• Lê Văn Cường – 14521116

• Phạm Hoài Nguyên – 14520605

• Phan Quang Duy – 14521161

**BÀI TẬP CHƯƠNG 4:**

Bài tập trang 21:

- Tìm hiểu các kỹ thuật refactoring

Một định nghĩa ngắn về **cải tiến mã nguồn** (*refactoring*) trong [lập trình](http://vi.wikipedia.org/wiki/L%E1%BA%ADp_tr%C3%ACnh) là làm hoàn thiện hơn thiết kế [mã nguồn](http://vi.wikipedia.org/wiki/M%C3%A3_ngu%E1%BB%93n) phần mềm đã có sẵn nhằm nhiều mục đích. Refactoring là cải tiến và làm tốt hơn chất lượng của mã nguồn trong một [ứng dụng](http://vi.wikipedia.org/w/index.php?title=%E1%BB%A8ng_d%E1%BB%A5ng&action=edit&redlink=1). Nó không làm thay đổi các chức năng chính, chức năng chung của ứng dụng, nhưng nó làm cho ứng dụng dễ bảo trì hơn, dễ phát triển hơn trong tương lai.

* Làm cho mã nguồn dễ đọc hơn. Sữa đổi định danh, từ ngữ, cách đặt tên cho các thành phần trong mã nguồn. Còn được gọi là chuẩn hóa từ ngữ (*coding convention*).
* Làm cho mã nguồn dễ hiểu hơn. Sắp xếp lại trật tự các dòng lệnh, các vòng lặp, các điều kiện, ràng buộc nhằm làm cho logic của mã nguồn tốt hơn, số lượng dòng lệnh (*line of code*) được [cực tiểu hóa](http://vi.wikipedia.org/w/index.php?title=C%E1%BB%B1c_ti%E1%BB%83u_h%C3%B3a&action=edit&redlink=1).
* [Tối ưu hóa](http://vi.wikipedia.org/wiki/T%E1%BB%91i_%C6%B0u_h%C3%B3a) xử lý. Việc sắp đặt lại các logic luồng làm việc của mã nguồn giúp cho luồng xử lý rõ ràng hơn và tránh các sai sót (*bug*).
* Tối ưu hóa tốc độ. Số lượng vòng lặp trong mã nguồn giảm, thời gian xử lý, tìm kiếm kết quả của các luồng xử lý tăng lên.
* Tăng [tính dùng lại](http://vi.wikipedia.org/w/index.php?title=T%C3%ADnh_d%C3%B9ng_l%E1%BA%A1i&action=edit&redlink=1): mã nguồn tốt, rõ ràng sẽ có lợi khi được sử dụng lại cho các module khác của cùng ứng dụng hoặc được dùng như một bộ thư viện sử dụng cho nhiều ứng dụng, module khác nhau.
* Tăng tính tiến hóa: Một mã nguồn tốt có lợi ích và chu kỳ sống cụ thể do [công nghệ thông tin](http://vi.wikipedia.org/wiki/C%C3%B4ng_ngh%E1%BB%87_th%C3%B4ng_tin) ngày càng phát triển. Mã nguồn tốt có thể có thời gian sử dụng lâu hơn và khả năng tự phát triển, [nâng cấp](http://vi.wikipedia.org/w/index.php?title=N%C3%A2ng_c%E1%BA%A5p&action=edit&redlink=1), [kế thừa](http://vi.wikipedia.org/w/index.php?title=K%E1%BA%BF_th%E1%BB%ABa&action=edit&redlink=1) khi ứng dụng có nhu cầu phát triển thêm mà không phải bị vứt bỏ để viết lại từ đầu.
* Tăng tính gần gũi với người dùng. Có những ứng dụng hay, nhưng lại phức tạp cho người dử dụng hay người đọc. Chẳng hạn như phần [giao tiếp người dùng](http://vi.wikipedia.org/w/index.php?title=Giao_ti%E1%BA%BFp_ng%C6%B0%E1%BB%9Di_d%C3%B9ng&action=edit&redlink=1) (*user interface*) cần được cải thiện để tăng tính dễ dùng, dễ hiểu, linh hoạt hơn và làm cho giao tiếp người dùng sử dụng được hết khả năng của mã nguồn cung cấp.

