Mục lục

I Tổng hợp kỹ thuật viết mã nguồn với các cấu trúc lập trình

I.1 Sử dụng chương 14 – Tổ chức các câu lệnh tuần tự.

Các kỹ thuật tiêu biểu gồm:

KT 14.1. The strongest principle for organizing straight-line code is ordering dependencies.

KT 14.2. Dependencies should be made obvious through the use of good routine names, parameter lists, comments, and—if the code is critical enough—housekeeping variables.

KT 14.3. If code doesn't have order dependencies, keep related statements as close together as possible.

I.2 Sử dụng chương 15 – Sử dụng câu lệnh If-Then

Các kỹ thuật tiêu biểu gồm:

KT 15.1. For simple if-else statements, pay attention to the order of the if and else clauses, especially if they process a lot of errors. Make sure the nominal case is clear.

KT 15.2. For if-then-else chains and case statements, choose an order that maximizes read-ability.

KT 15.3. To trap errors, use the default clause in a case statement or the last else in a chain of if-then-else statements.

KT 15.4. All control constructs are not created equal. Choose the control construct that's most appropriate for each section of code.

I.3 Sử dụng chương 16 – Vòng lặp

Các kỹ thuật tiêu biểu gồm:

KT 16.1. Loops are complicated. Keeping them simple helps readers of your code.

Vòng lặp thì phức tạp. Giữ cho vòng lặp trở nên đơn giản hơn cho người đọc dễ đọc code của bạn hơn.

Lựa chọn loại vòng lặp:

- Sử dụng biến đếm giúp xác định số lần lặp cụ thể.

- Đánh giá một vòng lặp ở thời gian mà vòng lặp sẽ thực hiện, nó sẽ kết thúc sau bao nhiêu vòng lặp hay lặp vô hạn.
- Các loại vòng lặp khác nhau thì khác biệt ở vị trí câu lệnh kiểm tra điều kiện kết thúc (vị trí này có thể ở đầu, giữa hoặc ở cuối vòng lặp).
- Các loại vòng lặp cũng được phân biệt bằng độ linh hoạt của chúng -> là việc các vòng lặp sẽ chạy trong một số lần nhất định hay phải kiểm tra sự kết thúc sau mỗi chu kì.
- Sử dụng linh hoạt vòng lặp while và for.

Vòng lặp while:

```
count = 0
while (count < 9):
    print('So thu tu cua ban la: ', count)
    count += 1
```

Vòng lặp for:

```
for letter in 'Python':
    print 'Chu cai hien tai: ', letter
qua = ['chuoi', 'tao', 'xoai']
for q in qua:
    print "Ban co thich an: ", q
```

KT16.2. Techniques for keeping loops simple include avoiding exotic kinds of loops, minimizing nesting, making entries and exits clear, and keeping housekeeping code in one place.

Kĩ thuật để làm cho vòng lặp trở nên đơn giản bao gồm tránh các vòng lặp lạ, tối thiểu các vòng lặp lồng nhau, tạo và thoát vòng lặp một cách rõ ràng, giữ các housekeeping code ở một vị trí.

Minimizing nesting:

- Độ dài vòng lặp có thể được tính bởi số dòng của vòng lặp hay số các vòng lặp lồng nhau.
- Nên tạo các vòng lặp ngắn sao cho có thể quan sát tất cả vòng lặp trong một lần.
- Hạn chế số lượng các vòng lặp lồng nhau ở con số 3.
- Nếu vòng lặp dài thì phải thật rõ ràng.

Lồng vòng lặp while để in các số nguyên tố nhỏ hơn 100:

```
i = 2
while (i < 100):
    j = 2
    while (j <= i / j):
        if i % j == 0:
            break
        j += 1
    if (j > i / j):
        print i, " la so nguyen to"
    i += 1
```

Making entries clearly:

- Đặt các giá trị khởi tạo cho các biến trong vòng lặp trước khi vào vòng lặp đó.
- Truy cập vòng lặp chỉ tại một vị trí. Việc này giúp tránh xảy ra lỗi khi ta copy hoặc di chuyển vòng lặp tới một vị trí khác trong mã nguồn. Hơn nữa, khi kích thước chương trình lớn, việc đặt các lệnh khởi tạo trước vòng lặp giúp cho việc thay đổi nó dễ dàng, nhanh chóng hơn.
- Sử dụng vòng lặp while, for cho vòng lặp vô hạn.

Making exits clearly:

- Đảm bảo rằng vòng lặp có thể kết thúc.
- Tạo điều kiện kết thúc rõ ràng cho vòng lặp.

Lặp vô hạn:

```
var = 1
while var == 1:      # lenh nay tao mot vong lap vo han
    num = raw_input("Hay nhap mot so: ")
    print "So da nhap: ", num
```

Sử dụng *break* và *continue* để điều khiển vòng lặp:

```
for letter in 'Python':     # vi du thu nhat
    if letter == 'h':
        break
    print 'Chu cai hien tai: ', letter

var = 10                    # vi du thu hai
while var > 0:
    print 'Gia tri bien hien tai la: ', var
    var = var - 1
    if var == 5:
        break
```

```
for letter in 'Python':     # Vi du thu nhat
   if letter == 'h':
      continue
   print 'Chu cai hien tai :', letter

var = 10                    # Vi du thu hai
while var > 0:
   var = var -1
   if var == 5:
      continue
   print 'Gia tri bien hien tai la :', var
```

Keep housekeeping code in one place:

- Housekeeping code là những lệnh như i++, j = j+1,... để điều khiển vòng lặp. Nên đặt các lệnh này ở đầu hoặc cuối vòng lặp.

KT 16.3. Loop indexes are subjected to a great deal of abuse. Name them clearly, and use them for only one purpose.

Các chỉ số vòng lặp bị lạm dụng rất nhiều. Đặt tên cho chúng thật rõ ràng và chỉ sử dụng cho một mục đích.

Cách dùng các chỉ số vòng lặp:

- Dùng biến đếm hoặc biến thứ tự.
- Biến đếm nên là số nguyên và không nên là số thực dấu phẩy động.
- Sử dụng các tên biến có ý nghĩa khi làm việc với các vòng lặp lồng nhau (thay vì dùng i, j, k,... thì đặt tên có sự gợi nhớ) -> không mắc phải sử dụng nhầm tên biến hay sử dụng một biến nhiều lần.
- Hạn chế phạm vi của các biến vòng lặp chỉ trong vòng lặp đó.

KT 16.4.Think through the loop carefully to verify that it operates normally under each case and terminates under all possible conditions.

Xem xét toàn bộ vòng lặp một cách cẩn thận để kiểm chứng rằng nó đã hoạt động bình thường trong mọi trường hợp và có thể kết thúc dưới mọi điều kiện.

- Đảm bảo rằng vòng lặp kết thúc được (bằng cách cho chạy thử một số lần lặp với các giá trị cụ thể, xem xét tất cả các trường hợp có thể xảy ra, các điểm kết thúc, các ngoại lệ).
- Viết các điều kiện kết thúc vòng lặp rõ ràng.
- Kiểm tra vòng lặp có kết thúc với kết quả như mong muốn hay không.

I.4 Sử dụng chương 17 – Các cấu trúc điều khiển khác

Các kỹ thuật tiêu biểu gồm:

KT 17.1. Multiple returns can enhance a routine's readability and maintainability, and they help prevent deeply nested logic. They should, nevertheless, be used carefully.

Chương trình con có nhiều lần trả về có thể nâng cao khả năng đọc và bảo trì của chương trình và giúp ngăn ngừa sai sót trong những đoạn lồng nhau. Dù vậy chúng nên được sử dụng cẩn thận.

