Bài 7
KIỂM THỬ

Trịnh Thành Trung
trungtt@soict.hust.edu.vn

1

KHÁI NIỆM

Mục đích

- Khó có thể khẳng định 1 chương trình lớn có làm việc chuẩn hay không
- Khi xây dựng 1 chương trình lớn, 1 lập trình viên chuyên nghiệp sẽ dành thời gian cho việc viết test code không ít hơn thời gian dành cho viết bản thân chương trình
- Lập trình viên chuyên nghiệp là người có khả năng, kiến thức rộng về các kỹ thuật và chiến lược testing

Testing and debugging

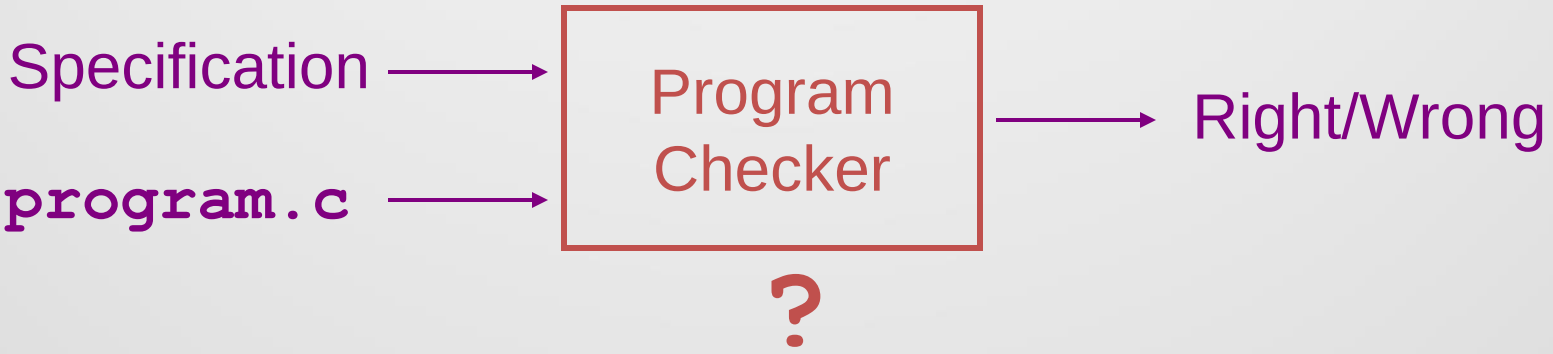
- **Test & debug đi cùng với nhau như 1 cặp:**
  - Testing tìm error; debug định vị và sửa chúng.
  - Ta có mô hình "testing/debugging cycle": Ta test, rồi debug, rồi lặp lại.
  - Bất kỳ 1 debugging nào nên được tiếp theo là 1 sự áp dụng lại của hàng loạt các test liên quan, đặc biệt là các bài test hồi quy. Điều này giúp tránh nảy sinh các lỗi mới khi debug.
  - Test & debug không nên được thực hiện bởi cùng 1 người.

Khái niệm Testing

- Beizer: Việc thực hiện test là để chứng minh tính đúng đắn giữa 1 phần tử và các đặc tả của nó.
- Myers: Là quá trình thực hiện 1 chương trình với mục đích tìm ra lỗi.
- IEEE: Là quá trình kiểm tra hay đánh giá 1 hệ thống hay 1 thành phần hệ thống một cách thủ công hay tự động để kiểm chứng rằng nó thỏa mãn những yêu cầu đặc thù hoặc để xác định sự khác biệt giữa kết quả mong đợi và kết quả thực tế

Program Verification

- Lý tưởng: Chứng minh được rằng chương trình của ta là chính xác, đúng đắn
  - Có thể chứng minh các thuộc tính của chương trình?
  - Có thể chứng minh điều đó kể cả khi chương trình kết thúc?



Program Testing

- Hiện thực: Thuyết phục bản thân rằng chương trình có thể làm việc

2

PHÂN LOẠI

External vs. Internal Testing

- Các loại testing
  - External testing
    - Thiết kế dữ liệu để test chương trình
  - Internal testing
    - Thiết kế program để chương trình tự test

External Testing

- External testing: TK dữ liệu để test chương trình
- Phân loại : External testing
  (1) Kiểm chứng giá trị biên: Boundary testing
  (2) Kiểm chứng lệnh: Statement testing
  (3) Kiểm chứng có hệ thống: Path testing
  (4) Kiểm chứng tải: Stress testing

Boundary Testing

(1) Boundary testing

– “Là kỹ thuật kiểm chứng sử dụng các giá trị nhập vào ở trên hoặc dưới một miền giới hạn của 1 đầu vào và với các giá trị đầu vào tạo ra các đầu ra ở biên của 1 đầu ra."

