

# IT4490 – Thiết kế và xây dựng phần mềm

## Bài 9. Lập trình

1

### For Your Amusement

- ❖ “Any fool can write code that a computer can understand. Good programmers write code that humans can understand” -- Martin Fowler
- ❖ “Good code is its own best documentation. As you’re about to add a comment, ask yourself, ‘How can I improve the code so that this comment isn’t needed?’ ” -- Steve McConnell
- ❖ “Programs must be written for people to read, and only incidentally for machines to execute.” -- Abelson / Sussman
- ❖ “Everything should be built top-down, except the first time.” -- Alan Perlis

### Các cách để code của bạn đúng

- ❖ Xác minh / đảm bảo chất lượng
  - Mục đích là để khám phá các vấn đề và tăng cường sự tự tin
  - Kết hợp giữa lý luận và kiểm tra
- ❖ Debugging
  - Tìm hiểu lý do tại sao một chương trình không hoạt động như dự kiến
- ❖ Lập trình phòng thủ
  - Lập trình với xác nhận và gỡ lỗi trong tâm trí
- ❖ Testing ≠ debugging
  - test: tiết lộ tồn tại của vấn đề; bộ kiểm thử cũng có thể tăng độ tin cậy tổng thể
  - debug: xác định vị trí + nguyên nhân của sự cố

### Nội dung

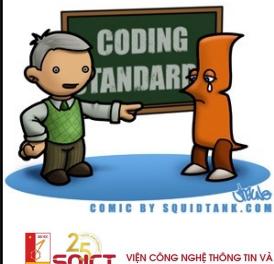
- ⇒ 1. Phong cách lập trình
- 2. Tinh chỉnh / tối ưu mã (tuning / optimization)
- 3. Tái cấu trúc mã (refactoring)
- 4. Debugging

3

4

## Phong cách lập trình

- ❖ Quy ước / tiêu chuẩn viết mã



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

5



5

## Phong cách lập trình thích hợp

- ❖ Sử dụng hằng số cho tất cả các giá trị không đổi (ví dụ: thuế suất).
- ❖ Sử dụng các biến bất cứ khi nào có thể
- ❖ Luôn khai báo các hằng số trước các biến khi bắt đầu thủ tục
- ❖ Luôn chú thích mã nguồn (đặc biệt là các đoạn mã không rõ ràng hoặc gây hiểu lầm), nhưng không chú thích quá mức.
- ❖ Sử dụng tên mô tả thích hợp (có / không có tiền tố) cho các định danh / đối tượng (ví dụ: btnDone).



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

6

## Phong cách lập trình thích hợp (2)

- ❖ Sử dụng dấu ngoặc cho các biểu thức toán học ngay cả khi không bắt buộc để có thể hiểu rõ ý định của nó.
  - Ví dụ:  $(3*2) + (4*2)$
- ❖ Luôn viết mã hiệu quả nhất có thể
- ❖ Tạo biểu mẫu input và output thân thiện với người dùng input and output

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

7

## Phong cách lập trình thích hợp (3)

- ❖ Thêm một phần tiêu đề ở đầu mã nguồn:
  - Tên người lập trình
  - Ngày
  - Tên của dự án đã lưu
  - Tên của giáo viên
  - Tên lớp
  - Tên của bất kỳ ai đã giúp bạn
  - Mô tả ngắn gọn về những gì chương trình thực hiện

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

8

## Công cụ kiểm tra style

- ❖ Demo video



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

9

9

## Tinh chỉnh mã (Code tuning)

- ❖ Sửa đổi mã chính xác để làm cho nó chạy hiệu quả hơn
- ❖ Không phải là cách hiệu quả nhất / rẻ nhất để cải thiện hiệu suất
- ❖ 20% các phương thức của chương trình tiêu tốn 80% thời gian thực thi của chương trình.



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

11

11

## Nội dung

1. Phong cách lập trình
2. Tinh chỉnh / tối ưu mã (tuning / optimization)
3. Tái cấu trúc mã (refactoring)
4. Debugging



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

10

10

## Code Tuning Myths

- ❖ Giảm các dòng mã bằng ngôn ngữ cấp cao sẽ cải thiện tốc độ hoặc kích thước của mã máy kết quả - sai!

```
for i = 1 to 10  
    a[ i ] = i  
end for
```

VS

```
a[ 1 ] = 1  
a[ 2 ] = 2  
a[ 3 ] = 3  
a[ 4 ] = 4  
a[ 5 ] = 5  
a[ 6 ] = 6  
a[ 7 ] = 7  
a[ 8 ] = 8  
a[ 9 ] = 9  
a[ 10 ] = 10
```



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

12

3

## Code Tuning Myths (2)

- ❖ Một chương trình nhanh cũng quan trọng như một chương trình đúng - sai!



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

13

13

## Code Tuning Myths (3)

- ❖ Một số hoạt động có thể nhanh hơn hoặc nhỏ hơn những hoạt động khác - sai!
  - Luôn đo lường hiệu suất!



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

14

14

## Khi nào thì tinh chỉnh

- ❖ Sử dụng thiết kế chất lượng cao
  - Thực hiện đúng chương trình.
  - Xây dựng theo mô-đun và dễ dàng sửa đổi
  - Khi hoàn thành và chính xác, hãy kiểm tra hiệu suất.
- ❖ Xem xét tối ưu hóa trình biên dịch
- ❖ Đo lường
- ❖ Viết mã rõ ràng, dễ hiểu và dễ sửa đổi.



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

15

15

16

## Đo lường

- ❖ Đo lường để tìm ra điểm nghẽn
- ❖ Các phép đo cần phải chính xác
- ❖ Các phép đo cần được lắp lại



## Các kỹ thuật Code Tuning

- ❖ Ngừng kiểm tra khi bạn biết câu trả lời

```
if ( 5 < x ) and ( y < 10 ) then ...
```

```
?
```

```
negativeInputFound = False;
for ( i = 0; i < iCount; i++ ) {
    if ( input[ i ] < 0 ) {
        negativeInputFound = True;
    }
}
```

## Tối ưu hóa trong các lần lặp lại

- ❖ Đo lường sự cải thiện sau mỗi lần tối ưu hóa
- ❖ Nếu tối ưu hóa không cải thiện hiệu suất - hãy hoàn nguyên nó



## Các kỹ thuật Code Tuning

- ❖ Sắp xếp các kiểm tra theo tần suất

```
Select char
Case "+", "="
    ProcessMathSymbol(char)
Case "0" To "9"
    ProcessDigit(char)
Case ",", ".", "!", "?"
    ProcessPunctuation(char)
Case " "
    ProcessSpace(char)
Case "A" To "Z", "a" To "z"
    ProcessAlpha(char)
Case Else
    ProcessError(char)
End Select
```

```
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
```

```
Select char
Case "A" To "Z", "a" To "z"
    ProcessAlpha(char)
Case " "
    ProcessSpace(char)
Case ",", ".", "!", "?"
    ProcessPunctuation(char)
Case "0" To "9"
    ProcessDigit(char)
Case "+", "="
    ProcessMathSymbol(char)
Case Else
    ProcessError(char)
End Select
```

```
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
```

## Các kỹ thuật Code Tuning

### ❖ Unswitching loops

```
for ( i = 0; i < count; i++ ) {  
    if ( sumType == SUMTYPE_NET ) {  
        netSum = netSum + amount[ i ];  
    }  
    else { grossSum = grossSum + amount[ i ]; }  
}
```

?