- Sử dụng *return* làm chương trình con dễ đọc hơn.
- Giúp ngăn chặn việc xảy ra lỗi khi chạy chương trình.
- Khi dùng *return* phải cẩn trọng xem xét hết các trường hợp.

```
def min(x, y):
    if x < y:
        return x
    elif y > x:
        return y
    else
        return x
```

KT17.2. Recursion provides elegant solutions to a small set of problems. Use it carefully, too.

Một tập các vấn đề nhỏ thì được giải quyết bằng phương pháp đệ quy. Phải sử dụng đệ quy thật cẩn thận.

- Thuật toán đệ quy là thuật toán dựa trên phương pháp chia để trị, chia bài toán thành các bài toán con giải quyết các vấn đề nhỏ hơn bằng cách tự gọi đến chính nó với đầu vào kích thước nhỏ hơn.
- Sử dụng thuật toán đệ quy là thuận tiện khi các bài toán con dễ xử lý trong khi với các bài toán lớn thì lại phức tạp.
- Đệ quy là một trong những cách tiếp cận hiệu quả và đơn giản đối với một bài toán khó, giúp cho người đọc code có thể dễ dàng nắm bắt được ý tưởng của lập trình viên.

Chú ý khi sử dụng đệ quy:

- Đảm bảo rằng đệ quy phải có điểm dừng.

- Tránh sử dụng đệ quy xoay vòng (ví dụ A gọi B, B gọi C, C gọi A), việc này là nguy hiểm vì dễ dẫn đến chạy vô hạn.
- Kiểm soát xem stack đang dùng còn đủ dung lượng không.
- Không nên sử dụng đệ quy cho các bài toán có thể giải quyết dễ dàng bằng vòng lặp nói chung (ví dụ như bài toán tính giai thừa, tìm số Fibonaci,...).

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else
        return n * factorial(n - 1)
```

KT17.3. In a few cases, gotos are the best way to write code that's readable and maintainable. Such cases are rare. Use gotos only as a last resort.

Trong một số ít trường hợp, lệnh goto là cách tốt nhất để viết code trở nên dễ đọc và dễ bảo trì. Tuy nhiên các trường hợp này rất hiếm khi xảy ra. Việc sử dụng gôt chỉ là phương án cuối cùng.

- Goto thường được dùng để xử lý các cấu trúc rẽ nhánh lồng nhau.
- Lý do goto không được khuyến khích sử dụng là nó có thể được thay thế bởi các lệnh khác (ví dụ break/continue) hoặc gom phần code mà goto chỉ đến vào trong hàm.
- Trong python không hỗ trợ lệnh goto.

II Tổng hợp kỹ thuật làm việc với các biến

II.1 Sử dụng chương 10 – Các kỹ thuật chung làm việc với biến

Các kĩ thuật tiêu biểu gồm:

KT10.1. Data initialization is prone to errors, so use the initialization techniques described in this chapter to avoid the problems caused by unexpected initial values.

KT10.2. Minimize the scope of each variable. Keep references to a variable close together. Keep it local to a routine or class. Avoid global data.

KT10.3. Keep statements that work with the same variables as close together as possible.

KT10.4. Early binding tends to limit flexibility but minimize complexity. Late binding tends to increase flexibility but at the price of increased complexity.

KT10.5. Use each variable for one and only one purpose.

II.2 Sử dụng chương 11 – Kỹ thuật đặt tên biến

Các kỹ thuật tiêu biểu gồm:

KT11.1. Good variable names are a key element of program readability. Specific kinds of variables such as loop indexes and status variables require specific considerations.

KT11.2. Names should be as specific as possible. Names that are vague enough or general enough to be used for more than one purpose are usually bad names.

KT11.3. Naming conventions distinguish among local, class, and global data. They distinguish among type names, named constants, enumerated types, and variables.

KT11.4. Regardless of the kind of project you're working on, you should adopt a variable naming convention. The kind of convention you adopt depends on the size of your program and the number of people working on it.

KT11.5. Abbreviations are rarely needed with modern programming languages. If you do use abbreviations, keep track of abbreviations in a project dictionary or use the standardized prefixes approach.

KT11.6. Code is read far more times than it is written. Be sure that the names you choose favor read-time convenience over write-time convenience.

II.3 Sử dụng chương 12 – Các kiểu dữ liệu cơ bản

Các kĩ thuật tiêu biểu gồm:

KT12.1. Working with specific data types means remembering many individual rules for each type. Use this chapter's checklist to make sure that you've considered the common problems.

Làm việc với những kiểu dữ liệu cụ thể nghĩa là phải nhớ nhiều những quy tắc riêng biệt cho từng kiểu. Như những quy tắc dưới đây:

- Kiểu dữ liệu số tổng quát:
  - Tránh những "magic number", những hằng số xuất hiện ở giữa code mà không có giải thích
  - Có thể code trực tiếp 0, 1 nếu cần thiết
  - Dự tính trường hợp chia cho 0
  - Tránh so sánh khác kiểu giữ liệu
  - Chú ý những cảnh báo của trình dịch
- Kiểu số nguyên:
  - Kiểm tra chia nguyên với chia thực
  - Kiểm tra tràn số
  - Kiểm tra tràn số trong các giá trị trung gian
- Kiểu số thực chấm phẩy động:
  - Tránh cộng và trừ những số có độ lớn khác nhau nhiều
  - Tránh so sánh bằng
  - Dự kiến lỗi làm tròn
- Ký tự và xâu:
  - Tránh những ký tự và xâu "magic"
  - Hiểu rõ ngôn ngữ bạn đang dùng hỗ trợ Unicode ra sao
  - Định rõ khu vực, vùng mà chương trình sẽ được sử dụng
  - Nếu cần hỗ trợ nhiều ngôn ngữ, hãy dùng Unicode
- Biến boolean:
  - Sử dụng kiểu boolean để làm rõ những điều kiện trong chương trình
  - Sử dụng kiểu boolean để đơn giản hoá những test phức tạp
  - Tự tạo một kiểu dữ liệu boolean, nếu như cần thiết
- Kiểu enum:

  - Sử dụng kiểu enum cho tính dễ đọc
  - Sử dụng kiểu enum cho tính tin cậy
  - Sử dụng kiểu enum để cho việc sửa đổi dễ dàng
  - Sử dụng kiểu enum thay thế cho kiểu boolean
  - Sử dụng giá trị ban đầu của enum để cho những giá trị không hợp lệ
  - Xác định rõ phần tử đầu và cuối của enum được sử dụng như thế nào trong toàn bộ project
  - Nếu ngôn ngữ không có kiểu enum thì có thể sử dụng class hoặc từ điển

- Hằng số:
  - Sử dụng hằng số có đặt tên trong khai báo dữ liệu
  - Sử dụng hằng số một cách thống nhất

- Mảng:
  - Đảm bảo rằng chỉ số của mảng ở trong miền cho phép
  - Luôn nghĩ rằng mảng là một cấu trúc tuần tự
  - Nếu mảng là đa chiều, đảm bảo thứ tự của các chỉ số
  - Kiểm tra xem chỉ số khi lặp có đúng như mong muốn
  - Để thêm một phần tử ở cuối mảng

Ví dụ:

