nên là một động từ hoặc cụm động từ. Một phương thức có nghĩa là làm một cái gì đó. Các phương thức truy xuất, thiết lập, điều kiện nên thêm các tiền tố như *get, set, is*.

```
string name = employee.getName(); //Lấy tên

customer.setName("mike"); // Đặt tên

if (paycheck.isPosted())... //Kiểm tra điều kiện
```

2.8 Pick One Word per Concept (Chọn một từ cho mỗi khái niệm)

Các phương thức có cùng một khái niêm trừu tượng hay đặc tính ta được chọn một tên cho các phương thức đó và có thể sử dụng trong các class khác nhau. Bạn sẽ không phải mất nhiều thời gian để tìm kiếm cũng như nhớ tên nào đã sử dụng.

```
fetch, retrieve, get // as equivalent methods
controller, manager, driver // confusing
```

2.9 Don't Pun (Không chơi chữ)

Không sử dụng một từ cho hai mục đích.

Ví dụ:

Nếu phương thức add được dùng để thêm hay nối phần tử mà bạn dùng cho phương thức thêm phần tử vào một tập hợp (insert) thì đây là chơi chữ.

Bạn nên tránh nhầm lẫn với khái niệm.

2.10 Use Solution Domain Names (Dùng tên của miền giải pháp)

Những người đọc code của bạn cũng là những lập trình viên nên bạn đừng lo sợ họ không hiểu hãy cứ dùng những từ ngữ khoa học máy tính, toán học, thuật toán, các tên kiểu mẫu,...

Ví dụ: AccountVisitor, JobQueue…

2.11 Add Meaningful Context (Thêm bối cảnh có ý nghĩa)

Có những tên mà bạn có thể hiểu nó theo nghĩa này nếu ở bối cảnh này hay nghĩa khác với bối cảnh khác. Và trong lập trình cũng vậy những cái tên như thế nên được đặt vào những bối cảnh là class, function, namespace. Hoặc bạn có thể thêm tiền tố cần thiết vào tên như một lựa chọn cuối cùng.

Một ví dụ nhỏ, bạn có các tên biến: firstName, lastName, street, houseNumber, city, state và zipcode. Khi đặt chúng ở cạnh nhau thì rõ ràng chúng sẽ tạo thành một địa chỉ. Nhưng nếu biến state được đặt một mình thì bạn có thể hiểu ra nó là một phần của địa chỉ không? Bạn có thể thêm bối cảnh cho nó bằng cách sử dụng tiền tố: addrFirstName, addrLastName, adddrState,.. Ít nhất người đọc sẽ hiểu được những biến này là một phần của một cấu trúc lớn hơn nào đó. Và tất nhiên, sẽ tốt hơn là nếu tạo ra một lớp có tên là Address. Lúc này thì cả trình biên dịch cũng hiểu là những biến này thuộc một cấu trúc lớn hơn là Address.

```
firstName, lasName, street, city, state,Zipcode

// giải pháp tốt hơn

addrFirstName, addrLastName, adddrState,..

//giải pháp tốt hơn nữa

Class Address
```

2.12 Don't Add Gratuitous Context (Đừng thêm những bối cảnh vô căn cứ)

Tên ngắn sẽ tốt hơn là tên dài nếu chúng rõ ràng. Khi ấy việc thêm bối cảnh vào tên là không cần thiết.

Tên accountAddress và customerAddress sẽ là một tên tốt cho các thể hiện của lớp Address nhưng sẽ không cần thiết nếu nó là tên lớp. Nếu bạn cần sự khác nhau giữa MAC addresses, port addresses, and Web addresses ta có thể thay bằng PostalAddress, MAC, và URI.

3. Function (Hàm)

Function là một đơn vị để tạo nên bất kỳ chương trình nào. Bạn đã từng phải đau đầu để hiểu những hàm chức năng dài lê thê với nhiều hàm if else, các biến chuỗi rối ren, hay những chức năng khác nhau được lồng chung vào một hàm. Và để đọc được các hàm như thế thì quả thật là tệ hại. Tuy nhiên chúng ta có thể thay đổi hàm một cách dễ hiểu hơn chỉ với một số phương pháp rút gọn đơn giản, hay thay đổi tên, tái cấu trúc hàm.

Vậy làm thế nào để làm được như thế hãy cùng tìm hiểu nào?

3.1 Small!

"Nguyên tắc đầu tiên của hàm là cần phải nhỏ. Nguyên tắc thứ hai là hàm cần phải nhỏ hơn nữa."

Đây không phải là một kết quả nghiên cứu hay một lời khẳng định chính xác hoàn toàn rằng các hàm được viết ngắn hơn sẽ tốt hơn cả. Nhưng đây là những kinh nghiệm hơn 40 năm lập trình về việc viết hàm với nhiều kích thước khác nhau của tác giả khi viết các hàm 3000 dòng, 100-300 dòng, hay là những hàm với 20-30 dòng. Kinh nghiệm từ những sai lầm chỉ ra rằng, các hàm nên được viết rất nhỏ.

Và tác giả chỉ ra rằng các hàm không nên dài quá 20 dòng và một dòng không nên quá 150 ký tự.

Dưới đây là một ví dụ cho kích thước của một hàm.

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

- Blocks and Indenting:

Những khối lệnh *if, else, while* nên được viết trong một dòng, và những dòng này nên được đặt thành một lời gọi hàm. Việc này không những giúp cho các hàm nhỏ mà còn giúp bạn dễ dàng hiểu chức năng của nó thông qua tên hàm.

Điều này sẽ giúp các hàm không bị lồng nhau. Vì thế các mức thụt lề của dòng lệnh không lớn quá một hoặc hai. Và các hàm dễ đọc và dễ hiểu hơn.

3.2 Do One Thing

"Hàm chỉ nên thực hiện một thứ. Chúng nên thực hiện việc đó cho tốt. Nó nên thực hiện một việc duy nhất."

Vấn đề đặt ra là "one thing" (một thứ) đó là gì?