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

21

## Các kỹ thuật Code Tuning

### ❖ Giảm thiểu công việc bên trong các vòng lặp

```
for ( i = 0; i < rateCount; i++ ) {  
    netRate[i] = baseRate[i] * rates->discounts->factors->net;  
}
```

```
?  
for ( i = 0; i < rateCount; i++ ) {  
    netRate[i] = baseRate[i] * ?  
}
```



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

22

## Các kỹ thuật Code Tuning

### ❖ Khởi tạo tại thời gian biên dịch

```
const double Log2 = 0.69314718055994529;
```



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

23

## Các kỹ thuật Code Tuning

### ❖ Sử dụng Lazy Evaluation

```
public int getSize() {  
    if(size == null) {  
        size = the_series.size();  
    }  
    return size;  
}
```



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

24

23

## Các kỹ thuật Code Tuning

```
var myClass = function() {  
    this.array_one = [1,2,3,4,5];  
    this.array_two = [1,2,3,4,5];  
    this.total = 0;  
}  
  
var my_instance = new myClass();  
  
for (var i=0; i < 4; i++) {  
    my_instance.total +=  
        (my_instance.array_one[i] +  
         my_instance.array_two[i]);  
}  
25
```



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

25

Khi lặp qua dữ liệu, hãy giữ các tham chiếu bộ nhớ tuần tự

```
var myClass = function() {  
    this.array_one = [1,2,3,4,5];  
    this.array_two = [1,2,3,4,5];  
    this.total = 0;  
}  
  
var my_instance = new myClass();  
  
?  
26
```



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

26

## Nội dung

1. Phong cách lập trình
2. Tinh chỉnh / tối ưu mã (tuning / optimization)
3. Tái cấu trúc mã (refactoring) ➡
4. Debugging



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

27

## Refactoring là gì?

Refactoring nghĩa là "  
để cải thiện thiết kế và  
chất lượng của mã  
nguồn hiện tại mà  
không thay đổi hành vi  
bên ngoài của nó".  
*Martin Fowler*



- ❖ Quy trình từng bước biến mã xấu thành mã tốt
  - Dựa trên "refactoring patterns" → các công thức nổi tiếng để cải thiện mã



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

28

## Code Refactoring

- Tái cấu trúc mã nguồn là gì?
  - Cải thiện thiết kế và chất lượng của mã nguồn hiện có mà không thay đổi hành vi của nó
  - Quy trình từng bước để biến mã xấu thành mã tốt (nếu có thể)
- Tại sao chúng ta cần tái cấu trúc?
  - Mã liên tục thay đổi và chất lượng của nó liên tục giảm (trừ khi được cấu trúc lại)
  - Các yêu cầu thường thay đổi và mã cần được thay đổi để tuân theo chúng



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

29

29

## Refactoring: các nguyên lý chính

- Giữ mọi thứ đơn giản - Keep it simple (KISS principle)
- Tránh trùng lặp - Avoid duplication (DRY principle)
- Make it expressive (self-documenting, comments, etc.)
- Giảm mã tổng thể (KISS principle)
- Phân tách mối quan tâm - Separate concerns (decoupling)
- Mức độ trừu tượng phù hợp (làm việc thông qua các mức trừu tượng hóa)
- Quy tắc hướng đạo sinh (Boy scout rule)
  - Để lại mã của bạn tốt hơn sau bạn tìm thấy nó



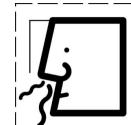
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

31

31

## Khi nào thì Refactor?

- “Bad smells in the code” cho thấy cần phải cấu trúc lại
- Refactor:
  - Để thêm một chức năng mới dễ dàng hơn
  - Là một phần của quá trình sửa lỗi
  - Khi xem lại mã của người khác
  - Có mòn nợ kỹ thuật (hoặc bất kỳ mã có vấn đề nào)
  - Khi thực hiện phát triển dựa trên kiểm thử
- Unit tests đảm bảo rằng việc tái cấu trúc không thay đổi hành vi
  - Nếu không có kiểm thử đơn vị nào, hãy viết chúng



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

30

30

## Refactoring: quy trình điển hình

1. Hãy lưu lại mã bạn bắt đầu
  - Check-in hoặc sao lưu mã hiện tại
2. Chuẩn bị các tests để đảm bảo hành vi sau khi mã được cấu trúc lại
  - Unit tests / characterization tests
3. Thực hiện refactoring từng cái một
  - Thực hiện tái cấu trúc nhỏ
  - Đừng đánh giá thấp những thay đổi nhỏ
4. Chạy các tests và chúng nên pass / hoặc hoàn nguyên
5. Check-in (vào hệ thống kiểm soát mã nguồn)



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

32

32

## Các mẹo Refactoring

- ❖ Hãy refactoring trên các phần nhỏ
- ❖ Từng lượt một (One at a time)
- ❖ Tạo danh sách checklist
- ❖ Tạo một "later" / TODO list
- ❖ Thường xuyên check-in / commit
- ❖ Thêm các tests cases
- ❖ Xét duyệt kết quả
  - Pair programming
- ❖ Sử dụng các công cụ (Visual Studio + add-ins / Eclipse + plugins / others)



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

33



25  
SOICT

ĐẠI HỌC BÁCH KHOA HÀ NỘI  
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG



## Code Refactoring Live Demo

34

## Code Smells

- Code smells (code thối / mã xấu) == các cấu trúc nhất định trong mã dẫn đến khả năng tái cấu trúc
- Các loại code smells:
  - The bloaters
  - The obfuscators
  - Object-oriented abusers
  - Change preventers
  - Dispensables
  - The couplers



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

35

## Code Smells: The Bloaters

- Phương thức dài (Long method)
  - Các phương pháp nhỏ luôn tốt hơn (đặt tên dễ dàng, dễ hiểu, mã ít trùng lặp)
- Lớp lớn (Large class)
  - Quá nhiều các biến hoặc phương thức thể hiện
  - Vi phạm nguyên lý "Single Responsibility"
- Primitive obsession (lạm dụng primitives)
  - Sử dụng quá mức các giá trị nguyên thủy, thay vì trùm tương hóa tốt hơn
  - Can be extracted in separate class with encapsulated validation