```
EPSILON = 0.0001
def are_same(a, b):
    return abs(a - b) < EPSILON
```

KT12.2. Creating your own types makes your programs easier to modify and more self-documenting, if your language supports that capability.

Tự tạo một kiểu dữ liệu mới sẽ làm cho chương trình dễ dàng sửa đổi và tăng tính tự mô tả, nếu như ngôn ngữ hỗ trợ khả năng này.
Một số những chỉ dẫn tự tạo một kiểu dữ liệu của mình:

- Tạo kiểu dữ liệu với tên hướng đến chức năng của nó
- Tránh những kiểu dữ liệu đã định nghĩa trước đó
- Đừng định nghĩa kiểu dữ liệu đã được định nghĩa
- Định nghĩa kiểu dữ liệu thay thế cho kiểu dữ liệu cũ để tăng tính khả chuyển

Trong Python không hỗ trợ tạo kiểu dữ liệu mới.

KT12.3. When you create a simple type using typedef or its equivalent, consider whether you should be creating a new class instead.

Nếu tự tạo một kiểu dữ liệu mới bằng typedef hoặc cách tương đương, xem xét việc sử dụng class để thay thế nó.

Ví dụ sử dụng class *StudentID* để thay thế cho *mssv* là một số:

```
class StudentID:
    # mssv
    def __init__(self, mssv):
        self.mssv = mssv
```

II.4 Sử dụng chương 13 – Các kiểu dữ liệu đặc biệt

Các kĩ thuật tiêu biểu gồm:

KT13.1. Structures can help make programs less complicated, easier to understand, and easier to maintain.

Kiểu dữ liệu có cấu trúc sẽ giúp chương trình giảm độ phức tạp, dễ dàng để hiểu, để duy trì nó hơn.
Một số những lưu ý khi sử dụng kiểu dữ liệu có cấu trúc:

- Sử dụng kiểu dữ liệu có cấu trúc để làm rõ mối quan hệ giữa các dữ liệu
- Sử dụng để đơn giản hoá những thủ tục lên khối dữ liệu
- Sử dụng để đơn giản hoá tham số đầu vào của hàm
- Sử dụng để giảm bớt việc bảo trì chương trình

Ví dụ:

```
class Person:
    def __init__(self, name, age, phone):
        self.name = name
        self.age = age
        self.phone = phone

def print_person(person):
    print("Ho ten: " + person.name)
    print("Tuoi: " + person.age)
    print("So dien thoai: " + person.phone)

person = Person("Ta Quang Tung", 19, 123)
print_person(person)
```

KT13.2. Whenever you consider using a structure, consider whether a class would work better.

Khi sử dụng một kiểu giữ liệu có cấu trúc, xem xét liệu rằng class sẽ tốt hơn.

Ví dụ:

```
http_request = {'method': 'get', 'url':'sis.hust.edu.vn', \
                        'id': 20154280}

def send_http_request(http_request):
    ...
```

Sẽ tốt hơn nếu được thay thế bằng:

```
class HttpRequest:
    def __init__(self, method, url, id):
        self.method = method
        self.url = url
        self.id = id
    def send(self):
        ...
```

KT13.3. Pointers are error-prone. Protect yourself by using access routines or classes and defensive-programming practices.

Con trỏ là kiểu dữ liệu dễ dàng gặp lỗi khi sử dụng. Bảo vệ bản thân bằng việc sử dụng hàm hoặc class khi muốn truy cập nó, hoặc sử dụng kỹ thuật lập trình "tự phòng thủ".

Những yêu cầu khi sử dụng biến con trỏ:

- Sử dụng một kiểu con trỏ rõ ràng thay vì kiểu con trỏ mặc định.
- Tránh cast kiểu dữ liệu con trỏ.
- Trong C, nếu bạn sử dụng dấu * với tham số ở hàm, luôn nhớ rằng giá trị có thể truyền ngược lại khi gọi hàm. Đồng thời khi bạn muốn truyền ngược lại giá trị, chắc chắn rằng phải có ít nhất một dấu sao trong một câu lệnh gán.
- Sử dụng sizeof() để xác định kích thước của biến khi cấp phát bộ nhớ cho nó.

Trong python không có kiểu con trỏ.

KT13.4. Avoid global variables, not just because they're dangerous, but because you can replace them with something better.

Tránh dử dụng biến toàn cục, không phải chỉ vì nó nguy hiểm mà còn bởi vì bạn có thể thay thế nó bằng những cách thức tốt hơn nhiều.

Những vấn đề chính khi sử dụng kiểu dữ liệu toàn cục:

- Vô tình thay đổi kiểu dữ liệu toàn cục.
- Có thể có những lỗi kì quặc khi sử dụng "tên thay thế" cho biến toàn cục. Chẳng hạn như việc truyền đồng thời tham số vào hàm là biến toàn cục, cũng như sử dụng biến toàn cục trong chính hàm.
- Vấn đề code được sử dụng nhiều lần trong khi nó truy cập đến biến toàn cục, chẳng hạn như lời gọi đệ quy, hoặc trong trường hợp đa luồng.
- Tính tái sử dụng của Code bị cản trở bởi kiểu toàn cục.
- Thứ tự khởi tạo không xác định khi sử dụng kiểu toàn cục.
- Tính module hoá và khả năng kiểm soát code bị huỷ hoại bởi sử dụng kiểu toàn cục.

Những lý do để sử dụng biến toàn cục:

- Bảo tồn những giá trị toàn cục, ví dụ bảng dữ liệu mà tất cả các hàm sử dụng.
- Liệt kê những giá trị hằng số, trong những ngôn ngữ không hỗ trợ hằng số như Python, Perl.
- Sử dụng như kiểu dữ liệu enum, như trong Python.
- Loại bỏ những dữ liệu "lang thang". Ví dụ như khi có 3 hàm, gọi lần lượt lẫn nhau theo đúng thứ tự 1, 2, 3. Hàm 1 và hàm 3 sử dụng cùng một object, nhưng hàm 2 thì không. Nếu truyền qua tham số thì ta vẫn phải truyền nó qua hàm 2, đó là dữ liệu "lang thang". Sử dụng biến toàn cục sẽ loại bỏ những dữ liệu này.

Sử dụng biến toàn cục chỉ khi nó là phương pháp cuối cùng.

- Bắt đầu bởi tạo một biến cục bộ và chỉ thay đổi nó thành toàn cục khi thực sự cần thiết.
- Phân biệt giữa biến toàn cục và biến trong class.
- Sử dụng những hàm, thủ tục để truy cập.

Làm thế nào để giảm bớt những rủi ro khi sử dụng biến toàn cục:

- Triển khai một quy ước đặt tên để làm cho biến toàn cục trở nên hiển nhiên.
- Tạo một danh sách các chú thích tốt cho tất cả các biến toàn cục.
- Đừng sử dụng biến toàn cục cho các kết quả trung gian.
- Đừng giả vờ rằng bạn không sử dụng biến toàn cục bằng cách đặt tất cả dữ liệu vào một object rất lớn và truyền nó khắp mọi nơi.

KT13.5. If you can't avoid global variables, work with them through access routines. Access routines give you everything that global variables give you, and more.

Nếu không thể tránh khỏi việc sử dụng biến toàn cục thì nên sử dụng những hàm, thủ tục để truy cập nó thay vì truy cập trực tiếp. Truy cập nhờ hàm, thủ tục sẽ cho bạn mọi thứ như khi sử dụng biến đó trực tiếp, và thêm nhiều hơn thế nữa.

Lợi ích của việc sử dụng hàm truy cập biến toàn cục:

- Bạn có thể tập trung hoá việc truy cập biến toàn cục. Nếu bạn nghĩ ra một cách thức truy cập mới cho dữ liệu đang sử dụng, khi đó bạn không phải thay đổi code nơi mà dữ liệu được sử dụng, chỉ cần thay đổi code trong hàm truy cập.
- Có thể đảm bảo rằng tất cả các truy cập đến biến toàn cục được "rào chắn".
- Bạn sẽ được lợi nhờ việc che giấu thông tin tự động.
- Hàm truy cập sẽ giúp dễ dàng cho việc chuyển sang kiểu dữ liệu trừu tượng.

Làm thế nào để sử dụng hàm truy cập:

- Yêu cầu tất cả các truy cập biến phải thông qua hàm truy cập.
- Đừng chỉ vứt toàn bộ những biến toàn cục vào một cái thùng.
- Sử dụng cơ chế khoá để bảo vệ biến toàn cục khi lập trình đa luồng.
- Xây dựng thêm một mức trừu tượng hoá vào trong hàm truy cập.
- Giữ tất cả các truy cập của dữ liệu ở cùng một mức trừu tượng, đừng cho phép sử dụng nhiều mức trừu tượng để truy cập giữ liệu.

Ví dụ về hàm truy cập kết hợp cơ chế đồng bộ:

