

Mục lục

I	Tổng hợp kỹ thuật viết mã nguồn với các cấu trúc lập trình	2
I.1	Sử dụng chương 14 – Tổ chức các câu lệnh tuần tự.	2
I.2	Sử dụng chương 15 – Sử dụng câu lệnh If-Then	2
I.3	Sử dụng chương 16 – Vòng lặp	2
I.4	Sử dụng chương 17 – Các cấu trúc điều khiển khác	3
II	Tổng hợp kỹ thuật làm việc với các biến	3
II.1	Sử dụng chương 10 – Các kỹ thuật chung làm việc với biến	3
II.2	Sử dụng chương 11 – Kỹ thuật đặt tên biến	4
II.3	Sử dụng chương 12 – Các kiểu dữ liệu cơ bản	4
II.4	Sử dụng chương 13 – Các kiểu dữ liệu đặc biệt	7
III	Tổng hợp kỹ thuật xây dựng chương trình, hàm, thủ tục	9
III.1	Sử dụng chương 5 – Các kỹ thuật thiết kế chương trình phần mềm – Design in Construction	9
III.2	Sử dụng chương 7 – Kỹ thuật xây dựng hàm/thủ tục High-Quality Routines	14
III.3	Sử dụng chương 8 – Các kỹ thuật bắt lỗi và phòng ngừa lỗi – Defensive Programming	15

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.

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.

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

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

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.

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

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.

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 kỹ 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 kỹ 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 nghĩ 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

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

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.

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

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

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

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.

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

Tránh sử 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 vớt 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.

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 kỹ 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 kỹ 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 kỳ 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 động 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 gọi 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 tổn 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 giải 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 kế 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 kỹ 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?", điều 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 kỳ 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 kỹ 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.

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

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

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.

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.

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

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

Các kỹ thuật tiêu biểu gồm:

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

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

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

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

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.

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.