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

36

35

9

## Code Smells: The Bloaters (2)

- Danh sách tham số dài - Long parameter list (**in / out / ref parameters**)
  - Có thể biểu thị phong cách thủ tục hơn là OO
  - Có thể là phương thức đang làm quá nhiều thứ
- Các khối dữ liệu (Data clumps)
  - Tập hợp dữ liệu luôn được sử dụng cùng nhau, nhưng không được tổ chức cùng nhau
  - E.g. trường thẻ tín dụng trong lớp Đơn hàng
- Bùng nổ tổ hợp
  - Ex. ListCars(), ListByRegion(), ListByManufacturer(), ListByManufacturerAndRegion(), etc.
  - Giải pháp có thể là Interpreter pattern (LINQ)



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

37

37

## Code Smells: The Obfuscators

- Regions
  - Mục đích của mã không rõ ràng và cần nhận xét (smell)
  - Mã quá dài để hiểu (smell)
  - Giải pháp: partial class, a new class, organize code
- Comments
  - Nên dùng để nói WHY, not WHAT or HOW
  - Good comments: cung cấp thông tin bổ sung, liên kết đến vấn đề, giải thích thuật toán, giải thích lý do, đưa ra ngữ cảnh
  - Link: [Funny comments](#)



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

39

39

## Code Smells: The Bloaters (3)

- Oddball solution
  - Một cách khác để giải quyết một vấn đề chung
  - Không sử dụng tính nhất quán
  - Giải pháp: Thay thế thuật toán hoặc sử dụng Adapter
- Class doesn't do much
  - Giải pháp: Hợp nhất với một lớp khác hoặc xóa bỏ
- Đòi hỏi setup / teardown code
  - Yêu cầu một số dòng mã trước khi sử dụng
  - Giải pháp: sử dụng đối tượng tham số, factory method, IDisposable



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

38

38

## Code Smells: The Obfuscators (2)

- Tên kém / không phù hợp (Poor / improper names)
  - Phải phù hợp, mô tả và nhất quán
- Vertical separation
  - Bạn nên xác định các biến ngay trước khi sử dụng lần đầu tiên để tránh cuộn
  - Trong JS, các biến được định nghĩa khi bắt đầu hàm → sử dụng các hàm nhỏ
- Inconsistency
  - Thực hiện theo POLA (Nguyên lý ít suy giảm nhất)
  - Sự không nhất quán gây nhầm lẫn và mất tập trung
- Obscured intent
  - Mã phải càng diễn đạt càng tốt



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

40

40

10

## Code Smells: OO Abusers

- Switch statement
  - Có thể được thay thế bằng tính đa hình
- Temporary field
  - Khi truyền dữ liệu giữa các phương thức
- Class depends on subclass
  - Các lớp không thể tách rời nhau (phụ thuộc vòng tròn)
  - Có thể phá vỡ nguyên lý thay thế Liskov
- Inappropriate static field
  - Strong coupling giữa static và callers
  - Những thứ tĩnh không thể thay thế hoặc sử dụng lại

## Code Smells: Change Preventers

- Divergent change
  - Một lớp thường được thay đổi theo nhiều cách khác nhau / lý do khác nhau
  - Vi phạm SRP (single responsibility principle)
  - Giải pháp: extract class
- Shotgun surgery
  - Một thay đổi đòi hỏi nhiều thay đổi trong nhiều lớp
  - Khó tìm chúng, dễ bỏ sót một số
  - Giải pháp: di chuyển các phương thức, di chuyển các trường, tổ chức lại mã

## Code Smells: Change Preventers (2)

- Conditional complexity
  - Độ phức tạp theo chương trình - Cyclomatic complexity (số lượng đường dẫn duy nhất mà mã có thể được đánh giá)
  - Các hiện tượng: deep nesting (arrow code) and buggy if-s
  - Các giải pháp: extract method, "Strategy" pattern, "State" pattern, "Decorator"
- Poorly written tests
  - Các test viết không tốt có thể ngăn cản sự thay đổi
  - Khớp nối chặt chẽ (Tight coupling)

## Code Smells: Dispensables

- Lazy class
  - Các lớp không đủ để chứng minh sự tồn tại của chúng nên bị xóa
  - Mỗi lớp đều tốn chi phí để hiểu và duy trì
- Data class
  - Một số lớp chỉ có trường và thuộc tính
  - Thiếu sự công nhận? Lớp logic tách thành các lớp khác?
  - Giải pháp: chuyển logic liên quan vào lớp

## Code Smells: Dispensables (2)

### ❖ Duplicated code

- Vi phạm nguyên lý DRY
- Kết quả của việc copy-paste code
- Giải pháp: extract method, extract class, pull-up method, *Template Method* pattern

### ❖ Dead code (mã không bao giờ sử dụng)

- Thường được phát hiện bởi các công cụ phân tích tĩnh

### ❖ Speculative generality

- "Một ngày nào đó chúng ta có thể cần cái này..."
- Nguyên tắc "YAGNI"



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

45

45

## Code Smells: The Couplers

### • The Law of Demeter (LoD)

- Một đối tượng nhất định phải giả định càng ít càng tốt về cấu trúc hoặc thuộc tính của bất kỳ thứ gì khác
- Bad e.g.: `customer.Wallet.RemoveMoney()`

### • Indecent exposure

- Một số lớp hoặc thành viên là công khai nhưng thực tế không nên là như vậy
- Vi phạm tính năng đóng gói
- Có thể dẫn đến sự thân mật không phù hợp



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

47

47

## Code Smells: The Couplers

### • Feature envy

- Phương thức có vẻ quan tâm hơn đến một lớp khác với lớp mà nó thực sự có trong
- Những điều cùng nhau thay đổi nên được đặt cùng nhau

### • Inappropriate intimacy

- Những lớp biết quá nhiều về nhau
- Smells: thừa kế, các mối quan hệ hai chiều
- Giải pháp: move method / field, extract class, change bidirectional to unidirectional association, replace inheritance with delegation



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

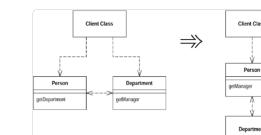
46

46

## Code Smells: The Couplers (3)

### • Message chains

- `Something.Another.SomeOther.Other.YetAnother`
- Khớp nối chát chẽ giữa máy khách và cấu trúc của điều hướng



### • Middle man

- Đôi khi việc ủy quyền đi quá xa
- Đôi khi chúng ta có thể xóa nó hoặc inline

### • Tramp data

- Chuyển dữ liệu chỉ vì thứ khác cần nó
- Solutions: Remove middle-man data, extract class



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

48

48

## Code Smells: The Couplers (4)

- Artificial coupling
  - Những thứ không phụ thuộc vào nhau không nên được ghép nối một cách giả tạo
- Hidden temporal coupling
  - Các hoạt động được thực hiện liên tiếp không nên đoán
  - E.g. pizza class should not know the steps of making pizza -> Template Method pattern
- Hidden dependencies
  - Các lớp nên khai báo các phụ thuộc của chúng trong hàm khởi tạo
  - new is glue / Dependency Inversion principle