```
__counter = 0
__mt = mutex()

def get_counter():
    __mt.lock()
    tmp = __counter
```

```
    __mt.unlock()
    return tmp

def set_counter(value):
    __mt.lock()
    __counter = value
    __mt.unlock()

def inc_counter():
    __mt.lock()
    __counter += 1
    __mt.unlock()
```

III Tổng hợp kỹ thuật xây dựng chương trình, hàm, thủ tục

III.1 Sử dụng chương 5 – Các kỹ thuật thiết kế chương trình phần mềm – Design in Construction

Các kĩ thuật tiêu biểu gồm:

KT5.1. Software's Primary Technical Imperative is managing complexity. This is greatly aided by a design focus on simplicity.

Yêu cầu kỹ thuật chính đối với phần mềm là kiểm soát sự phức tạp. Nó được hỗ trợ rất lớn bởi thiết kế tập trung vào sự đơn giản.

Tầm quan trọng của việc quản lý sự phức tạp trong phát triển phần mềm:

Khi một cuộc khảo sát về những dự án phần mềm báo cáo nguyên nhân dẫn đến thất bại, họ thường không xác định rõ là những vấn đề về mặt kĩ thuật là nguyên nhân chính gây nên thất bại của dự án. Một dự án thất bại phần lớn bởi vì yêu cầu không rõ ràng, lập kế hoạch không đầy đủ, và kiểm soát kém chặt chẽ. Nhưng khi một dự án thất bại vì lý do chính là kỹ thuật, lý do thường sẽ là không thể kiểm soát được sự phức tạp bên trong phần mềm đó. Phần mềm có thể được phép phát triển đến mức cực kì phức tạp mà không ai có thể hiểu thực sự nó đang làm gì. Khi dự án chạm mức không một ai có thể thực sự hiểu ảnh hưởng của những đoạn code này trong vùng này có tác độngđộng đến đoạn code trong vùng khác như thế nào, sự tiến triển của phần mềm sẽ bị kìm hãm xuống một nửa.

Kiểm soát sự phức tạp của phần mềm là vấn đề kỹ thuật quan trọng nhất trong phát triển phần mềm.

Sự phức tạp không phải là mới trong công việc phát triển phần mềm. Nhà tiên phong trong khoa học tính toán Edsger Dijkstra đưa ra một luận điểm rằng: tính toán

chỉ thực sự chuyên nghiệp khi một bộ óc có trách nhiệm phải mở rộng một dữ liệu từ một bit đến một vài megabyte, một tỉ lệ tầm 1 / 10^9, hay 9 bậc về độ lớn (Dijkstra 1989). Tỉ lệ đó thật đáng kinh ngạc, Dijkstra đặt nó trong câu nói: "Compared to that number of semantic levels, the average mathematical theory is almost flat. By evoking the need for deep conceptual hierarchies, the automatic computer confronts us with a radically new intellectual challenge that has no precedent in our history." – Nếu so sánh về số lượng mức ngữ nghĩa, trung bình một lý thuyết toán học gần như bằng phẳng. Bởi việc gợi nên sự cần thiết cho "hệ thống phân cấp khái niệm rất nhiều lớp", máy tính tự động thách thức chúng ta với một thử thách trí tuệ hoàn toàn mới, cái mà không hề có tiền lệ trong lịch sử. Tất nhiên phần mềm đã ngày càng trở nên phức tạp, và tỉ lệ của Dijkstra có thể là 1 / 10^{15} ngày hôm nay.

Dijkstra cho rằng không một ai có một bộ óc đủ lớn để chứa đựng một phần mềm máy tính hiện đại, nghĩa là một developer không nên cố gắng nhồi nhét toàn bộ phần mềm vào óc mình tại một thời điểm. Chúng ta phải cố gắng tổ chức chương trình bằng một cách nào đó để tại một thời điểm, ta có thể chỉ cần tập trung vào một phần của chương trình.

KT5.2. Simplicity is achieved in two general ways: minimizing the amount of essential complexity that anyone's brain has to deal with at any one time, and keeping accidental complexity from proliferating needlessly.

Làm thế nào chống lại sự phức tạp

Có 3 nguồn của sự tốn kém quá mức, sự thiết kế không hiệu quả:

- Một lời giải phức tạp cho một vấn đề đơn giản.
- Một lời giải đơn giản, không chính xác cho một vấn đề phức tạp.
- Một lời giản phức tạp, không hợp lý cho một vấn đề phức tạp.

Như là Dijkstra đã chỉ ra, phần mềm hiện đại bản thân nó là phức tạp, không kể bạn có cố gắng như thế nào, bạn chắc chắn sẽ phải động chạm vào một vài mức của sự phức tạp mà nó là cái vốn có của những vấn đề thực tế. Hai cách tiếp cận để kiểm soát sự phức tạp:

- Tối thiểu hoá sự phức tạp "thiết yếu" cho bất kì ai phải động đến, tại bất kì thời điểm nào.
- Giữ cho sự phức tạp "tình cờ" tránh khỏi việc trở nên không cần thiết.

KT5.3. Design is heuristic. Dogmatic adherence to any single methodology hurts creativity and hurts your programs.

Thiết kế là một công việc heuristic. Bất kì một phương pháp nào tuân thủ theo cách độc đoán, giáo điều sẽ phá hoại sự sáng tạo và phá hoại chương trình của bạn.

Những thách thức trong thiết kế phần mềm:

- Thiết kế là một "vấn đề mờ" (wicked problem) – "vấn đề mờ" là những vấn đề mà có thể định nghĩa rõ ràng chỉ khi bởi giải quyết nó, hoặc một phần của nó. Một trong những sự khác biệt giữa chương trình bạn phát triển ở trường và cái mà bạn phát triển khi chuyên nghiệp đó là vấn đề thiết kế được giao ở trường hiếm khi mơ hồ.
- Thiết kế là một quá trình luộm thuộm. Kết quả sau khi hoàn tất thiết kế phần mềm là tổ chức rõ ràng và sạch sẽ. Nhưng quá trình được sử dụng khi phát triển bản thiết kế gần như không "tươm tất" như kết quả cuối cùng. Mắc lỗi là điểm cần thiết của công việc thiết kế. Sự luộm thuộm đến từ việc khó để biết khi nào bản thiết kế của chúng ta đã đủ tốt. Việc thiết kết là một công việc "kết thúc mở", thông thường chỉ kết thúc khi bạn hết thời gian.
- Thiết kế là về việc đánh đổi và sự ưu tiên: Phần chính của một nhà thiết kế là phải đánh giá những đặc tính của phần mềm và cân bằng giữa những đặc tính đó.
- Thiết kế liên quan đến sự hạn chế: Sự ràng buộc trong giới hạn tài nguyên khi xây dựng phần mềm thúc đẩy sự đơn giản hoá trong lời giải, cái mà cuối cùng sẽ cải thiện lời giải của vấn đề.
- Thiết kế là một công việc không xác định: Nếu bạn cho ba người riêng biệt cùng thiết kế một phần mềm, họ chắc chắn sẽ trả về ba bản thiết kế gần như hoàn toàn khác nhau, và có thể tất cả số chúng đều hoàn toàn được chấp nhận.
- Thiết kế là một quá trình heuristic: Bởi vì thiết kế là công việc không xác định, các kỹ thuật thiết kế có xu hướng "heuristic" – Không bao giờ một phương thức có thể được áp dụng chính xác trong tất cả các trường hợp. Không có công cụ nào là đúng cho mọi thứ.
- Thiết kế là "sự xuất hiện": Thiết kế không phải được hình thành đầy đủ từ bộ óc của một người, nó liên quan và được cải thiện dần dần bởi sự xem xét, sử thảo luận, kinh nghiệm khi viết code hoặc sửa lại code.

KT5.4. Good design is iterative; the more design possibilities you try, the better your final design will be.

Một bản thiết kế tốt là quá trình lặp đi lặp lại, càng nhiều những khả năng thiết kế mà bạn thử, kết quả thiết kế cuối cùng của bạn sẽ càng tốt.

Những đặc trưng mong muốn của một bản thiết kế:

- Sự phức tạp tối thiểu.
- Dễ dàng bảo trì.
- Tối thiểu hoá sự kết nối.

- Tính mở rộng.
- Tính tái sử dụng.
- "Lượng đầu vào" cao – "Lượng đầu vào" ở đây ám chỉ việc một class được các class khác sử dụng như thế nào. "Lượng đầu vào" cao nghĩa là hệ thống phải có các class hữu ích được sử dụng rộng khắp chương trình.
- "Lượng đầu ra" thấp hoặc trung bình – "Lượng đầu ra" ở đây ám chỉ việc một class sử dụng số class khác bao nhiêu. "Lượng đầu ra" cao (nhiều hơn khoảng 7) chỉ định rằng class đó sử dụng một lượng lớn các class khác, và có thể gây nên sự phức tạp quá mức.