- Hầu hết các lỗi đều xảy ra ở các điều kiện biên - boundary conditions
- Nếu chương trình làm việc tốt ở đk biên, nó có thể sẽ làm việc đúng với các đk khác

Boundary Testing Example

- VD: đọc 1 dòng từ stdin và đưa vào mảng ký tự

```
int i;
char s[MAXLINE];
for (i=0; (s[i]=getchar()) != '\n' && i < MAXLINE-1; i++);
s[i] = '\0';
printf("String: |%s|\n", s);
```

- Xét các điều kiện biên
  - Dòng rỗng –bắt đầu với '\n'
    - In ra xâu rỗng (“\0”) => in ra “||” , ok
  - Nếu gặp EOF trước '\n‘
    - Tiếp tục gọi getchar() và lưu ÿ vào s[i], not ok
  - Nếu gặp ngay EOF (empty file)
    - Tiếp tục gọi getchar() và lưu ÿ vào s[i], not ok

Boundary Testing Example (cont.)

```
int i;
char s[MAXLINE];
for (i=0; (s[i]=getchar()) != '\n' && i < MAXLINE-1; i++);
s[i] = '\0';
printf("String: |%s|\n", s);
```

- Tiếp tục xét các ĐK biên (tt)
  - Dòng chứa đúng MAXLINE-1 ký tự
    - In ra đúng, với '\0' tại s[MAXLINE-1]
  - Dòng chứa đúng MAXLINE ký tự
    - Ký tự cuối cùng bị ghi đè, và dòng mới không bao giờ được đọc
  - Dòng dài hơn MAXLINE ký tự
    - 1 số ký tự, kể cả newline, không được đọc và sót lại trong stdin

Boundary Testing Example (cont.)

- Viết lại code

```
int i;
char s[MAXLINE];
for (i=0; i< MAXLINE-1; i++)
    if ((s[i] = getchar()) == '\n')
        break;
s[i] = '\0';
```

- Trường hợp gặp EOF

```
for (i=0; i<MAXLINE-1; i++)
    if ((s[i] = getchar()) == '\n' || s[i] == EOF)
        break;
s[i] = '\0';
```

- Các trường hợp khác?
  - Nearly full
  - Exactly full
  - Over full

SAI!
Vì sao?

Boundary Testing Example (cont.)

- **Rewrite yet again**

```
for (i=0; ; i++) {
   int c = getchar();
   if (c==EOF || c=='\n' || i==MAXLINE-1) {
               s[i] = '\0';
     break;
   }
   else s[i] = c;
}
```

- Vấn đề

(với MAXLINE=9)

Input:

```
Four
score and seven
years
```

Output:

```
FourØ
score anØ
 sevenØ
yearsØ
```

Where's the 'd'?

Ambiguity in Specification

- Nếu dòng quá dài, xử lý thế nào?
  - Giữ MAXLINE ký tự đầu, bỏ qua phần còn lại?
  - Giữ MAXLINE ký tự đầu + '\0', bỏ qua phần còn lại?
  - Giữ MAXLINE ký tự đầu+'\0', lưu phần còn lại cho lần gọi sau của input function?
- Có thể phần đặc tả - specification không hề đề cập khi MAXLINE bị quá
  - Có thể người ta không muốn dòng dài quá giới hạn trong mọi trường hợp
  - Đạo đức: kiểm tra đã phát hiện ra một vấn đề thiết kế, thậm chí có thể là một vấn đề đặc điểm kỹ thuật !
- Quyết định phải làm gì
  - Cắt những dòng quá dài?
  - Lưu phần còn lại để đọc như 1 dòng mới?

Kiểm tra đk trước và đk sau

- Xác định những thuộc tính cần đi trước (đk trước) và sau (đk sau) mã nguồn đc thi hành
- Ví dụ: các giá trị đầu vào phải thuộc 1 phạm vi cho trước

```
double avg( double a[], int n) {
  int i; double sum=0.0;
  for ( i = 0; i<n; i++)
    sum+=a[i];
  return sum/n;
}
```

Nếu n=0 ?, nếu n<0 ?

Có thể thay : return n <=0 ? 0.0: sum/n;

Tháng 11/1998, chiến hạm Yorktown bị chìm : nhập vào giá trị 0, chương trình không kiểm tra dl nhập dẫn đến chia cho 0, và lỗi làm tầu rối loạn, hệ thống đẩy ngưng hoạt động, tàu chìm !!!

Statement Testing

(2) Statement testing

- "Testing để thỏa mãn điều kiện rằng mỗi lệnh trong 1 chương trình phải thực hiện ít nhất 1 lần khi testing."