Ta sẽ lấy ví dụ ở trên của hàm renderPageWithSetupsAndTeardowns, sẽ có ba "thứ" cần phải làm:

- isTestPage(pageData) // Kiểm tra có phải là testPage
- includeSetupAndTeardownPages(pageData, isSuite); //Nếu phải thì thực hiện setup và Teardown Pages
- pageData.getHtml(); //Chuyển trang đó sang html

Ở ví dụ trên bạn sẽ khó hiểu là tại sau lại có tới ba "thứ" cần phải làm ở trên. Điều này bạn cần phải phân biệt rõ rõ one thing với multi steps. Một hàm không chỉ gồm một bước và mà gồm nhiều bước. Mỗi bước có thể là một hàm nào khác. Mục đích của chúng ta là phân rã chúng ra nhiều phần để xử lý thành từng hàm nhỏ nhất. Hay đơn giản hơn bạn có thể hiểu hàm là tập hợp các hàm khác nhỏ hơn để xử lý từng tác vụ.

3.3 One Level of Abstraction per Function

Để chắc rằng hàm của chúng ta thực hiện duy nhất "một thứ" ta cần đảm bảo các câu lệnh trong hàm đều phải cùng một mức độ trừu tượng.

Việc sử dụng lẫn lộn các mức độ trừu tượng ở trong hàm sẽ làm cho nó khó hiểu. Người khác đọc code sẽ khó để hiểu được cấu trúc hàm, cũng như những khái niệm quan trọng trong hàm.

Ví dụ như hàm getHtml()

```
// cấp độ trừu tượng cấp cao
getHtml()
// cấp độ trừu tượng trung cấp
```

```
        String pagePathName  = PathParser.render(pagePath);
        // cấp độ trừu tượng thấp
        .append("\n")
```

3.4 Reading Code from Top to Bottom: The Stepdown Rule

Chắc chắn rằng chúng ta đều muốn đọc code từ trên xuống một cách tự nhiên. Mỗi hàm đều được sắp xếp theo mức độ trừu tượng từ trên xuống và giảm dần mức độ trừu tượng từ trên xuống khi đọc danh sách các hàm.Và tác giả gọi đây là *Nguyên tắc Stepdown*.

Sẽ khó khăn để học và viết hàm ở một mức độ cấp trừu tượng. Nhưng học thủ thuật này rất quan trọng. Nó giữ cho hàm ngắn và luôn thỏa mãn "one thing" (một thứ).

3.5 Switch Statements (Câu lệnh switch)

Thật khó để rút gọn câu lệnh *switch*. Chúng ta không thể nào luôn luôn tránh câu lệnh *switch*, những chúng ta chắc chắn rằng mỗi câu lệnh *switch* luôn cất giữ bởi một class ở mức thấp hơn và không bao giờ được sử dụng lại. Chúng ta làm được, bằng đa hình.

```
public Money calculatePay(Employee e)
throws InvalidEmployeeType {
  switch (e.type) {
    case COMMISSIONED:
      return calculateCommissionedPay(e);
    case HOURLY:
      return calculateHourlyPay(e);
    case SALARIED:
      return calculateSalariedPay(e);
    default:
      throw new InvalidEmployeeType(e.type);
  }
```

```
}
```

Có một số vấn đề với function này.

1. Nó quá lớn, khi thêm loại của Employee nó sẽ lớn hơn nữa => nhiều hàm trong switch có cấu trúc tương tự được tạo ra.
2. Thứ hai, nó rõ ràng là làm nhiều hơn "one thing" (một thứ).
3. Thứ ba, xâm phạm nguyên tắc Single Responsibility Principle bởi vì có nhiều hơn một lý do để nó thay đổi.
4. Thứ tư, xâm phạm nguyên tắc Open Close Principle bởi vì nó phải thay đổi bất cứ khi nào có kiểu Employee mới được thêm vào.

Cách giải quyết ở đâu là hãy sử dụng đa hình cho các switch để tạo ra các trường thích hợp của Employee mới được thêm vào.

```
class Employee...
    int payAmount() {
        switch (getType()) {
            case EmployeeType.ENGINEER:
                return _monthlySalary;
            case EmployeeType.SALESMAN:
                return _monthlySalary + _commission;
            case EmployeeType.MANAGER:
                return _monthlySalary + _bonus;
            default:
                throw new Exception("Incorrect Employee");
        }
        }

class EmployeeType...
    abstract int payAmount(Employee emp);
class Salesman...
    int payAmount(Employee emp) {
        return emp.getMonthlySalary() + emp.getCommission();
```

```
        }
class Manager...
    int payAmount(Employee  emp) {
        return emp.getMonthlySalary()  + emp.getBonus();
}
```

3.6 Use Descriptive Names (Sử dụng tên mô tả)

Sử dụng tên mô tả những gì mà hàm làm. Đừng ngại bởi viết một cái tên dài. Tên mô tả dài thì sẽ tốt hơn là một tên ngắn mà bí ẩn và cũng tốt hơn là dùng một bình luận code. Quy tắc là dùng những từ ngữ dễ đọc để làm tên hàm và mô tả được chức năng của hàm đó.

Đừng ngại tốn thời gian cho việc đặt tên. Bạn thật nên thử nhiều tên và đọc code ở mỗi vị trí có nó.

Như ví dụ hàm testableHtml thay bằng includeSetupAndTeardownPages

Sử dụng tên mô tả sẽ giúp bạn làm rõ các thiết kế các module trong suy nghĩ bạn và giúp bạn cải thiện nó.

3.7 Function Arguments

Số lượng tham số trong một hàm nhiều nhất là 2 tham số đầu vào, lý tưởng nhất là 0. Ba tham số thì cần nên tránh. Và nhiều hơn ba đòi hỏi phải có biện luận rất đặc biệt và không nên sử dụng tùy tiện.

Đối với tham số đầu ra khó hiểu hơn là tham số đầu vào. Một hàm sau khi thực hiện các thao tác trên tham số đầu vào sẽ cho ta tham số đầu ra, và tham số này khác với tham số đầu vào.