- Tính khả chuyển.
- Sự "nghèo nàn" – "nghèo nàn" ở đây nghĩa là hệ thống không có những phần dư thừa.
- Sự phân tầng – Thiết kế hệ thống sao cho bạn có thể nhìn nó ở một mức phân cấp mà không phải đào sâu vào các mức khác.
- Sử dụng những kĩ thuật đã chuẩn hoá.

Sự lặp lại trong thiết kế:

Thiết kế là một quá trình lặp đi lặp lại: Bạn sẽ không chỉ đi từ điểm A đến điểm B, bạn sẽ đi từ điểm A tới B và quay ngược trở lại A.

Với mỗi lần lặp lại và thử với cách tiếp cận khác nhau, bạn sẽ nhìn vấn đề dưới góc nhìn ở cả mức thấp và mức cao. Bức tranh lớn mà bạn nhận được với những vấn đề ở mức cao sẽ giúp bạn đặt những vấn đề mức thấp ở đúng vị trí. Những chi tiết của vấn đề mức thấp cung cấp cho bạn nền tảng vững chắc cho những quyết định ở mức cao. Hoán đổi việc xem xét giữa mức thấp và mức cao là khá khó khăn, nhưng nó là thiết yếu đối với việc thiết kế hiệu quả. Khi tạo ra bản thiết kế đầu tiên mà có vẻ đủ tốt, đừng nên dừng lại. Nỗ lực lần thứ hai sẽ gần như luôn luôn tốt hơn lần đầu.

KT5.5. Information hiding is a particularly valuable concept. Asking "What should I hide?" settles many difficult design issues.

Ẩn giấu thông tin là một khái niệm có giá trị đặc biệt. Luôn luôn đặt câu hỏi "Cái gì tôi nên che giấu?" sẽ giải quyết nhiều những khó khăn trong vấn đề thiết kế.

Che giấu thông tin:

Che giấu thông tin là nền tảng của cả thiết kế hướng cấu trúc và hướng đối tượng. Trong thiết kế hướng cấu trúc, khái niệm "hộp đen" đến từ việc che giấu thông tin. Trong thiết kế hướng đối tượng, nó tạo ra những khái niệm đóng gói và module hoá, và nó có liên quan đến khái niệm về sự trừu tượng.

Bí mật và quyền riêng tư:
Trong che giấu thông tin, mỗi class (hoặc một hàm, một package) được đặc trưng bởi những quyết định về thiết kế hoặc xây dựng nó, cái mà nó che giấu từ tất cả những class khác. Vùng bí mật có thể là vùng có khả năng cao sẽ thay đổi, định dạng một file, hay là cách mà dữ liệu được thực hiện, hay những giữ liệu được che chắn cho phần còn lại của chương trình để mà lỗi trong phần này gây nên những hư hại nhỏ cho các phần khác của chương trình. Vai trò của một class là giữ cho những thông tin này ẩn và để bảo vệ nó khỏi những truy cập không mong muốn từ bên ngoài. Một thay đổi nhỏ có thể ảnh hưởng một vài hàm trong class, nhưng nó không nên vượt quá interface của class.

Một trong những nhiệm vụ chính khi thiết kế một class là quyết định xem những đặc điểm nào sẽ được biết đến bên ngoài class, và những cái nào sẽ được giữ bí mật. Một class có thể có 25 hàm, chỉ có 5 hàm là public, và 20 hàm còn lại chỉ sử dụng với mục đích trong của class. Một class có thể sử dụng một vài kiểu dữ liệu mà không phơi bày bất kì thông tin nào về nó.

Interface của một class nên tiết lộ ít thông tin nhất có thể về hoạt động bên trong của nó. Thiết kế một interface là một quá trình lặp đi lặp lại như những khía cạnh khác của thiết kế. Nếu bạn không đạt được một interface đúng từ đầu, thử một vài lần cho đến khi nó ổn định. Nếu nó không thể ổn định, bạn cần phải thử một cách tiếp cận khác.
Hai dạng của bí mật:

- Ẩn giấu sự phức tạp để mà não của bạn không phải động đến nó trừ khi bạn có quan tâm đặc biệt đến nó.
- Ẩn giấu nguồn gốc của sự thay đổi để mà khi thay đổi xảy ra, nó chỉ ảnh hưởng trong một vùng cục bộ.

Nguồn gốc của sự phức tạp bao gồm kiểu dữ liệu phức tạp, cấu trúc file, những test đúng sai, liên quan đến thuật toán,...

Những cản trở cho việc che giấu thông tin:

- Sự phân tán quá mức của thông tin.
- Phụ thuộc vòng tròn.
- Không sử dụng class thay cho những biến toàn cục.
- Việc tránh làm giảm hiệu năng của hệ thống.

Giá trị của việc che giấu thông tin: Suy nghĩ về việc che giấu thông tin sẽ truyền cảm hứng và nâng cao những lựa chọn khi thiết kế hơn là tư duy hướng đối tượng.

Việc che giấu thông tin còn hữu dụng trong việc thiết kế public interface của class. Khoảng cách giữa lý thuyết và thực tế trong thiết kế class là lớn, và giữa các người

thiết kế class, quyết định thông thường sẽ là interface của class phải thuận tiện để sử dụng nhất có thể. Điều đó thường dẫn đến những thông tin của class bị phơi bày nhiều nhất có thể.

Luôn đặt câu hỏi "Cái gì của class mà cần che giấu?", đều đó sẽ giúp giảm bớt rất nhiều những vấn đề khi thiết kế interface. Nếu bạn có thể đặt một hàm vào public interface của class mà không gây hại đến tính bí mật của nó thì hãy làm, còn không thì đừng nên.

Đặt câu hỏi về điều gì bạn cần để ẩn thông tin sẽ hỗ trợ cho việc lựa chọn phương pháp thiết kế ở bất kì một mức nào của phần mềm. Nó đẩy mạnh việc sử dụng những hằng số được đặt tên thay vì những hằng số trực tiếp, nó giúp cho việc tạo ra những tên hàm, tên biến tốt đồng thời hướng dẫn cho việc lựa chọn thiết kế class, các hệ thống con và cách thức liên kết giữa chúng.

KT5.6. Lots of useful, interesting information on design is available outside this book. The perspectives presented here are just the tip of the iceberg.

III.2 Sử dụng chương 7 – Kỹ thuật xây dựng hàm/thủ tục High-Quality Routines

Các kĩ thuật tiêu biểu gồm:

KT7.1. The most important reason for creating a routine is to improve the intellectual manageability of a program, and you can create a routine for many other good reasons. Saving space is a minor reason; improved readability, reliability, and modifiability are better reasons.

Lý do quan trọng nhất để tạo một hàm, thủ tục là để cải thiện khả năng kiểm soát những vấn đề liên quan đến "trí tuệ" của chương trình, và bạn có thể tạo một hàm cho rất nhiều những lý do khác nữa. Tiết kiệm không gian là lý do nhỏ, nâng cao khả năng đọc hiểu, tính tin cậy và khả năng sửa đổi là lý do quan trọng hơn cả.

Những lý do để tạo một hàm, thủ tục:

- Giảm bớt sự phức tạp: Tạo một hàm sẽ giúp che giấu thông tin để mà bạn sẽ không cần phải nghĩ về nó. Bạn sẽ phải nghĩ về nó khi viết hàm. Nhưng sau khi nó được tạo ra, bạn có thể quên đi chi tiết về hoạt động bên trong của nó và sử dụng mà không phải lo nghĩ về điều đấy. Lý do khác là để tối thiểu kích thước code, nâng cao khả năng bảo trì, cải thiệt tính đúng đắn của chương trình. Đó cũng là những lý do tốt, nhưng nếu không sức mạnh của tính trừu tượng nhờ việc tạo hàm, một chương trình phức tạp sẽ không thể quản lý được.

- Tạo một vùng code dễ đọc hiểu hơn: Đặt một vùng code vào một hàm đặt tên tốt là cách hợp lý nhất để giải thích mục đích của nó.

- Tránh lặp lại code.

- Ẩn giấu thứ tự thực hiện.
- Ẩn giấu các phép toán trên con trỏ.
- Cải thiện tính khả chuyển.
- Đơn giản hoá những test đúng sai phức tạp.
- Cải thiện hiệu năng.

KT7.2. Sometimes the operation that most benefits from being put into a routine of its own is a simple one.

Đôi khi, một phép toán được lợi nhiều nhất từ việc đặt nó vào trong hàm, thủ tục lại là một phép toán đơn giản.

Xây dựng cả một hàm chỉ chứa một, hai dòng code có thể là quá mức cần thiết. Nhưng kinh nghiệm đã chỉ ra rằng những hàm nhỏ như vậy lại có ích rất nhiều.

Tạo ra các hàm nhỏ có một vài lợi thế, một trong số đó là cải thiện tính dễ đọc hiểu của code.
Ví dụ dòng lệnh:

```
points = device_units * (POINT_PER_INCH / device_units_per_inch())
```

Có thể khá khó để nhận ra ngày dòng code đó có nghĩa là gì, nhưng nếu viết một hàm như thế này:

```
def device_units_to_point(device_units):
    return device_units * (POINT_PER_INCH /      \
                device_units_per_inch())
```

Khi đó dòng code trên có thể viết lại thành:

```
points = device_units_to_point(device_units)
```

Code đã trở nên dễ đọc hiểu hơn nhiều.

Ngoài ra có một lý do khác để đặt một phép toán nhỏ vào trong một hàm: Phép toán nhỏ có xu hướng trở thành một phép toán lớn hơn. Giải dụ như trong trường hợp trên *device_units_per_inch*() trả về giá trị 0, khi đó thay vì sửa lại mỗi dòng code sử dụng nó, ta chỉ cần sửa lại hàm đã viết:

```
def device_units_to_point(device_units):
    if device_units_per_inch() != 0:
        return device_units * (POINT_PER_INCH /      \
                    device_units_per_inch())
    else:
        return 0
```

KT7.3. You can classify routines into various kinds of cohesion, but you can make most routines functionally cohesive, which is best.

Bạn có thể phân chia các hàm, thủ tục ra làm nhiều loại gắn kết, nhưng sẽ tốt nhất nếu bạn có thể làm cho hầu hết các hàm "gắn kết theo chức năng".

Trong một hàm, sự gắn kết ám chỉ tới việc những phép toán bên trong hàm liên quan đến nhau như thế nào. Một vài lập trình viên sử dụng từ khác là "độ mạnh": Các phép toán trong hàm liên quan đến mạnh như thế nào?

Có một vài mức của sự gắn kết khi bàn luận về nó trong hàm:

- Sự gắn kết theo chức năng: Là loại gắn kết mạnh nhất, xảy ra khi hàm thực hiện một và chỉ một phép toán. Ví dụ như những hàm $sin()$, $get\_customer\_name()$, $erase\_file()$, $calculate\_loan\_payment()$, and $age\_from\_birthday()$.

- Sự gắn kết theo tuần tự: Là loại gắn kết tồn tại khi một hàm, thủ tục tồn tại những phép toán phải được thực hiện với một thứ tự nhất định, có sự chia sẻ dữ liệu từ bước này sang bước khác, và không tạo đủ chức năng nếu không hoạt động cùng nhau. Ví dụ như là một hàm tính toán tuổi của nhân viên và thời gian nghỉ hưu, với tham số là ngày tháng năm sinh. Nếu hàm đó sử dụng tuổi để tính toán thời gian nghỉ hưu, nó đã có sự gắn kết theo tuần tự. Nếu hàm đó tính toán tuổi và thời gian nghỉ hưu trong hai phép tính toán tách biệt nhau, nó sẽ chỉ có sự gắn kết nhờ giao tiếp.

- Sự gắn kết nhờ giao tiếp: Xảy ra khi các phép toán trong hàm sử dụng cùng dữ liệu mà không liên quan đến nhau theo bất cứ một cách nào. Chẳng hạn như một hàm in ra một báo cáo và sau đó khởi tạo lại giá trị của nó. Hai phép toán thao tác trên cùng một dữ liệu nhưng không liên quan gì tới nhau.

- Sự gắn kết tạm thời: Xảy ra khi một vài các phép toán được tổng hợp vào một hàm bởi vì chúng thực hiện cùng thời điểm. Ví dụ thông thương đó là các hàm như $startup()$, $shutdown()$, $create\_new\_employee()$.

- Sự gắn kết hướng thủ tục: Xảy ra khi những phép toán trong một hàm được thực hiện theo một thứ tự nhất định. Một ví dụ là một hàm lấy tên của nhân viên, địa chỉ, và rồi số điện thoại. Thứ tự của những phép toán này chỉ quan trọng bởi vì nó khớp với thứ tự mà người sử dụng hỏi từ màn hình. Để đạt được sự gắn kết tốt hơn, đặt những phép toán riêng rẽ vào thành một hàm, đảm bảo rằng bạn gọi hàm đó với một mục đích là làm một việc đơn nhất.

- Sự gắn kết theo logic: Xảy ra khi nhiều phép toán được nhồi vào cùng một hàm và chúng được lựa chọn nhờ vào một cờ điều khiển được truyền vào hàm. Chúng được gọi là gắn kết theo logic vì luồng điều khiển, hay "logic" là cái duy nhất buộc chúng lại với nhau – thường là một lệnh if hay một switch case lớn. Nó không hẳn là bởi vì các phép toán liên quan với nhau một cách logic.

- Sự gắn kết ngẫu nhiên: Xảy ra khi các phép toán trong hàm có thể thấy rõ ràng là không có liên quan gì với nhau.

Gần như là luôn luôn có thể tạo các hàm gắn kết theo chức năng, vậy nên cố gắng tập trung vào gắn kết theo chức năng để đặt hiểu quả cao nhất.

KT7.4. The name of a routine is an indication of its quality. If the name is bad and it's accurate, the routine might be poorly designed. If the name is bad and it's inaccurate, it's not telling you what the program does. Either way, a bad name means that the program needs to be changed.

Tên của một hàm, thủ tục sẽ chỉ định chất lượng của nó. Nếu tên của hàm là tồi và nó chính xác, hàm đó có thể đã được thiết kế kém. Nếu tên hàm đó là tồi và nó không chính xác, nó sẽ không nói cho bạn biết chương trình sẽ làm cái gì. Trong cả hai trường hợp, một tên tồi nghĩa rằng chương trình của bạn cần thay đổi.

Một vài những chỉ dẫn khi muốn tạo một tên có hiệu quả:

- Mô tả mọi thứ mà hàm, thủ tục đó thực hiện: Ví dụ $compute\_report\_totals()$ không mô tả được rõ hàm đó thực hiện những gì.
- Tránh những động từ vô nghĩa, không có nhiều tác dụng: Ví dụ như những hàm $handle\_calculation()$, $perform\_services()$.
- Tạo tên hàm dài nhất cần thiết: Tên một hàm trung bình là khoảng 15 đến 20 kí tự.
- Để đặt tên một hàm có giá trị trả về, sử dụng mô tả của giá trị trả về: Ví dụ $current\_color()$, $is\_ready()$.