  — Glossary of Computerized System and Software Development Terminology

Statement Testing Example

- **Example pseudocode:**

```
if (condition1)
   statement1;
else
   statement2;
if (condition2)
   statement3;
else
   statement4;
…
```

Statement testing:

Phải chắc chắn các lệnh “if” và 4 lệnh trong cac nhánh phải đc thực hiện

- **Đòi hỏi 2 tập dữ liệu;vd:**
  - *condition1* là đúng và *condition2* là đúng
    - Thực hiện *statement1* và *statement3*
  - *condition1* là sai và *condition2* là sai
    - Thực hiện *statement2* và *statement4*

Path Testing

(3) Path testing

- "Kiểm tra để đáp ứng các tiêu chuẩn đảm bảo rằng mỗi đường dẫn logic xuyên suốt chương trình được kiểm tra. Thường thì đường dẫn xuyên suốt chương trình này được nhóm thành một tập hữu hạn các lớp . Một đường dẫn từ mỗi lớp sau đó được kiểm tra. "

— Glossary of Computerized System and Software Development Terminology

- Khó hơn nhiều so với statement testing
  - Với các chương trình đơn giản, có thể liệt kê các nhánh đường dẫn xuyên suốt code
  - Ngược lại, bằng các đầu vào ngẫu nhiên tạo các đường dẫn theo chương trình

Path Testing Example

- Example pseudocode:

```
if (condition1)
   statement1;
else
   statement2;
…
if (condition2)
   statement3;
else
   statement4;
…
```

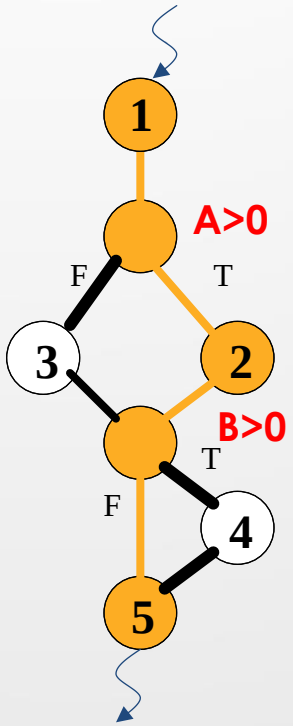
Path testing:

Cần đảm bảo tất cả các đường dẫn được thực hiện

- Đòi hỏi 4 tập dữ liệu:
  - *condition1* là true và *condition2* là true
  - *condition1* là true và *condition2* là false
  - *condition1* là false và *condition2* là true
  - *condition1* là false và *condition2* la false
- Chương trình thực tế => bùng nổ các tổ hợp!!!

Consider an example…

```
(1)  input(A,B)
     if (A>0)
(2)     Z = A;
     else
(3)     Z = 0;
     if (B>0)
(4)     Z = Z+B;
(5)  output(Z)
```



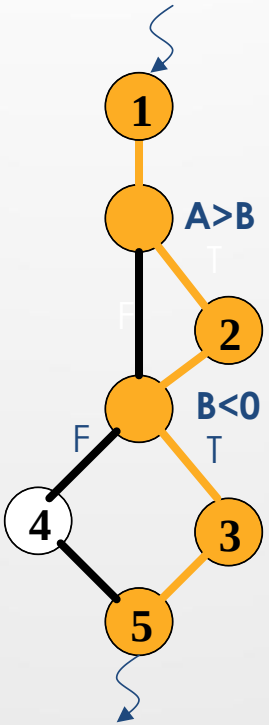
What is the path condition for path **<1,2,5>?**

(A>0) && (B≤0)

Consider ANOTHER example…

```
(1)  input(A,B)
     if (A>B)
(2)   B = B*B;
      if (B<0)
(3)     Z = A;
     else
(4)     Z = B;

(5)  output(Z)
```



What is the path condition for path **<1,2,3,5>?**

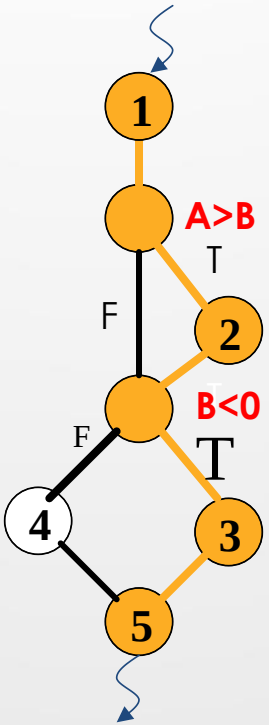
(A>B) && (B<0)

Consider ANOTHER example…

```
(1)  input(A,B)
     if (A>B)
(2)     B = B*B;

     if (B<0)
(3)     Z = A;
     else
(4)     Z = B;

(5)  output(Z)
```



What is the path condition for path <1,2,3,5>?

$(A>B) \wedge (B<0) \; (B^2<0) = \text{FALSE}$

Stress Testing

(4) Stress testing

- "Tiến hành thử nghiệm để đánh giá một hệ thống hay thành phần tại hoặc vượt quá các giới hạn của các yêu cầu cụ thể của nó"
  - Glossary of Computerized System and Software Development Terminology

- Phải tạo :
  - Một tập lớn đầu vào - Very large inputs
  - Các đầu vào ngẫu nhiên - Random inputs (binary vs. ASCII)
- Nên dùng máy tính để tạo đầu vào

Stress Testing Example 1

- Example program:

```
#include <stdio.h>
int main(void) {
    char c;
    while ((c = getchar()) != EOF)
        putchar(c);
    return 0;
}
```

Stress testing: Phải cung cấp **random** (binary and ASCII) inputs

- Mục tiêu: Copy tất cả các ký tự từ stdin vào stdout; nhưng lưu ý bug!!!
- Làm việc với tập dữ liệu ASCII chuẩn (tự tạo)
- Máy tính tự tạo ngẫu nhiên tập dữ liệu dạng 255 (decimal), hay 11111111 (binary), và EOF để dừng vòng lặp