Refactoring không hẳn làm thay đổi các cư xử, hoạt động bên ngài của [phần mềm](http://vi.wikipedia.org/wiki/Ph%E1%BA%A7n_m%E1%BB%81m). Chủ yếu là cải thiện phần cấu trúc bên trong nhằm làm tối ưu chức năng của phần mềm để phần mềm xử lý, nhanh hơn, tốt hơn, an toàn hơn và cố thể phù hợp với nhiều môi trường hoặc thay đổi mới cho người dùng trong quá trình sử dụng. Giảm thiểu những sai sót và tăng thời gian sống cho phần mềm. Là một bước không thể thiếu và có thể được áp dụng trong suốt các quá trình phát triển phần mềm.

**Exercise 6.1 Mục tiêu đạt được của bạn là gì khi cố găng hiểu chương trình**

* + Phạm vi vấn đề
  + Hiệu quả thực thi
  + Mối liên hệ Nhân – Quả (Cause-Effect)
  + Mối liên hệ sản phẩm – Môi trường
  + Đặc trưng Quyết định – Hỗ trợ

Chúng ta dành phần lớn thời gian của mình để đọc chương trình có sẵn và cố gắng hiểu ý nghĩa của chương trình. Tiếp sau là chúng ta chỉnh sửa lại những dòng code có sẵn đó, và cuối cùng thời gian chúng ta phát triển những đoạn chương trình mới hoàn toàn là vô cùng ít ỏi trển tổng thời gian làm việc. Phần lớn thời gian làm việc của một lập trình viên là dùng để đọc hiểu code. Công cụ nên được cung cấp nhiều có thể để hỗ trợ khám phá thông tin cho việc hiểu chương trình. Tư động hoá thì thích hợp hơn nhưng tự động hoàn toàn thì hiển nhiên không khả thi

**Exercise 6.2 Tại sao hiểu chương trình là quan trọng?**

* Phạm vi vấn đề

Nắm bắt được kiến thức phạm vi khá quan trọng. Tác động đến vấn đề vùng phạm vi chuyên biệt, cụ thể.

Hiệu quả thực thi

* Ở mức cao trừu tượng, nhân sự bảo trì cần phải nắm (dự đoán) kết quả chương trình sẽ được phát sinh kết quả gì từ đầu vào được cho mà không cần biết đơn vị chương trình được xây dựng để có kết quả tổng thể và kết quả được cho như thế nào.
* Ở mức thấp, họ cần biết kết quả mỗi đơn vị chương trình sẽ được tạo và thực thi.
* Kiến thức data flow, control flow, và thuật toán có thể thuận tiện hoàn thành thực thi mục tiêu này.

**Exercise 6.3 Giả sử bạn là lập trình viên, bạn được yêu cầu như sau (i) cung cấp tiện ích quản lý thông điệp cho hệ thống vận hành quản lý thông tin (MIS), và (ii) tích hợp hệ thống MIS vào gói văn phòng tự động. Những thông tin về MIS bạn cần làm gì, có tác động đến thay đổi không? Chỉ ra lý do**

1. Phân tích và tổng hợp info để quyết định restructure hay rewrite phân đoạn chương trình cụ thể hay không
2. Xác định tính khả thi của những thay đổi đề xuất và cho thông báo cấp quản lý bất kỳ những vấn đề thấy trước.

**Exercise 6.5: Liệt kê những loại khác nhau của chiến lược hiểu chương trình, phân biệt giữa chúng?**

Về đặc tính nguyên lý của mô hình :

Mô hình Top-down : Sự hiểu biết của chương trình là ánh xạ từ cách chương trình hoạt động đến những gì cần thực hiện.

Mô hình Bottom-up: Công nhận các mẫu lặp lại trong mã chương trình

Mô hình Opporttunistic: Kết hợp cả chiến lược từ trên xuống (top-down) và từ dưới lên( bottom-up)

Về đặc tính quá trình nhận thức:

Mô hình Top-down: Xây dựng lại các lĩnh vực kiến thức từ trên xuống và các ánh xạ của họ. Tái thiết dựa trên việc tạo ra giả thuyết, xác nhận và chu trình tinh chỉnh.

Mô hình Bottom-up : Mảnh của các mô hình được công nhận để sản xuất cấu trúc ngữ nghĩa cao cấp.

Mô hình Opporttunistic: top-down và bottom-up là các tín hiệu được khai thác khi sẵn sàng.