3.8 Common Monadic Forms

Có hai cách thức phổ biến để chuyền một tham số vào hàm là:

Một hàm lý luận kiểm tra tham số, trả về true hoặc false.

```
boolean fileExists("MyFile")
```

Một hàm sử dụng tham số, biến đổi nó thành một cái gì đó và trả lại nó.

```
InputStream  fileOpen("MyFile")
```

Một cách thức khác ít phổ biến sử dụng cho một hàm sự kiện có một tham số đầu vào nhưng không có tham số đầu ra.

```
void passwordAttemptFailedNtimes(int attempts)
```

Cố gắng tránh bất cứ Monadic Functions nào mà không theo những cách thức trên.

Ví dụ:

void includeSetupPageInto(StringBuffer pageText)

Nếu một hàm muốn biến đổi tham số thành một tham số đầu ra, thì hàm đó nên trả về giá trị.

```
StringBuffer transform(StringBuffer in)

//thì tốt hơn

void transform-(StringBuffer out)
```

- *Flag Arguments*

Một tham số đầu vào là flag (true, false) là một điều tồi tệ, vì nó sẽ làm cho hàm trở lên phức tạp và làm nhiều hơn "một thứ": một là khi flag có giá trị đúng, và hai khác khi flag có giá trị sai. Chúng ta nên tránh bằng cách tách trường hợp này thành 2 hàm.

```
render(true)

//tách thành 2 hàm xử lý đơn

renderForSuite()
renderForSingleTest()
```

3.9 Dyadic Functions

Hàm một tham số dễ hiểu hơn là hàm có hai tham số.

```
writeField(name) // dễ hiểu hơn
writeField(outputStream, name)
```

Có những trường hợp 2 tham số là hoàn toàn chấp nhận được. Như ví dụ dưới một Point (Điểm) sẽ phải có 2 giá trị, sẽ ra sao nếu một Point (Điểm) chỉ có một giá trị.

```
Point p = new Point(0,0);
```

Hàm bên dưới dù rõ ràng như thế nhưng vẫn có vấn đề. Bởi tham số expected (dự kiến) và actual (thực tế) không theo trật tự tự nhiên.

```
assertEquals (expected, actual)
```

=> *Hai tham số phải có sự gắn kết hoặc được sắp xếp theo một trật tự tự nhiên.*

3.10 Triads

Những hàm có 3 tham số sẽ càng khó hiểu hơn hàm 2 tham số. Các vấn đề về trong việc sử dụng sẽ nhiều hơn. Bạn nên thật sự cẩn thận trước khi tạo ra một hàm ba tham số.

Ví dụ:

```
assertEquals (message, expected, actual)
```

Tuy nhiên ta vẫn cần sử dụng hàm ba tham số để thể hiện ý nghĩa hàm.

```
assertEquals (1.0, amount, .001)// Hợp lý
```

3.11 Argument Objects

Khi có nhiều hơn 2 hay 3 tham số thì ta nên bao chúng lại một lớp riêng của chúng. Giảm số lượng các tham số bằng cách tạo ra các đối tượng cho nó.

```
Circle makeCircle (double x, double y, double radius);

Circle makeCircle (Point center, double radius);
```

3.12 Argument Lists

Đôi khi chúng ta muốn truyền một số lượng biến số của tham số vào một hàm.

```
String.format("%s worked %.2f hours.", name, hours);
```

Nếu các tham số đều xử lý như nhau, ta có thể gom chúng lại thành một tham số kiểu List.

3.13 Verbs and Keywords

Chọn một cái tên tốt sẽ dễ dàng để giải thích mục đích hàm, thứ tự và mục đích của tham số.

Trong trường hợp là một hàm đơn nguyên, hàm và tham số nên tạo thành một động từ hoặc một cặp danh từ đẹp. Ví dụ là write (name) sẽ viết lại là writeField (name)

Dạng từ khóa của tên hàm, chúng ta sẽ mã hóa tên của các tham số vào tên hàm. Ví dụ: *assertEquals* có thể được viết lại là *assertExpectedEqualsActual (expected, actual)*, điều này làm giảm đáng kể vấn đề nhớ thứ tự các tham số.

3.14 Have No Side Effects (Không có tác dụng phụ)

Hàm bạn viết hứa hẹn chỉ làm "một thứ", nhưng không may nó cũng sẽ thực hiện một vài "thứ" ẩn nào đó. Nó có thể làm thay đổi giá trị biến trong lớp của nó. Đôi khi nó thay đổi thông số vào hàm hay vào hệ thống toàn cục. Trong cả hai trường hợp chúng sẽ gây ảnh hưởng tai hại đến chương trình.

Ví dụ:

```
public boolean set (String attribute, String value);
```

- ⇨ Hàm này có tác dụng set giá trị của tham số attribute là value. Thành công sẽ trả về true.
- ⇨ Thế nhưng hàm set này dùng để thay đổi giá trị sao lại trả về true false
- ⇨ Một là làm một thứ gì đó, hoặc trả lời true false, nhưng ko được làm cả hai.

Ta có thể thay đổi tên hàm thành setAndCheckIfExists nhưng như vậy cũng chả tốt hơn.

Tốt hơn hết ta nên tách thành hai phần riêng biệt

```
if (attributeExists("username")) {
setAttribute("username", "unclebob");
...
}
```

3.15 Don't Repeat Yourself

Đừng lặp lại đoạn code của bạn đây là một lời khuyên chân thành. Bạn hãy tưởng tượng những đoạn code trùng lặp được viết lại nhiều lần trong một chương trình và rồi khi bạn cần sửa chữa những đoạn code đó thì bạn phải sửa chính xác tất cả các đoạn code được lặp lại trong chương trình. Một sự sai sót nhỏ cũng gây ảnh hưởng tới kết quả của chương trình.

Đã có các nguyên tắc được xây dựng nhằm kiểm soát và hạn chế sự trùng lặp đó như: Lập trình hướng đối tượng, lập trình cấu trúc,...

3.16 Structured Programming

Nguyên tắc Dijkstra trong lập trình cấu trúc: Mọi hàm và mọi block ở trong một hàm nên có một đầu vào và một đầu ra. Điều này có nghĩa là chỉ nên có một câu lệnh trả về trong hàm, không có break hay continue trong một vòng lặp, và không bao giờ có bất cứ câu lệnh *goto* nào.