Stress Testing Example 2

- Example program:

```
#include <stdio.h>
int main(void) {
    short charCount = 0;
    while (getchar() != EOF)
        charCount++;
    printf("%hd\n", charCount);
    return 0;
}
```

Stress testing: Phải cung cấp **very large** inputs

- Mục tiêu: Đếm và in số các kỹ tự trong stdin
- Làm việc với tập dữ lieu có kích thước phù hợp
- Sẽ có lỗi với tập dữ liệu do máy tạo chứa hơn 32767 characters

Uses of assert

- Typical uses of **`assert`**
  - Validate formal parameters

```
size_t Str_getLength(const char *str) {
   assert(str != NULL);
   …
}
```

  - Check for "impossible" logical flow

```
switch (state) {
   case START: … break;
   case COMMENT: … break;
   …
   default: assert(0); /* Never should get here */
}
```

  - Make sure dynamic memory allocation requests worked
    assert(ptr != NULL);

Internal Testing

- Internal testing: Thiết kế chương trình để chương trình tự kiểm thử
- Internal testing techniques
  - (1) Kiểm tra bất biến - Testing invariants
  - (2) Kiểm tra các thuộc tính lưu trữ -Verifying conservation properties
  - (3) Kiểm tra các giá trị trả về -Checking function return values
  - (4) Tạm thay đổi code -Changing code temporarily
  - (5) Giữ nguyên mã thử nghiệm -Leaving testing code intact

Testing Invariants

(1) Testing invariants

- Thử nghiệm các đk trước và sau
- Vài khía cạnh của cấu trức dữ liệu không đc thay đổi
- 1 hàm tác động đến cấu trúc dữ liệu phải kiểm tra các bất biến ở đầu và cuối nó

- Ví dụ: Hàm "doubly-linked list insertion"
  - Kiểm tra ở đầu và cuối
    - Xoay doubly-linked list
    - Khi node x trỏ ngược lại node y, thì liệu node y có trỏ ngược lại node x?

- Example: "binary search tree insertion" function
  - Kiểm tra ở đầu và cuối
    - Xoay tree
    - Các nodes có còn đc sắp xếp không ?

Testing Invariants (cont.)

- Tiện cho việc dùng **`assert`** để test invariants

```
#ifndef NDEBUG
int isValid(MyType object) {
    …
    Test invariants here.
    Return 1 (TRUE) if object passes
    all tests, and 0 (FALSE) otherwise.
    …
}
#endif

void myFunction(MyType object) {
    assert(isValid(object));
    …
    Manipulate object here.
    …
    assert(isValid(object));
}
```

Có thể dùng NDEBUG trong code, giống như **`assert`**

Kiểm tra các thuộc tính lưu trữ

- Khái quát hóa của testing invariants
- 1 hàm cần kiểm tra các cấu trúc dữ liệu bị tác động tại các điểm đầu và cuối

- VD: hàm **`Str_concat()`**
  - Tại điểm đầu, tìm độ dài của 2 xâu đã cho; tính tổng
  - Tại điểm cuối, tìm độ dài của xâu kết quả
  - 2 độ dài có bằng nhau không ?

- VD: Hàm chèn thêm PT vào danh sách -List insertion function
  - Tại điểm khởi đầu, tính độ dài ds
  - Tại điểm cuối, Tính độ dài mới
  - Độ dài mới = độ dài cũ + 1?

Kiểm tra GT trả về Checking Return Values

- Trong Java và C++:
  - Phương thức bị phát hiện có lỗi có thể tung ra một “checked exception"
  - Phương thưc triệu gọi phải xử lý ngoại lệ
- Trong C:
  - Không có cơ chế xử lý exception
  - Hàm phát hiện có lỗi chủ yếu thông qua giá trị trả về
  - Người LT thường dễ dàng quên kiểm tra GT trả về
  - Nói chung là chúng ta nên kiểm tra GT trả về

Checking Return Values (cont.)

– VD: **`scanf()`** trả về số của các giá trị đc đọc

Bad code

```
int i;
scanf("%d", &i);
```

Good code

```
int i;
if (scanf("%d", &i) != 1)
   /* Error */
```

Bad code???

```
int i = 100;
printf("%d", i);
```

Good code, or overkill???

```
int i = 100;
if (printf("%d", i) != 3)
   /* Error */
```

– VD: **`printf()`** có thể bị lỗi nếu ghi ra file và đĩa bị đầy; Hàm này trả về số ký tự được ghi (không phải giá trị)

Tạm thay đổi code

- Tạm thay đổi code để tạo ranh giới nhân tạo hoặc stress tests
- VD: chương trình sắp xếp trên mảng
  - Tạm đặt kích thước mảng nhỏ
  - chương trình có xử lý tràn số hay không ?
- Viết 1 phiên bản hàm cấp phát bộ nhớ và phát hiện ra lỗi sớm, để kiểm chứng đoạn mã nguồn bị lỗi thiếu bộ nhớ :

```
void *testmalloc( size_t n) {
  static int count =0;
   if (++count  > 10)
       return NULL;
  else
       return malloc(n);
}
```