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

49

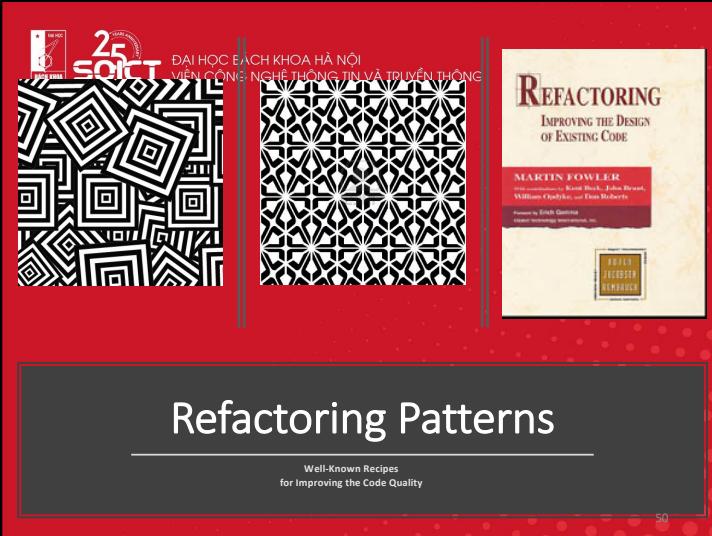
## Rafactoring Patterns

- Khi nào chúng ta nên thực hiện cấu trúc lại mã?
  - Bad smells trong mã cho thấy cần phải cấu trúc lại
- Các kiểm thử đơn vị đảm bảo rằng việc tái cấu trúc sẽ duy trì hành vi
- Rafactoring patterns
  - Các đoạn mã lặp lại lớn → trích xuất mã trùng lặp trong các phương thức riêng biệt
  - Large methods → phân chia chúng một cách hợp lý
  - Thân vòng lặp hoặc lồng nhau sâu → extract method



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

51



50

## Refactoring Patterns

Well-Known Recipes  
for Improving the Code Quality

50

## Refactoring Patterns (2)

- Lớp hoặc phương thức có tính liên kết yếu → tách thành nhiều lớp / phương thức
- Thay đổi đơn lẻ thực hiện thay đổi trong một số lớp → các lớp có sự liên kết chặt chẽ với nhau → xem xét thiết kế lại
- Dữ liệu liên quan luôn được sử dụng cùng nhau nhưng không phải là một phần của một lớp đơn lẻ → nhóm chúng trong một lớp
- Một phương thức có quá nhiều tham số → tạo một lớp để nhóm các tham số lại với nhau
- Một phương thức gọi nhiều phương thức từ lớp khác hơn là từ lớp của chính nó → di chuyển nó



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

52

13

### Rafactoring Patterns (3)

- Hai lớp được kết hợp chặt chẽ với nhau → hợp nhất chúng hoặc thiết kế lại chúng để tách biệt trách nhiệm của chúng
- Các trường công khai không phải hằng số → đặt chúng ở chế độ riêng tư và xác định các thuộc tính truy cập
- Magic numbers trong mã → xem xét trích xuất các hằng số
- Lớp / phương thức / biến có tên không hợp lệ → đổi tên nó
- Điều kiện boolean phức tạp → chia nó thành một số biểu thức hoặc lệnh gọi phương thức

### Rafactoring Patterns (4)

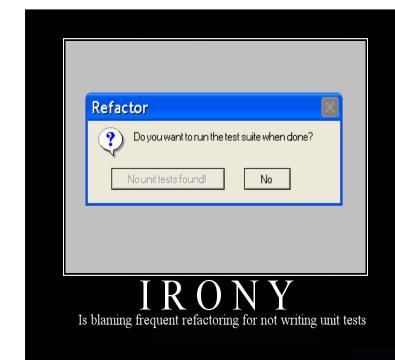
### Rafactoring Patterns (4)

- Biểu thức phức tạp → chia nó thành một vài phần đơn giản
- Một tập hợp các hằng được sử dụng làm kiểu liệt kê → chuyển đổi nó thành kiểu liệt kê
- Logic phương thức quá phức tạp → trích xuất một số phương thức đơn giản hơn hoặc thậm chí tạo một lớp mới
- Các lớp, phương thức, tham số, biến không sử dụng → loại bỏ chúng
- Dữ liệu lớn được truyền theo giá trị mà không có lý do chính đáng → chuyển nó theo tham chiếu

### Rafactoring Patterns (5)

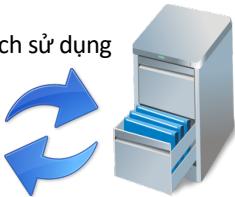
- Một số lớp chia sẻ chức năng lặp lại → trích xuất lớp cơ sở và sử dụng lại mã chung
- Các lớp khác nhau cần được khởi tạo tùy thuộc vào cài đặt cấu hình → sử dụng factory
- Mã không được định dạng tốt → định dạng lại nó
- Quá nhiều lớp trong một không gian tên duy nhất → chia các lớp một cách hợp lý thành nhiều không gian tên hơn
- Không sử dụng các định nghĩa → loại bỏ chúng
- Các thông báo lỗi không mang tính mô tả → cải thiện chúng
- Không có lập trình phòng thủ → thêm nó

### Refactoring Levels



## Data-Level Refactoring

- ❖ Thay một số ma thuật bằng một hằng số được đặt tên
- ❖ Đổi tên một biến bằng tên nhiều thông tin hơn
- ❖ Thay thế một biểu thức bằng một phương thức
  - Để đơn giản hóa nó hoặc tránh trùng lặp mã
- ❖ Di chuyển một biểu thức inline
- ❖ Giới thiệu một biến trung gian
  - Giới thiệu biến giải thích
- ❖ Chuyển đổi một biến sử dụng nhiều lần thành nhiều biến sử dụng một lần
  - Tạo biến riêng biệt cho từng cách sử dụng



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

57

## Data-Level Refactoring (2)

- Tạo một biến cục bộ cho các mục đích cục bộ thay vì một tham số
- Chuyển đổi một dữ liệu nguyên thủy thành một lớp
  - Hành vi bổ sung / logic xác thực (tiền)
- Chuyển đổi một tập mã kiểu (hằng số) thành enum
- Chuyển đổi một tập mã kiểu thành một lớp với các lớp con có hành vi khác nhau
- Thay đổi một mảng thành một đối tượng
  - Khi bạn sử dụng một mảng có các kiểu khác nhau trong đó
- Đóng gói một tập hợp