Điều này có thể mang lại lợi ích ít khi hàm rất nhỏ. Và chỉ khi hàm lớn thì các nguyên tắc này mới đem lại lợi ích đáng kể.

Vì vậy nên hàm của bạn nhỏ, các câu lệnh return, break, hay continue sẽ không gây tổn hại gì, một số còn có lợi hơn là tuân theo nguyên tắc trên. Mặt khác, goto chỉ có ý nghĩa trong các hàm lớn, vì vậy cần tránh.

4. Formatting

Khi người khác muốn tìm hiểu chi tiết một chương trình, chúng ta muốn họ được chú ý bởi sự ngăn nắp, súc tích và không bị phân tâm khi đang đề cập đến một chi tiết nào đó thay vì khiến họ bối rối bởi những dòng code hỗn độn.

Chính vì thế chúng ta cần quan tâm đến việc tổ chức các dòng code tuân thủ theo luật một khuôn thức chung có một trật tự nào đó khi làm dự án với nhóm hoặc cá nhân.

Mục tiêu:
Viết các dòng lệnh theo một khuôn thức chung thực sự quan trọng vì nó là sự giao tiếp mà một nhà phát triển luôn đặt lên hàng đầu trong công việc.

Nếu các bạn nghĩ rằng với một nhà lập trình viên chuyên nghiệp, thực thi thành công các dòng lệnh là quan trọng nhất thì bạn đã sai. Mục đích của việc lập trình theo luật một khuôn thức chung giúp chúng ta có thể dễ dàng đọc hiểu từ đó việc thay đổi, bổ sung vào chương trình cho những bản cập nhật sau này sẽ trở nên tiện lợi hơn. Vì thế phong cách lập trình hay nói cách khác là tuân thủ viết các dòng lệnh theo một khuôn thức chung giúp ta dễ dàng đọc và hình dung là yếu tố quan trọng và mẫu mực trong việc phát triển và duy trì chương trình.

4.1 Vertical Formatting

Chúng ta nên tách chương trình hoặc hệ thống thành các file có kích thước nhỏ, có ít các dòng lệnh vì nó giúp chúng ta đọc các dòng lệnh dễ dàng hơn so với việc gộp hàng nghìn dòng lệnh vào một file.

"Small files are usually easier to understand than large files are."

4.1.1 The Newspaper Metaphor

Khi đọc báo, chúng ta đọc từ trên xuống, phần tiêu đề sẽ nói lên phần nội dung bài viết và đoạn văn bản đầu tiên sẽ tóm tắt văn bản theo sau nó. Mã nguồn cũng tương tự như vậy, khi đặt tên mã nguồn, chúng ta nên đặt tên đơn giản và dễ hiểu để xác định phần nội dung chúng ta sẽ đọc

có phải là phần chúng ta đang cần hay không. Vị trí đầu tiên nhất sẽ cung cấp cho chúng ta thông tin về thuật toán và các thuật ngữ để chúng ta có cái nhìn tổng quát và độ chi tiết tăng dần khi ta càng đọc đến cuối file.

4.1.2 Vertical openness between concepts (phân tách giữa các khái niệm)

Gần như tất cả các dòng lệnh đều được đọc từ trái qua phải và từ trên xuống dưới. Mỗi dòng biểu thị một biểu thức hoặc một mệnh đề và mỗi nhóm các dòng lệnh thể hiện một lối suy nghĩ hoàn chỉnh. Chính vì thế chúng nên được phân cách ra với nhau bởi một dòng trắng.

Hãy xem hình 5.1, các dòng trắng phân tách các định nghĩa, các hàm. Đây được xem như là luật cơ bản ảnh hưởng sâu sắc đến cách trình bày các dòng code khi lập trình.

Listing 5-1
BoldWidget.java

```
package fitnesse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL
  );

  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));
  }

  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

Hình 5.2, đây là một ví dụ cho thấy sự rối mắt khi các hàm, các biểu thức, … được viết luôn tuồng, không phân cách.

Listing 5-2
BoldWidget.java

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL);
  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));}
  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

4.1.3 Vertical density (mật độ theo chiều dọc)

Nếu như các dòng trắng phân tách các concepts giúp cho chúng ta đọc các dòng lệnh dễ dàng hơn thì thuật ngữ vertical density (mật độ theo chiều dọc) nhấn mạnh đến sự gắn kết chặt chẽ các dòng code có liên quan đến nhau.

hình 5.3 cho thấy các dòng comments vô ích làm cho tính liên kết chặt chẽ của 2 biến bị phá vỡ.

Listing 5-3

```
public class ReporterConfig {

  /**
   * The class name of the reporter listener
   */
  private String m_className;

  /**
   * The properties of the reporter listener
   */
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);
  }
```

hình 5.4 rõ ràng dễ đọc hơn, ta dễ dàng nhận ra một lớp với hai biến và một method

Listing 5-4

```
public class ReporterConfig {
  private String m_className;
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);
  }
```

4.1.4 Vertical distance

Đã bao giờ các bạn cuộn lên và cuộn xuống mã nguồn để xem các hàm có quan hệ với nhau như thế nào và bị thất lạc trong mớ hỗn độn của các dòng code? Điều này thực sự khó chịu vì khi bạn đang muốn tìm hiểu chức năng của hệ thống lại bị tốn nhiều thời gian để ghi nhớ vị trí các phần của một chương trình trong mã nguồn. Chính vì thế, các concepts có liên quan mật thiết với nhau nên được đặt gần nhau. Rõ ràng là luật này không có tác dụng với các concepts nằm ở các file khác nhau nhưng những concepts có quan hệ mật thiết nên được để chung ở một file trừ khi chúng ta có lí do thuyết phục để đặt chúng ở những file khác nhau.

4.1.5 Variable Declarations (khai báo biến)

Các biến nên được khai báo gần với cách sử dụng gần nhất có thể bởi vì các hàm của chúng ta rất ngắn, các biến cục bộ nên nằm phía trên cùng của các hàm như trong hình dưới.