Để nguyên đoạn code kiểm tra

- Hãy để nguyên trạng các đoạn kiểm tra trên code
- Có thể khoanh lại = **`#ifndef NDEBUG`** … **`#endif`**
- Kiểm tra với tùy chọn –DNDEBUG gcc
  - Bặt/Tắt assert macro
  - Cũng có thể Bật/tắt debugging code

- **Cẩn trọng với mâu thuẫn - conflict:**
  - Mở rộng thử nghiệm nội bộ có thể giảm chi phí bảo trì
  - Code rõ ràng có thể giảm chi phí bảo trì
  - Nhưng ... Mở rộng thử nghiệm nội bộ có thể làm giảm độ rõ ràng của Code !

3

CÁC CHIẾN LƯỢC KIỂM THỬ

Các chiến lược testing

- General testing strategies
  (1) Kiểm chứng tăng dần -Testing incrementally
  (2) So sánh các cài đặt -Comparing implementations
  (3) Kiểm chứng tự động - Automation
  (4) Bug-driven testing
  (5) Tiêm, gài lỗi - Fault injection

Testing Incrementally

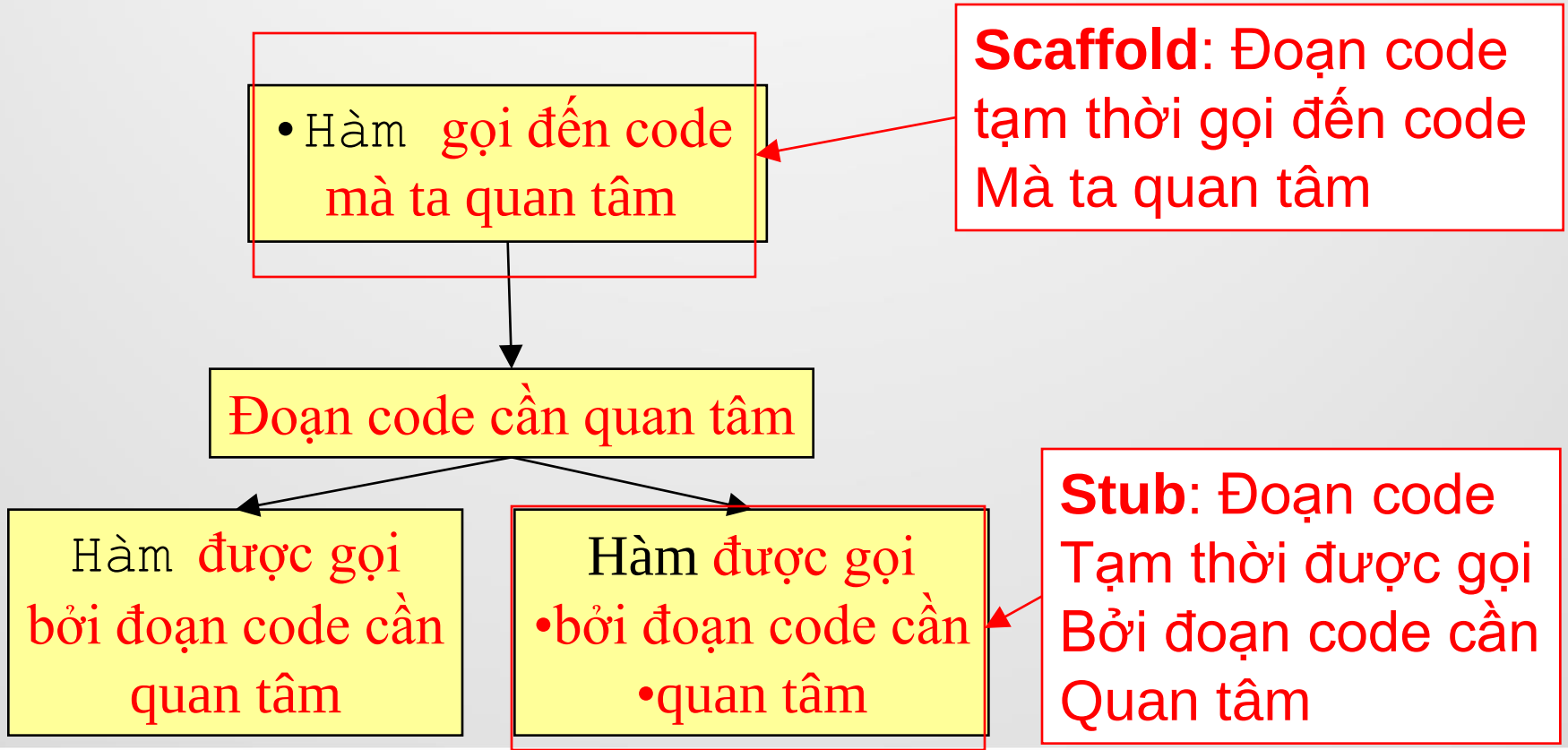
(1) Testing incrementally

- Test khi viết code
  - Thêm test khi tạo 1 lựa chọn mới - new cases
  - Test phần đơn giản trước phần phức tạp
  - Test units (tức là từng module riêng lẻ) trước khi testing toàn hệ thống
- Thực hiện regression testing – kiểm thử hồi quy
  - Xử lý đc 1 lỗi thường tạo ra những lỗi mới trong 1 hệ thống lớn, vì vậy …
  - Phải đảm bảo chắc chắn hệ thống không "thoái lui" kiểu như chức năng trước kia đang làm việc giờ bị broken, nên…
  - Test mọi khả năng để so sanh phiên bản mới với phiên bản cũ

Testing Incrementally (cont.)

(1) Testing incrementally (cont.)

– Tạo "giàn giáo" - scaffolds và "mẫu" -stubs để test đoạn code mà ta quan tâm



So sánh các cài đặt

- (2) Compare implementations
  - Hãy chắc chắn rằng các triển khai độc lập hoạt động như nhau
  - Example: So sánh hành vi của chương trình mà bạn dịch (TB C++3.0) với GCC
  - Example: So sánh hành vi của các hàm bạn tạo trong str.h với các hàm trong thư viện string.h
  - Đôi khi 1 kết quả có thể đc tính bằng 2 cách khác nhau, 1 bài toán có thể giải bằng 2 phương pháp, thuật toán khác nhau. Ta có thể xây dựng cả 2 chương trình, nếu chúng có cùng kết quả thì có thể khẳng định cả 2 cùng đúng, còn kết quả khác nhau thì ít nhất 1 trong 2 chương trình bị sai.