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

58

## Statement-Level Refactoring

- Phân rã một biểu thức boolean
- Di chuyển một biểu thức boolean phức tạp thành một hàm boolean được đặt tên tốt
- Sử dụng break hoặc return thay vì biến điều khiển vòng lặp
- Trả lại ngay khi bạn biết câu trả lời thay vì chỉ định giá trị trả về
- Hợp nhất mã trùng lặp trong điều kiện
- Thay thế các điều kiện bằng đa hình
- Sử dụng mẫu thiết kế đối tượng null thay vì kiểm tra null



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

59

## Method-Level Refactoring

- ❖ Trích xuất Phương thức / nội tuyến phương thức
- ❖ Đổi tên một phương thức
- ❖ Chuyển một phương thức dài thành một lớp
- ❖ Thêm / bớt tham số
- ❖ Kết hợp các phương thức tương tự bằng cách tham số hóa chúng
- ❖ Thay thế một thuật toán phức tạp bằng một thuật toán đơn giản hơn
- ❖ Các phương thức riêng biệt có hành vi phụ thuộc vào các tham số được truyền vào (tạo các phương thức mới)
- ❖ Chuyển toàn bộ đối tượng thay vì các trường cụ thể
- ❖ Đóng gói downcast / return các kiểu giao diện

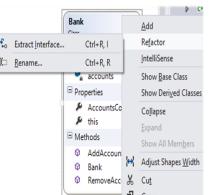


VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

60

## Class-Level Refactoring

- ❖ Thay đổi cấu trúc thành lớp và ngược lại
- ❖ Kéo các thành viên lên / đẩy các thành viên xuống hệ thống phân cấp
- ❖ Trích xuất mã chuyên biệt thành một lớp con
- ❖ Kết hợp mã tương tự vào một lớp cha
- ❖ Thu gọn hệ thống phân cấp
- ❖ Thay thế kế thừa bằng ủy quyền
- ❖ Thay thế ủy quyền bằng kế thừa



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

61

61

## Class Interface Refactoring (2)

- ❖ Đóng gói một biến thành viên bị bộc lộ
  - Luôn sử dụng thuộc tính
  - Xác định quyền truy cập thích hợp vào getters và setters
    - Xóa setters thành dữ liệu chỉ đọc
- ❖ Ẩn dữ liệu và quy trình không nhằm mục đích sử dụng bên ngoài lớp / cấu trúc phân cấp
  - private -> protected -> internal -> public
- ❖ Sử dụng chiến lược để tránh phân cấp lớp lớn
- ❖ Áp dụng các mẫu thiết kế khác để giải quyết các vấn đề phổ biến về phân cấp lớp và lớp (**Facade**, **Adapter**, v.v.)



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

63

63

## Class Interface Refactorings

- Trích xuất (các) giao diện / giữ phân tách giao diện
- Di chuyển một phương thức sang một lớp khác
- Tách một lớp / hợp nhất các lớp / xóa một lớp
- Ẩn lớp ủy nhiệm
  - A gọi B và C khi A nên gọi B và B gọi C
- Loại bỏ người đàn ông ở giữa
- Giới thiệu (sử dụng) một lớp mở rộng
  - Khi bạn không có quyền truy cập vào lớp ban đầu
  - Hoặc sử dụng mẫu Decorator



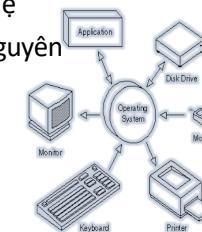
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

62

62

## System-Level Refactoring

- ❖ Di chuyển lớp (tập hợp các lớp) sang không gian tên / hợp ngữ khác
- ❖ Cung cấp một phương thức gốc thay vì một hàm tạo đơn giản / sử dụng API thông thạo
- ❖ Thay thế mã lỗi bằng các ngoại lệ
- ❖ Trích xuất chuỗi thành tệp tài nguyên
- ❖ Sử dụng dependency injection
- ❖ Áp dụng các mẫu kiến trúc



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

64

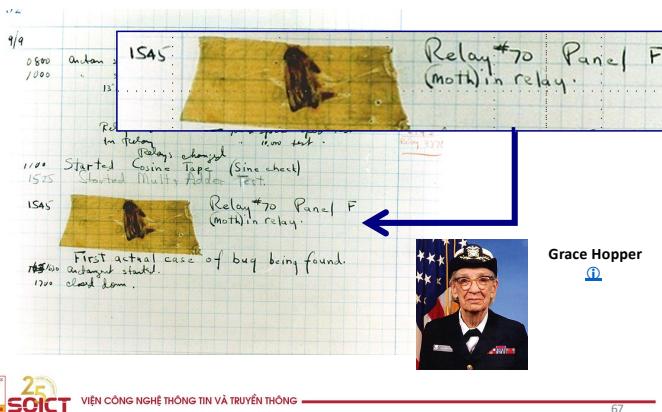
64

## Nội dung

1. Phong cách lập trình
2. Tinh chỉnh / tối ưu mã (tuning / optimization)
3. Tái cấu trúc mã (refactoring)
4. Debugging

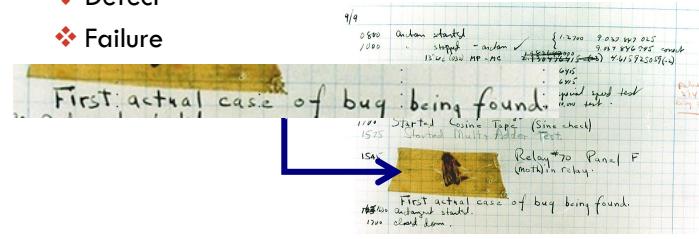
## Khi nào “bug”

- ❖ Alan Perlis 



## 4.1. Tổng quan

- ❖ Error
- ❖ Bug
- ❖ Fault
- ❖ Defect
- ❖ Failure



## 4.2. Phòng thủ theo chiều sâu

- ❖ Không thể xảy ra lỗi
  - Java làm cho lỗi ghi đè bộ nhớ không thể xảy ra
- ❖ Không giới thiệu các khiếm khuyết
  - Tính đúng đắn: làm mọi thứ đúng ngay lần đầu tiên
- ❖ Hiển thị lỗi ngay lập tức
  - Khả năng hiển thị lỗi cục bộ: tốt nhất là không thành công ngay lập tức
  - Ví dụ: assertions
- ❖ Phương án cuối cùng là gỡ lỗi
  - Cần thiết khi thất bại (ảnh hưởng) khác xa với nguyên nhân (khuyết tật)
  - Phương pháp khoa học: Thiết kế các thí nghiệm để thu thập thông tin về khuyết tật
    - Khá dễ dàng trong một chương trình có mô-đun tốt, ẩn biểu diễn, thông số kỹ thuật, kiểm tra đơn vị, v.v.
    - Khó hơn và khó hơn nhiều với một thiết kế kém, ví dụ: với phơi sáng tràn lan