**Exercise 6.6: Những chiến lược gì bạn đã dùng và trong những hoàn cảnh nào?**

Nhóm chúng em đều đã sử dụng qua 3 chiến lược :

Top-down

Bottom-up

Opportunistic

**Exercise 6.7 Liệt kê và giải thích các yếu tố chính tác động đến việc hiểu một chương trình**

Phạm vi kiến thức: Chuyên gia, Vấn đề, Ứng dụng, hệ thống.

Thực nghiệm chương trình, vấn đề thực thi: Độ phân rã, tính môđun, tính che dấu thông tin, Thuật toán, Chương trình, cách đặt tên, ghi chú.

Tài liệu: bên ngoài, bên trong tổ chức.

Tổ chức/ thuyết trình.

Công cụ hỗ trợ nắm bắt thông tin: công cụ phân tích tĩnh/ động.

**Exercise 6.7 Liệt kê và giải thích các yếu tố chính tác động đến việc hiểu một chương trình**

* Phạm vi kiến thức: Chuyên gia, Vấn đề, Ứng dụng, hệ thống

Người lập trình càng có kinh nghiệm phạm vi ứng dụng với ngôn ngữ lập trình, càng dễ và nhanh chóng hiểu chương trình và cũng như toàn bộ hệ thống hiệu quả

* Thực nghiệm chương trình, vấn đề thực thi: Độ phân rã, tính môđun, tính che dấu thông tin, Thuật toán, Chương trình, cách đặt tên, ghi chú
* Kiểu/ cách thức đặt tên
* Ghi chú chương trình
* Cơ chế phân rã:

Phân rã mô đun

Lập trình có cấu trúc

* Tài liệu: bên ngoài, bên trong tổ chức

Tài liệu hệ thống rất hữu ích và quan trọng bởi nó không chỉ đầu mối liên lạc tác giả gốc của hệ thống thông tin.

Đó là phần báo cáo trang trọng trong công nghiệp phần mềm: từ dự án khác, phòng ban, và công ty khác. Như vậy, khi người bảo trì cần truy xuất vào hệ thống tài liệu để có thể hiểu chức năng, thiết kế, thực thi vàvấn đế liên quan đến bảo trì thành công.

Đôi khi, tài liệu hệ thống không chính xác, quá lỗi thời chưa cập nhật. Trong trường hợp như vậy, người bảo trì phải thường xuyên xem sưu liệu nội bộ với chính chương trình nguồn – ghi chú chương trình.

* Tổ chức/ thuyết trình:

Thuyết minh chương được cải tiến có thể cải thiện khả năng hiểu chương trình:

Thuận tiện biểu thức rõ ràng và chính xác mô hình chương trình và truyền thông của các mô hình này đối với người đọc chương trình

Nhấn mạnh đến luồng kiểm soát, cấu trúc phân cấp chương trình và tính logic và tổng hợp của người lập trình – mục đích gạch dưới cấu trúc và cải thiện tính dễ nhìn của chương trình nguồn qua cách sử dụng ngắt dòng, khoảng trắng, khối và tô bóng

* Công cụ hỗ trợ nắm bắt thông tin: công cụ phân tích tĩnh/ động

Có công cụ được sử dụng để tổ chức và thể hiện chương trình nguồn theo cách thực hiện càng rõ ràng càng dễ đọc và như vậy càng dễ hiểu.

'Book Paradigm' là pretty-printer, static analyser và browser.

Nhiều công cụ đọc hiểu được thiết kế phục vụ trợ giúp cho người đọc hiểu, tăng tốc độ, qui trình hiểu. Tuy nhiên, đầu ra của công cụ này không cung cấp sự giải thích chức năng của mỗi thành phần. Ở đây mô tả Book Paradigm và một số đặc chưng của nó

**Exercise 6.8 Bạn có thể cải thiện khả năng đọc hiểu chương trình Hình 6.10 bằng những cách xử lý nào?**

Phân các dòng code theo từng lệnh rõ ràng

**Exercise 6.9 Liệt kê tất cả công cụ bảo trì đã có trong hệ thống của bạn. Có gắng thử 3 trong những công cụ này, với mỗi loại chức năng chính của chúng là gì và làm thể nào nó cải thiện khả năng đọc.**