```
private static void readPreferences() {
  InputStream is= null;
  try {
    is= new FileInputStream(getPreferencesFile());
    setPreferences(new Properties(getPreferences()));
    getPreferences().load(is);
  } catch (IOException e) {
    try {
      if (is != null)
        is.close();
    } catch (IOException e1) {
    }
  }
}
```

4.1.6 Instance variables (biến đối tượng)

Mặt khác đối tượng nên được khai báo ở phần đầu của class, bởi vì trong một class tổ chức tốt, các biến đối tượng được dùng rất nhiều từ các phương thức của lớp.

4.1.7 Dependent functions (hàm phụ thuộc)

Nếu một hàm gọi một hàm khác, thì chúng nên được đặt gần nhau và hàm gọi nên nằm bên trên hàm được gọi, nó giúp code của chúng ta được mạch lạc và tự nhiên hơn.

4.1.8 Conceptual affinity (mối quan hệ khái niệm)

Các mối quan hệ dựa trên sự phụ thuộc, ví dụ như một hàm gọi một hàm khác hoặc một hàm sử dụng một biến nào đó. Tuy nhiên, còn một vài những nhân tố khác xây dựng nên các mối liên hệ, ví dụ như những hàm có toán tử gần giống nhau như hình dưới.

```
public class Assert {
  static public void assertTrue(String message, boolean condition) {
    if (!condition)
      fail(message);
  }

  static public void assertTrue(boolean condition) {
    assertTrue(null, condition);
  }

  static public void assertFalse(String message, boolean condition) {
    assertTrue(message, !condition);
  }

  static public void assertFalse(boolean condition) {
    assertFalse(null, condition);
  }
...
```

4.1.9 Vertical ordering (sắp xếp theo chiều dọc)

Thông thường chúng ta muốn rằng các hàm được gọi nằm bên dưới hàm gọi đến nó, điều này làm cho các phần trong mã nguồn mạch lạc, liên tục hơn từ cấp cao đến cấp thấp. Tương tự như một tờ báo, chúng ta muốn những concept quan trọng nhất đặt đầu tiên và những chi tiết cấp thấp đặt ở cuối. Điều này cho phép ta đọc lướt qua mã nguồn, lấy ý chính thay vì bị những chi tiết làm phân tâm.

4.2 Horizontal Formatting (theo chiều ngang)

Chúng ta hãy giữ cho các dòng lệnh được ngắn, tránh dòng lệnh quá dài khiến chúng ta phải cuộn trang sang phải.

4.2.1 horizontal openness and density

Chúng ta sử dụng khoảng trắng theo chiều ngang để liên kết những thứ có liên quan chặt chẽ với nhau và cũng như làm giảm bớt sự liên quan lẫn nhau cho những thứ không liên quan mật thiết với nhau.

Cách thức comment một đoạn dài như thế nào để không gây rối mắt hoặc gián đoạn suy nghĩ khi làm việc với code

4.2.2 horizontal alignment (dóng hàng)

```
public class FitNesseExpediter implements ResponseSender
{
  private   Socket          socket;
  private   InputStream     input;
  private   OutputStream    output;
  private   Request         request;
  private   Response        response;
  private   FitNesseContext context;
  protected long            requestParsingTimeLimit;
  private   long            requestProgress;
  private   long            requestParsingDeadline;
  private   boolean         hasError;

  public FitNesseExpediter(Socket          s,
                           FitNesseContext context) throws Exception
  {
    this.context =            context;
    socket =                  s;
    input =                   s.getInputStream();
    output =                  s.getOutputStream();
    requestParsingTimeLimit = 10000;
  }
```

Dóng hàng không thực sự cần thiết. Việc này nhấn mạnh đến những yếu tố không cần thiết khiến chúng ta lệch đi khỏi mục đích thật sự.

Như ví dụ trên, chúng ta dường như đọc các dòng lệnh theo từng cột. Chính vì thế, dóng hàng và dóng phép gán thực sự không cần thiết.

4.2.3 Indentation (thụt lề)

Mã nguồn là một sự phân cấp các thứ bậc. Để khiến cho mã nguồn dễ đọc hơn hay nói đúng hơn là nhìn rõ hơn sự phân cấp, ta sử dụng cách thụt đầu dòng cho các dòng lệnh. Ví dụ như các phương thức trong cùng một lớp sẽ thụt đầu dòng về phải một mức, các câu lệnh trong một phương thức lại phải thụt đầu dòng về bên phải một mức so với dóng khai báo phương thức, … Nếu như viết dòng lệnh mà không thụt đầu dòng thì sẽ thực sự là khó đọc như hình dưới.

```
public class FitNesseServer implements SocketServer { private FitNesseContext
context; public FitNesseServer(FitNesseContext context) { this.context =
context; } public void serve(Socket s) { serve(s, 10000); } public void
serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new
FitNesseExpediter(s, context);
sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); }
catch(Exception e) { e.printStackTrace(); } } }