#### 4.2.1. Phòng thủ đầu tiên: Không thể theo thiết kế

- ❖ Bằng ngôn ngữ
  - Java làm cho lỗi ghi đè bộ nhớ không thể xảy ra
- ❖ Trong các giao thức / thư viện / mô-đun
  - TCP / IP đảm bảo rằng dữ liệu không được sắp xếp lại
  - BigInteger đảm bảo rằng không có tràn
- ❖ Trong các quy ước tự áp đặt
  - Cấm đệ quy ngăn chặn đệ quy vô hạn / không đủ ngăn xếp - mặc dù nó có thể đẩy vấn đề sang chỗ khác
  - Cấu trúc dữ liệu bất biến đảm bảo bình đẳng về hành vi
  - Thận trọng: Bạn phải duy trì kỷ luật



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

69

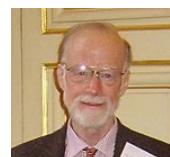
69

#### Cố gắng vì sự đơn giản

"There are two ways of constructing a software design:

- ❖ One way is to make it so simple that there are obviously no deficiencies, and
- ❖ the other way is to make it so complicated that there are no obvious deficiencies.

The first method is far more difficult."



Sir Anthony Hoare



"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."



Brian Kernighan



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

71

71

#### 4.2.2. Phòng thủ thứ hai: Tính đúng đắn

- ❖ Làm mọi thứ đúng ngay lần đầu tiên
  - Hãy suy nghĩ trước khi viết mã. Đừng viết mã trước khi bạn nghĩ!
  - Nếu bạn đang tạo ra nhiều kiểm khuyết để tìm, thì bạn cũng đang tạo ra những kiểm khuyết khó tìm - đừng sử dụng trình biên dịch như một cái nạng
- ❖ Đặc biệt đúng, khi gỡ lỗi sẽ khó
  - Đồng thời, môi trường thời gian thực, không có quyền truy cập vào môi trường khách hàng, v.v.
- ❖ Đơn giản là chìa khóa
  - Môđun hóa (Modularity)
    - Chia chương trình thành nhiều phần dễ hiểu
    - Sử dụng các kiểu dữ liệu trừu tượng với giao diện được xác định rõ
    - Sử dụng chương trình phòng thủ; tránh tiếp xúc đại diện
  - Đặc tả (Specification)
    - Viết thông số kỹ thuật cho tất cả các mô-đun để có một hợp đồng rõ ràng, được xác định rõ ràng giữa mỗi mô-đun và các khách hàng của nó



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

70

70

#### 4.2.3. Phòng thủ thứ ba: Khả năng hiển thị ngay lập tức

- ❑ Nếu chúng ta không thể ngăn lỗi, chúng ta có thể cố gắng bản địa hóa chúng thành một phần nhỏ của chương trình
  - ❑ Assertions: phát hiện lỗi sớm, trước khi chúng lây nhiễm và có thể được che giấu bằng cách tính toán thêm
  - ❑ Kiểm thử đơn vị: khi bạn kiểm tra riêng một mô-đun, bạn có thể tin rằng bất kỳ lỗi nào bạn tìm thấy là do lỗi trong đơn vị đó (trừ khi nó nằm trong trình điều khiển kiểm tra)
  - ❑ Kiểm thử hồi quy: chạy kiểm tra thường xuyên nhất có thể khi thay đổi mã. Nếu không thành công, rất có thể có lỗi trong mã bạn vừa thay đổi
- ❑ Khi bản địa hóa thành một phương thức hoặc mô-đun nhỏ, các kiểm khuyết thường có thể được tìm thấy đơn giản bằng cách nghiên cứu văn bản chương trình



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

72

18

### Lợi ích của việc hiển thị ngay lập tức

- ❖ Khó khăn chính của việc gỡ lỗi là tìm ra lỗi: đoạn mã chịu trách nhiệm cho một vấn đề được quan sát
  - Một phương thức có thể trả về một kết quả sai, nhưng bản thân nó không có lỗi, nếu có sự thay đổi trước đó về đại diện
- ❖ Sự cố được quan sát càng sớm thì càng dễ sửa chữa
  - Thường xuyên kiểm tra biến đại diện giúp
- ❖ Cách tiếp cận chung: không nhanh
  - Kiểm tra các biến, dừng chỉ giả định chúng
  - Dừng (thường) cố gắng khôi phục lỗi - nó có thể chỉ che giấu chúng



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

73

73

### Đừng che giấu lỗi

```
// k is guaranteed to be present in a
int i = 0;
while (true) {
    if (a[i]==k) break;
    i++;
}
```

- ❖ Đoạn mã này tìm kiếm một mảng a cho một giá trị k.
  - Giá trị được đảm bảo nằm trong mảng
  - Điều gì sẽ xảy ra nếu bảo đảm đó bị phá vỡ (do khiếm khuyết)?
- ❖ Cám dỗ: làm cho mã “mạnh mẽ hơn” bằng cách không thất bại



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

74

74

### Đừng che giấu lỗi

```
// k is guaranteed to be present in a
int i = 0;
while (i<a.length) {
    if (a[i]==k) break;
    i++;
}
assert (i==a.length) : "key not found";
```

- ❖ Các khẳng định cho phép chúng ta ghi lại và kiểm tra các biến
  - Hủy bỏ / gỡ lỗi chương trình ngay khi phát hiện sự cố: biến một lỗi thành một thất bại
- ❖ Nhưng xác nhận không được kiểm tra cho đến khi chúng ta sử dụng dữ liệu, có thể rất lâu sau lỗi ban đầu
  - “Tại sao không phải là khóa trong mảng?”



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

75

75

76

### Kiểm tra Production Code

- ❖ Bạn có nên đưa các xác nhận và kiểm tra vào mã sản xuất không?
  - Có: dùng chương trình nếu kiểm tra không thành công - không muốn để mất cơ hội chương trình sẽ làm sai
  - Không: có thể cần chương trình để tiếp tục, có thể lỗi không gây ra hậu quả xấu như vậy (lỗi có thể chấp nhận được)
  - Câu trả lời đúng phụ thuộc vào ngữ cảnh!



Ariane 5 - chương trình bị tạm dừng vì tràn giá trị chưa sử dụng, trường hợp ngoại lệ được ném ra nhưng không được xử lý cho đến cấp cao nhất, sự cố tên lửa... [mặc dù câu chuyện đầy đủ phức tạp hơn]

### 4.3. Debugging

- ❖ Bước 1 - tìm một trường hợp thử nghiệm nhỏ, có thể lặp lại tạo ra lỗi (có thể tốn nhiều công sức, nhưng giúp làm rõ lỗi và cũng cung cấp cho bạn một cái gì đó để hồi quy)
  - Đừng chuyển sang bước tiếp theo cho đến khi bạn có một test có thể lặp lại
- ❖ Bước 2 - thu hẹp vị trí và nguyên nhân lân cận
  - Nghiên cứu dữ liệu / giả thuyết / thử nghiệm / lặp lại
  - Có thể thay đổi mã để có thêm thông tin
  - Đừng chuyển sang bước tiếp theo cho đến khi bạn hiểu nguyên nhân
- ❖ Bước 3 - sửa lỗi
  - Đó có phải là lỗi đánh máy đơn giản, hay lỗi thiết kế? Nó có xảy ra ở nơi khác không?
- ❖ Bước 4 - thêm trường hợp thử nghiệm vào bộ hồi quy
  - Lỗi này có được khắc phục không? Có bất kỳ lỗi mới nào khác được giới thiệu không?