Kiểm chứng tự động - Automation

(3) Automation

- Là quá trình xử lý 1 cách tự động các bước thực hiện test case = 1 công cụ nhằm rut ngắn thời gian kiểm thử.
- Ba quá trình kiểm chúng bao gồm :
  - Thực hiện kiểm chứng nhiều lần
  - Dùng nhiều bộ dữ liệu nhập
  - Nhiều lần so sánh dữ liệu xuất
- vì vậy cần kiểm chứng = chương trình để : tránh mệt mỏi, giảm sự bất cẩn …
- Tạo testing code
  - Viết 1 bộ kiểm chứng để kiểm tra toàn bộ chương trình mỗi khi có sự thay đổi, sau khi biên dịch thành công
- Cần biết cái gì được chờ đợi
  - Tạo ra các đầu ra, sao cho dễ dàng nhận biết là đúng hay sai

- Tự động kiểm chứng có thể cung cấp:
  - **Tốt hơn nhiều** so với kiểm chứng thủ công

- Tự động hóa kiểm chứng lùi
  - Tuần tự kiểm chứng so sánh các phiên bản mới với những phiên bản cũ tương ứng.
  - Mục đích : đảm bảo việc sửa lỗi sẽ không làm ảnh hưởng những phần khác trừ khi chúng ta muốn
  - 1 số hệ thống có công cụ trợ giúp kiểm chứng tự động :
    - Ngôn ngữ scripts : cho phép viết các đoạn script để test tuần tự
    - Unix : có các thao tác trên tệp tin như cmp và diff để so sanh dữ liệu xuất, sort sắp xếp các phần tử, grep để kiểm chứng dữ liệu xuất, wc, sum va freq để tổng kết dữ liệu xuất
  - Khi kiểm chứng lùi, cần đảm bảo phiên bản cũ là đúng, nếu sai thì rất khó xác định và kết quả sẽ không chính xác
  - Cần phải kiểm tra chính việc kiểm chứng lùi 1 cách định kỳ để đảm bảo nó vẫn hợp lệ

- Dùng các công cụ test
  - QuickTest professional
  - TestMaker
  - Rational Robot
  - Jtest
  - Nunit
  - Selenium
  - ….

Bug-Driven Testing

(4) Kiểm chứng hướng lỗi : Bug-driven testing

- Tìm thấy 1 bug => Ngay lập tức tạo 1 test để bắt lỗi
- Đơn giản hóa việc kiểm chứng lùi

(5) Fault injection

- Chủ động (tạm thời) cài các bugs!!!
- Rồi quyết định nếu tìm thấy chúng
- Kiểm chứng bản thân kiểm chứng!!!