-----


public class FitNesseServer implements SocketServer {
  private FitNesseContext context;
```

4.3 Team Rules

Các thành viên trong nhóm nên tuân thủ theo một số quy tắt định dạng của nhóm để tạo nên tính nhất quán.

5. Object and Data Structures

5.1 Data Abstraction (dữ liệu trừu tượng)

Ẩn sự thực thi (hiding implementation) là sự trừu tượng. Một lớp không chỉ đơn giản là dùng các phương thức get và set để truy cập các biến của nó mà hơn nữa nó cho thấy giao diện trừu tượng cho phép người dùng thao tác với bản chất của dữ liệu mà không cần biết nó được thực thi như thế nào.

Data/Object Anti-Symmetry (dữ liệu/ đối tượng bất đối xứng).

Đối tượng ẩn dữ liệu sau sự trừu tượng và trình bày các hàm thực hiện tính toán trên dữ liệu. Cấu trúc dữ liệu trình bày dữ liệu và không có các hàm ý nghĩa.

"Objects hide their data behind abstractions and expose functions that operate on that data. Data structure expose their data and have no meaningful functions."

5.2 The law of Demeter

Luật Demeter nói rằng một module không nên biết nhiều về đối tượng mà nó đang thao tác trên đó.

Phương thức f của class C chỉ được gọi:

- Những phương thức của C
- Một đối tượng được tạo ra bởi f
- Mội đối tượng được truyền vào f như là một đối số
- Một đối tượng giữ một biến thực thể của C

phương thức không nên dẫn ra những phương thức trên đối tượng được trả về bởi các hàm hợp lệ. Hay một cách khác "trò chuyện với bạn, đừng nói với người lạ."

Đoạn code sau được xem như là đã vi phạm luật Demeter, bởi vì nó gọi hàm getScratchDir() khi hàm getOptions() trả về, rồi lại gọi getAbsolutePath() khi hàm getScratchDir() trả về.

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

5.3 Train Wrecks (tàu trật bánh)

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

Đoạn code trên được gọi là "Train Wrecks" bởi vì nó rất hỗn độn và cần nên tránh. Ta thay đoạn code trên bằng đoạn code dưới như sau:

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```

Nhưng đoạn code trên có thể vi phạm luật Demeter tùy thuộc vào ctxt, opts, ScratchDir là đối tượng hay cấu trúc dữ liệu. Nếu chúng là đối tượng, thì cấu trúc bên trong trong chúng nên được ẩn đi hơn là phơi bày ra, vì thế hiểu biết về cấu trúc bên trong rõ ràng là đã vi phạm luật Demeter. Hơn nữa, nếu chúng là cấu trúc dữ liệu không thể hiện thì luật Demeter không được áp dụng vào đây.

6. Error Handling (Kiểm soát lỗi)

Kiểm soát lỗi là thứ chúng ta cần phải làm khi lập trình.

6.1 Use Exceptions Rather Than Return Codes

```
public class FitNesseServer implements SocketServer { private FitNesseContext
context; public FitNesseServer(FitNesseContext context) { this.context =
context; } public void serve(Socket s) { serve(s, 10000); } public void
serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new
FitNesseExpediter(s, context);
sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); }
catch(Exception e) { e.printStackTrace(); } } }

-----


public class FitNesseServer implements SocketServer {
  private FitNesseContext context;
```

Ở ví dụ trên, ta thấy cách viết rất hỗn độn. Do đó, chúng ta cần truyền ngoại lệ (exception) giúp cho code rõ ràng và sạch sẽ hơn như hình bên dưới.

```
public class DeviceController {
  ...

  public void sendShutDown() {
    try {
      tryToShutDown();
    } catch (DeviceShutDownError e) {
      logger.log(e);
    }
  }
```

```
    private void tryToShutDown() throws DeviceShutDownError {
      DeviceHandle handle = getHandle(DEV1);
      DeviceRecord record = retrieveDeviceRecord(handle);

      pauseDevice(handle);
      clearDeviceWorkQueue(handle);
      closeDevice(handle);
    }

    private DeviceHandle getHandle(DeviceID id) {
      ...
      throw new DeviceShutDownError("Invalid handle for: " + id.toString());
      ...
    }