### 4.2.3. Gỡ lỗi: Phương sách cuối cùng

- ❖ Khiếm khuyết xảy ra - con người không hoàn hảo
  - Mức trung bình trong ngành: 10 lỗi trên 1000 dòng mã ("kloc")
- ❖ Các khiếm khuyết không thể bản địa hóa ngay lập tức xảy ra
  - Tìm thấy trong quá trình kiểm tra tích hợp
  - Hoặc do người dùng báo cáo
- ❖ Chi phí tìm và sửa lỗi thường tăng lên theo thứ tự độ lớn cho mỗi giai đoạn vòng đời mà nó đi qua
  - bước 1 - Làm rõ triệu chứng (đơn giản hóa đầu vào), tạo kiểm tra
  - bước 2 - Tìm và hiểu nguyên nhân, tạo thử nghiệm tốt hơn
  - bước 3 - Sửa chữa
  - bước 4 - Chạy lại tất cả các bài kiểm tra

### Gỡ lỗi và phương pháp khoa học

- ❖ Gỡ lỗi phải có hệ thống
  - Hãy cẩn thận quyết định xem phải làm gì - thất bại có thể là một trường hợp thất bại trong thử nghiệm
  - Ghi lại mọi thứ bạn làm
  - Đừng để bị hút vào những con đường không có kết quả
- ❖ Hình thành giả thuyết
- ❖ Thiết kế một thử nghiệm
- ❖ Thực hiện thử nghiệm
- ❖ Điều chỉnh giả thuyết của bạn và tiếp tục



### Ví dụ về giảm kích thước đầu vào

```
// returns true iff sub is a substring of full  
// (i.e. iff there exists A,B s.t. full=A+sub+B)  
boolean contains(String full, String sub);
```

- ❖ Báo cáo lỗi người dùng
  - không thể tìm thấy chuỗi "very happy" trong:  
"Fáilte, you are very welcome! Hi Seán! I am very very happy to see you all."
- ❖ Ít phản hồi lý tưởng
  - Xem các ký tự có dấu, về việc không nghĩ đến unicode và tìm hiểu các văn bản Java của bạn để xem cách xử lý
  - Cố gắng theo dõi việc thực hiện ví dụ này
- ❖ Phản hồi tốt hơn: đơn giản hóa / làm rõ các triệu chứng



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

81

81

### Giảm kích thước đầu vào tương đối

- ❖ Đôi khi sẽ rất hữu ích nếu bạn tìm thấy hai trường hợp thử nghiệm gần như giống hệt nhau, trong đó một trường hợp đưa ra câu trả lời chính xác và trường hợp còn lại thì không
  - Can't find "very happy" within
    - "I am very very happy to see you all."
  - Can find "very happy" within
    - "I am very happy to see you all."



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

83

83

### Giảm kích thước đầu vào tuyệt đối

- ❖ Tìm một trường hợp thử nghiệm đơn giản bằng cách chia để trị
- ❖ Pare test down – can't find "very happy" within
  - "Fáilte, you are very welcome! Hi Seán! I am very very happy to see you all."
  - "I am very very happy to see you all."
  - "very very happy"
- ❖ Can find "very happy" within
  - "very happy"
- ❖ Can't find "ab" within "aab"



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

82

82

### Chiến lược chung: đơn giản hóa

- ❖ Nói chung: tìm đầu vào đơn giản nhất sẽ dẫn đến thất bại
  - Thường không phải là đầu vào tiết lộ sự tồn tại của khuyết
- ❖ Bắt đầu với dữ liệu tiết lộ lỗi
  - Tiếp tục phân tích nó (tim kiém nhị phân "bởi bạn" có thể giúp ích)
  - Thường dẫn trực tiếp đến sự hiểu biết về nguyên nhân
- ❖ Khi không xử lý các cuộc gọi phương thức đơn giản
  - "Đầu vào kiểm tra" là tập hợp các bước gây ra lỗi một cách đáng tin cậy
  - Cùng một ý tưởng cơ bản



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

84

84

## Bản địa hóa một khuyết điểm

### ☐ Tận dụng tính mô-đun

- ☐ Bắt đầu với mọi thứ, lấy đi từng phần cho đến khi thất bại
- ☐ Bắt đầu mà không có gì, thêm các mảnh trở lại cho đến khi thất bại xuất hiện

### ☐ Tận dụng lý luận mô-đun

- ☐ Theo dõi chương trình, xem kết quả trung gian

### ☐ Tìm kiếm nhị phân tăng tốc quá trình

- ☐ Lỗi xảy ra ở đâu đó giữa câu lệnh đầu tiên và câu lệnh cuối cùng
- ☐ Thực hiện tìm kiếm nhị phân trên tập hợp các câu lệnh có thứ tự đó



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

85

85

## Tìm kiếm nhị phân trên mã lỗi

```
public class MotionDetector {  
    private boolean first = true;  
    private Matrix prev = new Matrix();  
  
    public Point apply(Matrix current) {  
        if (first) {  
            prev = current;  
        }  
        Matrix motion = new Matrix();  
        getDifference(prev, current, motion);  
        applyThreshold(motion, motion, 10);  
        labelImage(motion, motion);  
        Hist hist = getHistogram(motion);  
        int top = hist.getMostFrequent();  
        applyThreshold(motion, motion, top, top);  
        Point result = getCentroid(motion);  
        prev.copy(current);  
        return result;  
    }  
}
```



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

86

86

## Tìm kiếm nhị phân trên mã lỗi

```
public class MotionDetector {  
    private boolean first = true;  
    private Matrix prev = new Matrix();  
  
    public Point apply(Matrix current) {  
        if (first) {  
            prev = current;  
        }  
        Matrix motion = new Matrix();  
        getDifference(prev, current, motion);  
        applyThreshold(motion, motion, 10);  
        labelImage(motion, motion);  
        Hist hist = getHistogram(motion);  
        int top = hist.getMostFrequent();  
        applyThreshold(motion, motion, top, top);  
        Point result = getCentroid(motion);  
        prev.copy(current);  
        return result;  
    }  
}
```



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

87

87

## Phát hiện lỗi trong thế giới thực

### ❖ Hệ thống thực

- Lớn và phức tạp (duh!)
- Bộ sưu tập các mô-đun, được viết bởi nhiều người
- Đầu vào phức tạp
- Nhiều tương tác bên ngoài
- Không xác định