- Để đặt tên một hàm không có giá trị trả về (hay thủ tục), sử dụng một động từ mạnh theo sau là một thực thể. Ví dụ như $print\_document()$, $check\_order\_info()$. Trong ngôn ngữ hướng đối tượng, bạn không cần phải bao gồm cả tên thực thể theo sau động từ vì bản thân thực thể đã đi trước lời gọi hàm. Ví dụ như $document.print()$, $order\_info.check()$. Tên giống như $document.print\_document()$ là dư thừa và có thể dẫn đến không chính xác. Ví dụ $Check$ là một class kế thừa từ $Document$, $check.print()$ sẽ nghĩa in ra một thông tin kiểm tra nào đó, nhưng $check.print\_document()$ thì không được như vậy.
- Sử dụng những từ trái ngược một cách chính xác. Ví dụ những cặp:
  - $add/remove$
  - $begin/end$
  - $create/destroy$
  - $first/last$

  - *get/set*

- Thành lập các quy ước cho các phép toán phổ biến. Ví dụ như:

```
employee.id.get()
dependent.get_id()
supervisor()
candidate()
```

  Là trường hợp mà một cách lấy id mà có nhiều phương pháp khác nhau. Gây ra những sự khó chịu, nhầm lẫn không đáng có.

KT7.5. Functions should be used only when the primary purpose of the function is to return the specific value described by the function's name.

Hàm có giá trị trả về nên được sử dụng khi mục đích chính của nó là trả về một giá trị cụ thể được mô tả bởi tên hàm đó.

Một vài những xem xét khi sử dụng hàm có giá trị trả về:

- Khi nào thì sử dụng hàm có hoặc không có giá trị trả về (hoặc thủ tục): Hàm nên thuần tuý chỉ trả về một giá trị, giá trị đó là phép tính toán mà hàm đó thực hiện. Có nghĩa là hàm chỉ nhận những tham số và trả về giá trị thông qua tên hàm đó mà thôi. Tên của hàm sẽ luôn luôn được đặt cho giá trị trả về của nó. Một hàm không có giá trị trả về (hay thủ tục) thì ngược lại, có thể nhận đầu vào, chỉnh sửa và xuất ra nhờ tham số. Một thực tế phổ biến là một hàm có thể hoạt động như một thủ tục nhưng lại có giá trị trả về. Một cách logic, nó giống một thủ tục (hàm không có giá trị trả về) nhưng vì nó trả về giá trị, nó chính thức là một hàm. Ví dụ như:

```
if report.format_output(formatted_report) == SUCCESS:
    ...
```

  Một cách thay thế cho những hàm này là:

```
report.format_output(formatted_report, output_status)
if output_status == SUCCESS:
    ...
```

- Thiết lập giá trị trả về cho hàm: Sử dụng hàm tạo một rủi ro rằng hàm đó sẽ trả về giá trị không chính xác. Nó thường xảy ra khi hàm có một vài phần mà một trong số đó không đặt giá trị trả về.
  - Kiểm tra toàn bộ những khả năng trả về. Khi tạo hàm, xem xét đầy đủ các trường hợp có thể để đảm bảo rằng hàm trả về những giá trị đã được định trước.
  - Đừng trả về tham chiếu hoặc con trỏ tới một giá trị trong hàm.

KT7.6. Careful programmers use macro routines with care and only as a last resort.

Lập trình viên cần sử dụng hàm macro một cách cẩn thận và chỉ sử dụng khi nó là phương án cuối cùng.

Trong C, nếu bạn sử dụng macro, hãy lưu ý những điều sau:

- Đặt các câu lệnh trong macro luôn luôn đi kèm với dấu ngoặc đơn.
- Đặt một macro nhiều câu lệnh trong cặp dấu ngoặc nhọn.
- Đặt tên macro giống như hàm để mà nó có thể thay thế bằng hàm khi cần thiết.

Sử dụng hàm macro có nhiều hạn chế. Như Bjarne Stroustrup, người thiết kế ra ngôn ngữ C++ nói rằng "Almost every macro demonstrates a flaw in the programming language, in the program, or in the programmer.... When you use macros, you should expect inferior service from tools such as debuggers, cross-reference tools, and profilers" – Gần như mọi macro thể hiện một lỗi hổng trong ngôn ngữ lập trình, trong chương trình, hoặc bên trong chính lập trình viên... Khi bạn sử dụng macro, bạn nên mong đợi vào dịch vụ cấp thấp của các công cụ như trình gỡ rối, công cụ tham chiếu chéo, và những hồ sơ của chương trình. Macro hữu dụng cho việc hỗ việc biên dịch có điều kiện, nhưng một lập trình viên cẩn thận sẽ chỉ sử dụng macro như một cách thay thế cho hàm khi nó là phương án cuối cùng.

III.3 Sử dụng chương 8 – Các kỹ thuật bẫy lỗi và phòng ngừa lỗi – Defensive Programming

Các kĩ thuật tiêu biểu gồm:

KT8.1.Production code should handle errors in a more sophisticated way than "garbage in, garbage out."

Khi code như là một sản phẩm thương mại thì nó nên xử lý những lỗi một cách tinh vi hơn chỉ là việc "rác vào thế nào, rác ra như vậy".

Bảo vệ chương trình của bạn từ những đầu vào không hợp lệ:

Trong một phần mềm thương mại, việc "rác vào thế nào, rác ra như vậy" không hề tốt. Một chương trình tốt sẽ không bao giờ để "rác đi ra", bất kể cái gì nó nhận vào. Một chương trình tốt sử dụng "rác vào, không gì xuất ra" hay "rác vào, thông báo lỗi xuất ra". Theo chuẩn ngày nay, "Rác vào thế nào, rác ra như vậy" đánh dấu cho một chương trình luộm thuộm, thiếu tính bảo mật.

Có ba cách để xử lý "rác" vào chương trình của bạn:

- Kiểm tra tất cả những giá trị của dữ liệu từ nguồn bên ngoài: ví dụ file, dữ liệu người dùng, mạng, hoặc một vài interface bên ngoài chương trình.

- Kiểm tra giá trị của tất cả tham số vào hàm.
- Quyết định xem làm thế nào để xử lý những giá trị đầu vào không hợp lệ.

Ví dụ đơn giản về kiểm tra tham số đầu vào:

```
def is_prime(n):
    a = int(n)  # make sure parameter is integer
    if a <= 1:
        return False
    for i in range(2, a):
        if a % i == 0:
            return False
    return True
```

KT8.2. Defensive-programming techniques make errors easier to find, easier to fix, and less damaging to production code.

Kỹ thuật lập trình phòng thủ làm cho những lỗi xảy ra dễ dàng được tìm thấy, được sửa chữa và gây hại ít hơn chương những đoạn code của chương trình.

KT8.3. Assertions can help detect errors early, especially in large systems, high-reliability systems, and fast-changing code bases.

Assertion - "Sự tuyên bố" có thể giúp phát hiện những lỗi sớm, đặc biệt trong một hệ thống lớn, độ tin cậy cao, thường xuyên thay đổi mã nguồn.

Một assertion là những dòng code được sử dụng trong khi phát triển phần mềm – thông thường nó là một hàm hoặc một macro – Cái mà cho phép chương trình được kiểm tra bản thân nó khi chạy. Khi một assert đúng, không có gì xảy ra. Khi nó sai, nó báo hiệu rằng một lỗi không mong đợi đã xảy ra. Assertion thực sự hữu dụng trong những chương trình lớn, phức tạp, độ tin cậy cao. Nó cho phép lập trình viên nhanh chóng tìm ra những sai sót, những thứ không phù hợp trong giả định của interface đang dùng, những lỗi xảy ra khi thay đổi code.

Assertion có thể được sử dụng để kiểm tra những giả định sau:

- Những tham số đầu vào có giá trị rơi vào khoảng xác định.
- File được mở (đóng) khi hàm bắt đầu thực hiện (hoặc khi kết thúc).
- Con trỏ file được đặt đúng vị trí khi bắt đầu (hoặc kết thúc) đọc file.
- File là chỉ đọc, chỉ ghi, hay cả đọc cả ghi.
- Con trỏ là khác NULL.
- Giá trị của biến đầu vào không bị thay đổi bởi hàm.

- Một array hoặc một kiểu dữ liệu chứa đựng có ít nhất số phần tử lớn hơn mức nào đó.
- Cái cấu trúc dữ liệu chứa đựng là trống rỗng (hoặc đầy) trước khi hoặc kết thúc hàm.
- ...

Như hàm ở kĩ thuật phía trên, ta sử dụng assert:

```
def is_prime(n):
    assert(isinstance(n, int))
    assert(n >= 2)
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

KT8.4 The decision about how to handle bad inputs is a key error-handling decision and a key high-level design decision.

Những quyết định về cách làm thế nào xử lý những đầu vào không hợp lệ là những quyết định chủ đạo trong việc xử lý lỗi và công việc thiết kế ở mức cao.

Bảo vệ hàm, chương trình của bạn khỏi những đầu vào không hợp lệ là bước đầu tiên và quan trọng nhất trong việc tránh những lỗi quan trọng xảy ra trong chương trình. Nó sẽ thúc đẩy những quyết định của bạn trong thiết kế hệ thống ra sảo để đảm bảo an toàn, từ đó tăng độ bảo mật cho chính hệ thống.

KT8.5. Exceptions provide a means of handling errors that operates in a different dimension from the normal flow of the code. They are a valuable addition to the programmer's intellectual toolbox when used with care, and they should be weighed against other error-processing techniques.

Những ngoại lệ cung cấp một cách thức xử lý lỗi mà trong đó hoạt động của nó xảy ra ở một chiều khác so với dòng chảy thông thường của code. Chúng là một công cụ thông minh, đầy giá trị của lập trình viên khi biết sử dụng nó cẩn thận, và chúng nên được xem xét sử dụng nhiều hơn so với các kỹ thuật xử lý lỗi khác.

Những lưu ý khi sử dụng exception:

- Sử dụng ngoại lệ để thông báo phần khác của chương trình về những lỗi mà nó không nên bỏ qua.
- Throw một ngoại lệ chỉ khi mà điều kiện đó thực sự là điều kiện đặc biệt.

- Đừng throw một ngoại lệ khi mà bạn có thể xử lý lỗi đó ngay trong hàm của mình.
- Tránh throw một ngoại lệ ở trong constructor và destructor trừ khi bạn catch nó ở cùng một nơi.
- Throw một ngoại lệ phải đúng mức trừu tượng.
- Gom tất cả những thông tin dẫn đến ngoại lệ vào trong bản thân ngoại lệ đó.
- Tránh tạo khối catch trống.
- Biết về những ngoại lệ mà thư viện của bạn throw.
- Xem xét về việc xây dựng một nơi báo cáo ngoại lệ tập trung.
- Chuẩn hoá cách sử dụng ngoại lệ trong project của bạn.
- Xem xét việc thay thế ngoại lệ bằng một cách khác.

Như ví dụ ở trên, ta thay thế nó bằng exception:

```
def is_prime(n):
    if not isinstance(n, int):
        raise TypeError("n is not an integer")
    if n < 2:
        raise ValueError("n is less than 2")

    for i in range(2, n):
        if n % i == 0:
            return False
    return True
try:
    print(is_prime(1))
except TypeError as e:
    print("Type error exception!")
    print(e)
except ValueError as e:
    print("Value error exception!")
    print(e)
```

KT8.6. Constraints that apply to the production system do not necessarily apply to the development version. You can use that to your advantage, adding code to the development version that helps to flush out errors quickly.

Những ràng buộc áp dụng cho hệ thống được xuất ra thị trường không nhất thiết áp dụng cho phiên bản đang phát triển. Bạn có thể sử dụng những điều đó cho lợi thế của

mình, thêm những code vào phiên bản đang phát triển của phần mềm sẽ giúp cho bạn nhanh chóng tìm ra những lỗi xảy ra với hệ thống.

Phiên bản xuất ra thị trường phải chạy nhanh, nhưng phiên bản để phát triển có thể chạy chậm. Phiên bản hoạt động thực tế có thể rất khắt khe khi xử dụng tài nguyên, phiên bản đang phát triển có thể sử dụng cực kì nhiều tài nguyên. Phiên bản xuất ra không nên xuất hiện những phương thức gây hại cho hệ thống, phiên bản phát triển có thể có những phương thức thêm vào mà không đảm bảo độ an toàn.