    ...
  }
```

6.2 Write Your Try-Catch-Finally Statement First

Một trong những điều thú vị nhất về những ngoại lệ là nó xác định được vùng cụ thể trong chương trình. Khi thực thi code trong Try, chúng ta có thể bỏ ngang bất kì điểm nào và tiếp tục tại Catch.Vì thế chúng ta nên sử dụng Try-Catch-Finally khi viết code vì điều này khiến chúng ta xác định được mong muốn của người dùng thông qua việc bỏ qua các ngoại lệ.

6.3 Use Unchecked Exceptions

Trong một số trường hợp những ngoại lệ không được kiểm tra tại thời gian thực thi. Mặc dù Kiểm tra các ngoại lệ mang về một số lợi ích nhất định nhưng trong phát triển ứng dụng nói chung, chi phí phụ thuộc lớn hơn lợi ích.

6.4 Provide Context with Exception (cung cấp bối cảnh cho ngoại lệ)

Các ngoại lệ được ném đi nên cung cấp ngữ cảnh để xác định nguồn và vị trí của lỗi.

6.5 Define the Normal Flow

chúng ta có thể viết những dòng lệnh đơn giản bằng cách tạo một lớp hoặc tạo ra một đối tượng để kiểm soát lỗi của chương trình khi xảy ra những trường hợp đặc biệt. Khi đó, tính logic của chương trình sẽ trở nên đơn giản vì những trường hợp ngoại lệ đã được kiểm soát.

6.6 Don’t Return null

Khi chúng ta trả về null, chúng ta tự tạo công việc cho bản thân và thêm vấn đề cho người gọi nó. Tất cả vấn đề chính là thiếu mất một kiểm tra null để rồi gửi ứng dụng ra khỏi tầm kiểm soát.

6.7 Don’t pass Null

Trả về null từ phương thức là việc làm không tốt, nhưng bỏ qua null thậm chí còn tồi tệ hơn.Vì vậy, cần tránh bỏ qua null khi viết code nếu có thể. Trong nhiều ngôn ngữ lập trình, không có cách nào xử lý tốt với null khi nó được thông qua bởi lời gọi ngẫu nhiên. Vì vậy, bạn cần biết rằng khi nào null xuất hiện để xử lý lỗi tốt hơn.

7. Unit tests

Khi viết chương trình, tất nhiên bằng mắt thường bạn không thể nào chắc chắn rằng đoạn code của mình là hoàn toàn đúng đắn, và sẽ cho ra kết quả như mong đợi. Vì vậy, trong lập trình, giai đoạn test cũng đóng vai trò khá quan trọng trong việc hoàn thiện sản phẩm. Ngày nay, chúng ta gọi những chuyên gia trong lĩnh vực này là Test Driven Development. Nhưng bất cứ một lập trình viên nào cũng nên biết cách viết một test thật sạch sẽ, dễ bảo trì và nâng cấp. Nên chúng tôi khuyên khích bạn nên biết viết "automated unit tests". Trong chương này chúng ta sẽ cùng thảo luận về cách viết unit test đúng cách.

7.1 The three Laws of TDD (3 luật TDD)

7.1.1 Luật TDD

Yêu cầu chúng ta phải viết unit tests trước khi viết production code (đoạn code chương trình).

7.1.2 Ba luật TDD:

- Không viết bất kì một production code nào cho tới khi viết được một unit test bị fail.
- Unit test vừa viết nhất định phải fail.
- Không viết nhiều hơn một production code cái mà đủ để pass đoạn unit test vừa viết (nghĩa là, sau khi viết được production code, phải hiểu chỉnh cho test làm sao để đoạn chương trình có thể pass được test).

Như vậy, bạn thấy đó, việc viết test sẽ song song với việc viết code. Điều này dẫn tới hậu quả là, có khi số lượng dòng lệnh test sẽ tương đương với số lượng dòng lệnh chương trình. Dẫn chúng ta tới việc phải đối mặt với lượng dòng code khổng lồ "daunting management problem".

7.2 Tại sao cần giữ test sạch sẽ

- Sẽ có những bạn đánh giá thấp việc viết test, nên các bạn đã phá luật và thực hiện theo ý mình với phương châm "Quick and dirty". Nghĩa là khi viết code, các biến không cần được đặt tên tốt, các test dùng để kiểm tra thủ tục không cần ngắn và súc tích. Đoạn test cũng không cần tổ chức chặt chẽ và phân bố cục rõ ràng. Miễn là test chạy và miễn là nó giải quyết được vấn đề của đoạn code chương trình đang cần kiểm tra, vậy là đủ.
- Nhưng có một điều bạn nên biết rằng thà không có test còn hơn phải sử dụng "test bẩn". Bởi vì test càng dơ, chúng ta càng khó sửa chữa. Khi không thể hiệu chỉnh đoạn test, bạn sẽ phải viết nhiều hơn một đoạn test để phù hợp với chương trình bạn đang viết (khi chương trình cũ cần cải tiến hoặc thay đổi đôi chút). Dẫn tới hậu quả bạn sẽ tốn nhiều thời gian viết test hơn cả viết chương trình chính. Chi phí bảo trì sẽ tăng nếu như chương trình càng ngày càng lớn, và chỉ toàn chứa những đoạn "test bẩn". Một hậu quả khác nữa là việc giải quyết các đoạn "test dơ" sẽ khiến cho các developer cảm thấy sợ việc thay đổi code. Dẫn đến năng suất làm việc cũng giảm. Vậy chúng ta có thể nói rằng, việc viết test cũng quan trọng như việc viết chương trình chính.

7.3 Lợi ích của việc viết "test sạch"

- giữ test sạch sẽ sẽ làm production code của bạn được linh hoạt, có thể duy trì, tái sử dụng. Và lý do quan trọng nhất là nếu bạn có đoạn test sạch bạn sẽ không cảm thấy quá sợ hãi khi thay đổi đoạn code chương trình.

7.4 Cách viết một "test sạch"

- Một điều duy nhất tạo nên test sạch: READABILITY (có thể đọc và hiểu).
- READABILITY được cấu thành từ 3 yếu tố: clarity (sự rõ ràng), simplicity (dễ hiểu), và density of expression (súc tích).
- Dựa vào BUILD-OPERATE-CHECK pattern để biết cấu trúc một đoạn test:
  - Phần đầu: xây dựng dữ liệu test.
  - Phần giữa: tính toán trên bộ dữ liệu test.
  - Phần cuối: kiểm tra sự tính toán xem có trả về kết quả mong đợi hay không.

7.5 Domain-Specific Testing Language (Ngôn ngữ kiểm tra miền cụ thể)

- Mặc dù có công cụ để hỗ trợ việc viết test code, nhưng đừng vội thiết kế testing API, bạn hãy tự viết một đoạn code để kiểm tra kết quả chương trình. Phát triển từ những đoạn test code đơn giản, đoạn code mà gặp quá nhiều lỗi bởi những chi tiết rườm rà không rõ ràng.

7.6 A Dual Standard (tiêu chuẩn kép)

- Test code phải đơn giản, súc tích và có ý nghĩa, nhưng không cần phải hiệu quả như production code. Đôi khi, có những điều mà ban không bao giờ phải làm trong production environment cái mà bắt buộc phải có trong test environment, thường thì những điểm khác biệt đó sẽ liên quan tới vấn đề bộ nhớ và hiệu suất của CPU. Ví dụ như các chương trình ứng dụng sẽ thường chạy trong các hệ thống nhúng nên đoạn code thường bị ràng buộc bởi máy tính và bộ nhớ, còn test evironment thì không, vì thế khi viết test lập trình viên có vẻ tự do hơn. Nhưng code của test và production không bao giờ khác biệt về vấn đề sạch sẽ.