### ❖ Sao chép có thể là một vấn đề

- Không thường xuyên

### ❖ Thiết bị đo lường loại bỏ lỗi

- Khiếm khuyết các rào cản trừu tượng chéo
- Độ trễ thời gian lớn từ corruption (defect) đến phát hiện (failure)



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

88

88

## Heisenbugs

- ❖ Chương trình tuần tự, xác định - lỗi có thể lặp lại
- ❖ Nhưng thế giới thực không tốt đẹp như vậy...
  - Thay đổi đầu vào / môi trường liên tục
  - Thời gian phụ thuộc
  - Đồng thời và song song
- ❖ Thất bại xảy ra ngẫu nhiên
- ❖ Khó tái tạo
  - Sử dụng trình gỡ lỗi hoặc xác nhận → lỗi sẽ biến mất
  - Chỉ xảy ra khi chịu tải nặng
  - Chỉ thỉnh thoảng xảy ra một lần

## Thủ thuật cho lỗi khó

- ❖ Xây dựng lại hệ thống từ đầu hoặc khởi động lại / khởi động lại
  - Tìm lỗi trong hệ thống xây dựng hoặc cấu trúc dữ liệu liên tục của bạn
- ❖ Giải thích vấn đề cho một người bạn
- ❖ Đảm bảo rằng đó là một lỗi - chương trình có thể đang hoạt động chính xác mà bạn không nhận ra!
- ❖ Giảm thiểu đầu vào được yêu cầu để thực hiện lỗi (có lỗi)
- ❖ Thêm séc vào chương trình
  - Giảm thiểu khoảng cách giữa lỗi và phát hiện / lỗi
  - Sử dụng tìm kiếm nhị phân để thu hẹp các vị trí có thể
- ❖ Sử dụng nhật ký để ghi lại các sự kiện trong lịch sử

## Ghi nhật ký sự kiện

- ❖ Xây dựng nhật ký sự kiện (bộ đệm tròn) và ghi các sự kiện trong quá trình thực thi chương trình khi chương trình chạy ở tốc độ
- ❖ Khi phát hiện lỗi, hãy dừng chương trình và kiểm tra nhật ký để giúp bạn xây dựng lại quá khứ
- ❖ Nhật ký có thể là tất cả những gì bạn biết về môi trường của khách hàng - giúp bạn tái tạo lỗi

## Lỗi ở đâu?

- Nếu lỗi không ở nơi bạn nghĩ, hãy tự hỏi mình xem nó không thể ở đâu; giải thích vì sao
- Trước tiên, hãy tìm những sai lầm ngớ ngẩn, ví dụ:
  - Reversed order of arguments: `Collections.copy(src, dest)`
  - Spelling of identifiers: `int hashCode()`
    - `@Override` can help catch method name typos
  - Same object vs. equal: `a == b` versus `a.equals(b)`
  - Failure to reinitialize a variable
  - Deep vs. shallow copy
- Đảm bảo rằng bạn có mã nguồn chính xác
  - Biên dịch lại mọi thứ

## Khi mọi việc trở nên khó khăn

- ❖ Xem xét lại các giả định
  - Ví dụ: hệ điều hành có thay đổi không? Có chỗ trên ổ cứng không?
  - Gỡ lỗi mã, không phải nhận xét - đảm bảo nhận xét và thông số kỹ thuật mô tả mã
- ❖ Bắt đầu ghi lại hệ thống của bạn
  - Cung cấp một góc mới và làm nổi bật khu vực nhầm lẫn
- ❖ Được trợ giúp
  - Tất cả chúng ta đều phát triển những điểm mù
  - Giải thích vấn đề thường giúp
- ❖ Bỏ đi
  - Độ trễ giao dịch để đạt hiệu quả - đi ngủ!
  - Một lý do tốt để bắt đầu sớm



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

93

93

## Công cụ hỗ trợ: Trình cắm thêm Eclipse

- ❖ Checkstyle: Giúp lập trình viên viết mã Java tuân theo tiêu chuẩn viết mã
- ❖ FindBugs: Sử dụng phân tích tĩnh để tìm lỗi trong mã Java
  - Standalone Swing application
  - Eclipse plug-in
  - Integrated into the build process (Ant or Maven)



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

95

95

## Các khái niệm chính được xem xét

- ❖ Kiểm tra và gỡ lỗi khác nhau
  - Kiểm tra cho thấy sự tồn tại của các lỗi
  - Gỡ lỗi xác định vị trí chính xác của các lỗi
- ❖ Mục tiêu là làm cho chương trình phù hợp
- ❖ Gỡ lỗi phải là một quá trình có hệ thống
  - Sử dụng phương pháp khoa học
- ❖ Hiểu nguồn gốc của khuyết tật
  - Để tìm những cái tương tự và ngăn chặn chúng trong tương lai



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

94

94

## Các tính năng của Findbugs

- ❖ Không quan tâm đến các tiêu chuẩn định dạng hoặc mã hóa
- ❖ Phát hiện các lỗi tiềm ẩn và các vấn đề về hiệu suất
  - Có thể phát hiện nhiều loại lỗi phổ biến, khó tìm
  - Sử dụng "bug patterns"

NullPointerException

```
Address address = client.getAddress();
if ((address != null) || (address.getPostCode() != null)) {
    ...
}
```

Uninitialized field

```
public class ShoppingCart {
    private List items;
    public addItem(Item item) {
        items.add(item);
    }
}
```



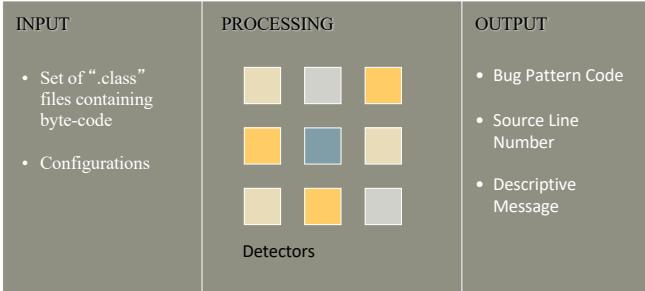
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

96

96

24

## Findbug hoạt động như thế nào?



## Finbugs có thể làm gì?

❖ FindBugs đi kèm với hơn 200 quy tắc được chia thành các loại khác nhau:

- Tính đúng đắn
  - Ví dụ. vòng lặp đệ quy vô hạn, đọc một trường không bao giờ được ghi
- Thực hành không tốt
  - Ví dụ. mã giảm ngoại lệ hoặc không đóng được tệp
- Hiệu suất
  - Độ đúng đa luồng
- Tinh ranh
  - Ví dụ. biến cục bộ không sử dụng hoặc phải không được kiểm tra

Thank you  
for your  
attentions!