4

MỘT SỐ KỸ THUẬT KIỂM THỬ

Ai test cái gì - Who Tests What

- Programmers
  - White-box testing
  - Ưu điểm: Người triển khai nắm rõ mọi luồng dữ liệu
  - Nhược: Bị ảnh hưởng bởi cách thức code đc thiết kê/viết
- Quality Assurance (QA) engineers
  - Black-box testing
  - Pro: Không có khái niệm về implementation
  - Con: Không muốn test mọi logical paths
- Customers
  - Field testing
  - Pros: Có các cách sử dụng chương trình bất ngờ;dễ gây lỗi
  - Cons: Không đủ trường hợp; khách hàng không thích tham gia vào quá trình test ;

Testing Techniques

- **Black-Box:** Testing chỉ dựa trên việc phân tích các yêu cầu - requirements (unit/component specification, user documentation, v.v.). Còn được gọi là *functional testing.*
- **White-Box:** Testing dựa trên việc phân tích các logic bên trong - internal logic (design, code, v.v.). (Nhưng kết quả mong đợi vẫn đến từ requirements.) Còn đc gọi là *structural testing.*

WBT

- Còn được gọi là clear box testing, glass box testing, transparent box testing, or structural testing, thường thiết kế các trường hợp kiểm thử dựa vào cấu trúc bên trong của phần mềm.
- WBT đòi hỏi kĩ thuật lập trình am hiểu cấu trúc bên trong của phần mềm (các đường, luồng dữ liệu, chức năng, kết quả).
- Phương thức :
  - Chọn các đầu vào và xem các đầu ra

White Box Testing
Đặc điểm

- Phụ thuộc vào các cài đặt hiện tại của hệ thống và của phần mềm, nếu có sự thay đổi thì các bài test cũng cần thay đổi theo.
- Được ứng dụng trong các kiểm tra ở cấp độ mô đun (điển hình) , tích hợp (có khả năng) và hệ thống của quá trình test phần mềm.

Black Box Testing

- Black-box testing sử dụng mô tả bên ngoài của phần mềm để kiểm thử, ,bao gồm các đặc tả (specifications), yêu cầu (requirements) và thiết kế (design).
- Không có sự hiểu biết cấu trúc bên trong của phần mềm
- Các dạng đầu vào có dạng hàm hoặc không , hợp lệ và không không hợp lệ và biết trước đầu hợp lệ và không không hợp lệ và biết trước đầu ra
- Được sử dụng để kiểm thử phần mềmtại mức : mô đun, tích hợp, hàm, hệ thống và chấp nhận.
- Lợi điểm của kiểm thử hộp đen là khả năng đơn giản hoá kiểm thử tại các mức độ được đánh giá là khó kiểm thử
- Yếu điểm là khó đánh giá còn bộ giá trị nào chưa được kiểm thử hay không

- Các kí thuật chính của kiểm thử hộp đen :
- Decision Table testing
- Pairwise testing
- State transition tables
- Tests of Customer Requirement
- Equivalence partitioning
- Boundary value analysis
- Failure Test Cases

Grey Box Testing

- Là sự kết hợp của kiểm thử hộp đen và kiểm thử hộp trắng khi mà người kiểm thử biết được một phần cấu trúc bên trong của phần mềm
- Như vậy không phải là KT hộp đen
- Là dạng kiểm thử tốt và có sự kết hợp các kĩ thuật của cả kiểm thử hộp đen và các kĩ thuật của cả kiểm thử hộp đen và hộp trắng

Other Test

- Sanity testing
- Smoke testing
- Software testing
- Stress testing
- Test automation
- Web Application Security Scanner
- Fuzzing
- Acceptance testing
- Sanwich Testing

Sanity testing

- Kiểm thử đúng đắn là kiểm thử lướt qua; được thực hiện khi có đủ điều kiện nâng cao ứng dụng nhờ việc hàm hoá đặc tả.
- Kiểm thử các thành phần thoái hoá của phần mềm.
  - Bao gồm các kiểm thử vào vùng lõi của các
  - Bao gồm các kiểm thử vào vùng lõi của các hàm GUI cơ sở để xem kết nối dữ liệu, máy chủ ứng dụng, máy in , vv... .

Smoke testing

- Smoke Testing xảy ra khi thành phần mới được thêm vào và được tích hợp vào phần code đã có của phần mềm. Nó đảm bảo việc làm việc ăn khớp của khối code mới
- Là bước đầu kiểm tra sau đó cần kiểm thử thêm các bằng các kĩ thuật khác .