7.7 One assert per test (một câu khẳng định trên một test)

- Số lượng assert statement (câu khẳng định) trong một test nên được tối thiểu hóa để người đọc có thể hiểu mục đích của đoạn test một cách nhanh và dễ hình dung.

7.8 Single concept per test (một khái niêm trên một test)

- Bạn hãy nhớ chỉ nên có duy nhất một khái niệm được định nghĩa trong hàm test. Nhiều khái niệm trong cùng một hàm test sẽ làm người đọc bị rối: "tại sao những dòng test này lại nằm đây, mục đích của nó là gì", "hàm test này được gọi bởi phần nào trong chương trình",...
- Cách giải quyết đối với những test đã lỡ định nghĩa quá nhiều khái niệm: chia nhỏ các mục đích của đoạn test ra thành nhiều hàm test nhỏ khác nhau.

7.9 F.I.R.S.T

- Clean test cần đảm bảo 5 yếu tố:
  - Fast: test phải chạy nhanh.
  - Independent: Test không phụ thuộc vào các đoạn test khác.
  - Repeatable: Test phải lặp lại được trong các môi trường khác nhau.
  - Self-Validating: Test nên có output là kiểu boolean, kể cả khi pass hay fail.
  - Timely: Thời gian viết test phải đúng thời điểm, trước khi viết production code. Để tránh việc lập trình viên nản khi nhận ra production code quá khó để test.

7.10 Kết luận

- Test đóng vai trò không nhỏ trong toàn bộ dự án. Có thể nói viết test cũng quan trọng như viết chương trình. Bởi vì test sẽ đảm bảo và nâng cao tính linh hoạt, bảo trì và tái sử dụng của đoạn code chương trình. Vì vậy bạn hãy luôn giữ test của mình được sạch sẽ bằng cách làm chúng thật cô đọng, súc tích nhưng vẫn đầy đủ ý nghĩa.

8. Classes

Việc đặt các dòng lệnh vào vị trí phù hợp và kết nối chúng lại với nhau để người đọc dễ hiểu vẫn chưa thể khiến code của chúng ta hoàn toàn sạch sẽ. Ở chương này, bạn sẽ tiến tới mức độ cao hơn của việc tổ chức chương trình. Chúng ta hãy cùng nhau bàn về class.

8.1 Class organization (sự tổ chức class)

- Áp dụng luật tổ chức theo chiều từ cao xuống thấp, sẽ giúp đoạn code của chúng ta trong giống như một bài báo. Đầu tiên là danh sách các biến trong đó: hằng số tĩnh công khai (public static constants) để đầu tiên nếu có, biến tĩnh public (public static variables) được khai báo tiếp theo, sau nữa là biến tĩnh private (private static variables). Cuối cùng là các biến thể hiện (instance variables).

8.2 Encapsulation (sự đóng gói)

- Đối với class, sự đóng gói rất quan trọng vì nó giữ vai trò đảm bảo an toàn cho dữ liệu. Khi test muốn gọi hàm hoặc truy cập vào biến, chúng ta nên đổi dữ liệu thành kiểu protected hoặc để các biến trong phạm vi mà test có thể truy cập. Nếu tất cả các cách trên không thể giải quyết được vấn đề của bạn, thì lúc này việc đánh mất tính đóng gói dữ liệu mới nên được sử dụng. Tuyệt đối không gỡ bỏ tính đóng gói của class trong mọi trường hợp khi chưa tìm mọi cách để truy cập dữ liệu mà bạn cần.

8.3 Class should be small! (class nên nhỏ)

- "Nhỏ, nhỏ nữa, nhỏ mãi" là luật cơ bản nhất phải nhớ khi thiết kế một class. Nhưng nhỏ tới cỡ nào là đủ??? Thông thường chúng ta dựa vào số dòng lệnh để đánh giá kích cỡ của một fucntion. Nhưng đối với class, chúng ta ước lượng kích cỡ class dựa trên số lượng "trách nhiệm" (resposibilities).
- Đặt tên class một cách thông minh là cách tốt nhất để cho người đọc dễ nhận ra chắc năng và trách nhiệm của một class
- Cách đặt tên class:
  - Sử dụng một cái tên súc tích: nghĩa là cái tên nên mô tả ngắn gọn một cái clas trong 25 từ.
  - Không sử dùng từ đa nghĩa và các liên từ như “if”, “and”, “or”, “but”, vì những từ này sẽ khiến class trông như có rất nhiều trách nhiệm.

8.4 The Single Responsibility Principle (SRP) (nguyên tắc một trách nhiệm)

- Luật chơi ở đây là “một trách nhiệm – một lý do để thay đổi”.
- Chúng ta cần những hệ thống bao gồm những class nhỏ hơn là ít class lớn. Vì thỉnh thoảng chúng ta cần truy cập vào các class để tìm kiếm thông tin, nếu đó là một class lớn, thực hiện quá chiều chức năng, bạn sẽ rất tốn thời gian để tìm ra điểm quan trọng nhất mà bạn quan tâm. Vì vậy, việc chia nhỏ các class sẽ giúp bạn dễ dàng tổ chức, dọn dẹp và tìm kiếm. Tránh tốn thời gian vào những thông tin không cần thiết.

8.5 Cohesion (sự kết hợp) && Maintaining Cohesion Results in Many Small Classes (duy trì sự kết hợp trong những class nhỏ)

- Phương thức càng kiểm soát được nhiều biến, thì sự kết dính trong một class càng cao. Nghĩa là những phương thức và biến trong class càng tương tác nhiều thì tính đoàn kết càng cao. Và hãy nhớ rằng, chúng ta cố tình không tạo ra một class có sự kết hợp cao nhất, mà chúng ta chỉ muốn sự kết hợp trong class được cao nhất thôi.
- Vậy sự kết dính tối đa ở đây là các phương thức và các biến của class cùng phụ thuộc và có sự gắn kết với nhau như một tổng thể hợp lý.
- Khi một clas mất tính đoàn kết, hãy chia rẻ chúng, đừng để chúng phá hủy kết cấu chặt chẽ của bạn.

8.6 Organize for change (tổ chức cho sự thay đổi)

- Trong nhiều hệ thống, chương trình luôn được thay đổi và nâng cấp để đáp ứng với yêu cầu người dùng. Vì vậy việc bắt kịp xu hướng thường ép chúng ta phải đối mặt với nhiều nguy cơ của sự thay đổi này.
- Cách đối mặt với rủi ro khi chương trình cần thay đổi:
    - Giảm thiểu việc thay đổi chương trình.
    - Nếu cần cải tiến chương trình, chúng ta không sửa chương trình, mà hãy mở rộng chương trình và liên kết những đoạn code đang tồn tại lại với nhau.

LỜI KẾT

Vậy viết code sạch không khó như chúng ta vẫn nghĩ, nhưng cũng không phải chuyện dễ dàng có thể một sớm một chiều mà thành thạo. Đã là một lập trình viên, chúng ta nên biết những nguyên tắc quan trọng về việc viết code sạch như đã trình bày ở trên. Nhưng chúng ta không chỉ dừng lại ở vị trị một lập trình viên bình thường, chúng ta muốn chuyên nghiệp và kĩ năng, vậy thì càng phải nắm thật rõ những nguyên tắc trong sách clean code, và không ngừng luyện tập để nâng cao trình độ và chuyên môn của bản thân. Chúng tôi chúc các bạn thành công trên con đường tìm kiếm tri thức của mình và rất mong một chút kiến thức nho nhỏ ở trên có thể giúp các bạn phần nào đỡ vất vả hơn trên con đường ấy. Một lần nữa, cảm ơn rất nhiều.