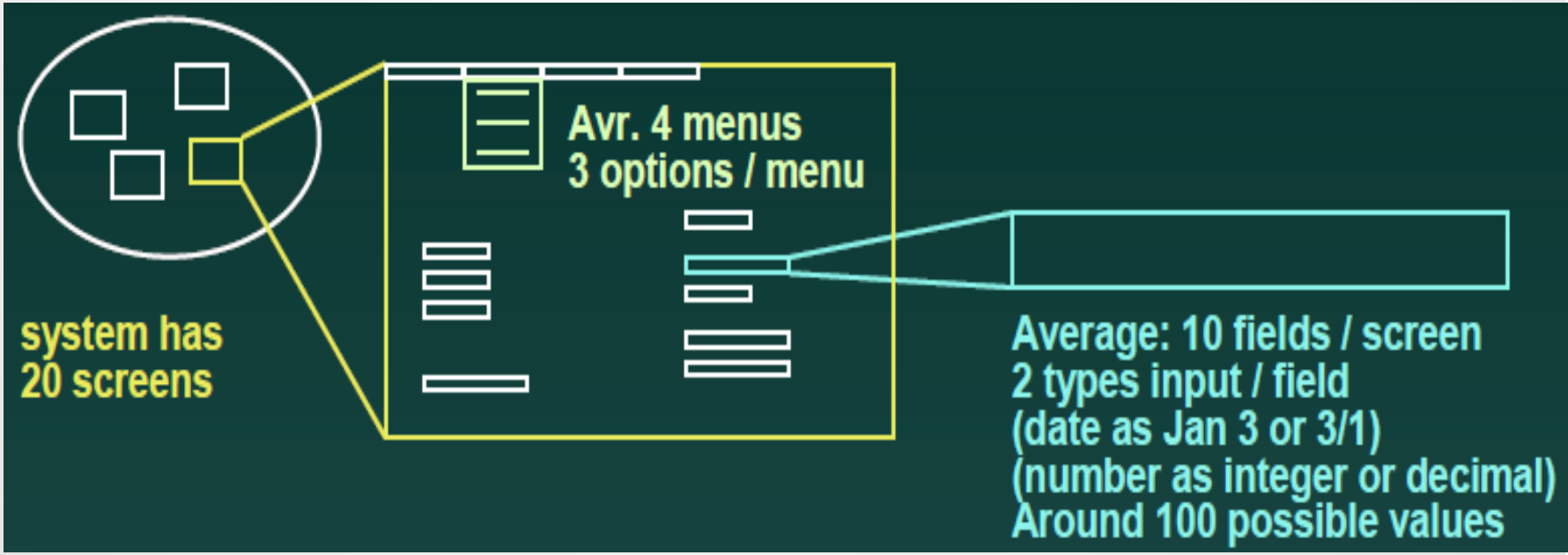
WBT – Cac ky thuat

- Kiểm thử luồng, lộ trình (Deriving Test Cases)
  - Lộ trình cơ sở (Basis pmath Testing)
- Luồng điều khiển / Phạm vi •(Control-flow / Coverag e Testing))
  - Phương thức -Method Coverage
  - Câu lệnh –StatementCoverge
  - Nhánh -Branch Coverge
  - Điều kiện –Condition Converage
- Kiểm thử luồng dữ liệu (Data Flow Test)
- Trường hợp hỏng ‘rác'– Failure ‘Dirty' Case Test
- Flow Groaps Revisited

Levels or Phases of Testing

- Unit: testing các mẫu công việc nhỏ nhất của lập trình viên để có thể lập kế hoạch và theo dõi hợp lý (vd : function, procedure, module, object class, ...)
- Component: testing 1 tập hợp các units tạo thành 1 thành phần (vd : program, package, task, interacting object classes, ...)
- Product: testing các thành phần tạo thành 1 sản phẩm (subsystem, application, ...)
- System: testing toàn bộ hệ thống
- Testing thường:
  - Bắt đầu = functional (black-box) tests,
  - Rồi thêm = structural (white-box) tests, và
  - Tiến hành từ unit level đến system level với 1 hoặc một vài bước tích hợp

Tại sao không "test everything"?



- **Chi phí cho 'exhaustive' testing:**

  20 x 4 x 3 x 10 x 2 x 100 = 480,000 tests

- **Nếu 1 giây cho 1 test, 8000 phút, 133 giờ, 17.7 ngày**
  (chưa kể nhầm lẫn hoặc test đi test lại)

  nếu 10 secs = 34 wks, 1 min = 4 yrs, 10 min = 40 yrs

Bao nhiêu testing là đủ?

- Không bao giờ đủ !
- Khi bạn thực hiện những test mà bạn đã lên kế hoạch
- Khi khách hàng / người sử dụng thấy thỏa mãn
- Khi bạn đã chứng minh được / tin tưởng rằng hệ thống hoạt động đúng, chính xác
- Phụ thuộc vào risks for your system
- Càng ít thời gian, càng nhiều để thời gian để test luôn có giới hạn
- Dùng RISK để xác định:
  - Cái gì phải test trước
  - Cái gì phải test nhiều
  - Mỗi phần tử cần test kỹ như thế nào ? Tức là đâu là trọng tâm
  - Cái gì không cần test (tại thời điểm này…)

The testing paradox

- Mục đích của testing: để tìm ra lỗi
- Tìm thấy lỗi làm mất (hủy hoại) sự tự tin - confidence
- => Mục đích của testing: hủy hoại sự tự tin
- Nhưng mục đích của testing: Xây dựng niềm tin, tự tin
- => Cách tốt nhất để xây dựng niềm tin là: Cố gắng hủy hoại nó

Summary

- External testing taxonomy
  - Boundary testing
  - Statement testing
  - Path testing
  - Stress testing
- Internal testing techniques
  - Checking invariants
  - Verifying conservation properties
  - Checking function return values
  - Changing code temporarily
  - Leaving testing code intact

Summary (cont.)

- General testing strategies
  - Testing incrementally
    - Regression testing
    - Scaffolds and stubs
  - Automation
  - Comparing independent implementations
  - Bug-driven testing
  - Fault injection

- Test the code, the tests, and the specification!
- Kiểm chứng code, kiểm chứng việc kiểm chứng – và kiểm chứng cả đặc tả !