1. Khả năng nắm bắt chương trình và reverse engineering.

Tranformation Tool

Dependence Analyzer

Data flow Analyzer

Program Slicer

1. Kiểm thử.

Simulator

Generator

1. Quản lý cấu hình.

Source Code Control System

**Exercise 6.10 Tại sao quan trọng đối với người bảo trì thu được hiểu biết tốt chiến lược nắm bắt chương trình khác nhau và vấn đề dựa trên kinh nghiệm**

Có một chiến lượt nắm bắt chương trình tốt, và một kinh nghiệm dày dặn là điều cần thiết của người làm công việc bảo trì. Bảo trì phần mềm hay bất cứ các khâu nào trong công nghệ phần mềm đều cần đến kinh nghiệm, bởi thực nghiệm cho thấy, những vấn đề bảo trì ngoài những phát sinh mới thì hầu hết còn lại là những việc bạn đã thực hiện, những lỗi bạn đã từng fixed. Kinh nghiệm dày dặn cho bạn cách tổ chức công việc hợp lí, xử lí nhanh nhất, nó cũng là một liều thuốc tâm lí cho bạn. Đặt lại vấn đề xử lí bảo trì cho hệ thống thư viện chỉ 2 giờ, toàn bộ thời gian này chỉ đủ cho bạn…bấn tít mù lên vì chẳng biết phải bắt đầu làm gì, làm thế nào, và lúc đó hiểu được cái giá của kinh nghiệm. Ít nhất nó cho bạn sự bình tĩnh và cảm giác “sẽ làm được”- thứ rất xa xỉ với những người non kinh nghiệm. Người có kinh nghiệm hiểu được vấn đề ở đây trong khi người non kinh nghiệm bấn rối. Người có kinh nghiệm tự hoạch định được sẽ làm gì, từng bước, trong khi người non kinh nghiệm chẳng biết được mình có làm được hay không?

Một điều đáng suy ngẫm cho việc bảo trì. Trải qua và học lấy, để có kinh nghiệm.

**Exercise 7.1 Giải thích khác nhau giữa những loại khác nhau kỹ thuật reverse engineering và cho ví dụ thích hợp.**

Reverse Engineering là thao tác phân tích ở mức thấp một hệ thống hoặc phần mềm từ đó hiểu được kiến trúc, dữ liệu, các chức năng và cách thức thực hiện các chức năng đó. Các thao tác cụ thể như sau:

- Phân tích mã nhị phân của phần mềm

- Theo dõi các thông tin tương tác (vào/ra) của phần mềm so với thế giới bên ngoài

Reverse Engineering (SE) là một kỹ thuật đối lập với Forward Engineering- một phương pháp tiếp cận phân tích hệ thống phần mềm theo hướng truyền thống. SE thường được thực hiện theo 2 kỹ thuật là Tái cấu trúc và Tái cơ cấu.

Tái cơ cấu có thể hiểu hiểu là cách dịch ngược chương trình theo hướng không thay đổi chức năng và ngữ nghĩa từ hình thức cũ sang một hình thức mới. Tuy nhiên trong hình thức này thì chương trình được hiểu sau khi Tái cơ cấu sẽ không đảm bảo được sự nguyên vẹn, tạo nên những sự thoái hóa và phức tạp. Do đó, cần phải cơ cấu lại một cách hợp lí hơn theo một tiêu chuẩn nhất định.

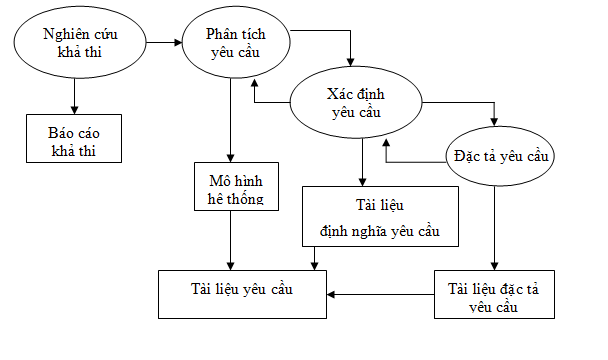
Tái cấu trúc, đây là quá trình xem xét và thay đổi một hệ thống mục tiêu để thực hiện một sửa đổi mong muốn. Tái cấu trúc bao gồm hai bước. Thứ nhất, kỹ thuật đảo ngược được áp dụng cho hệ thống đích để hiểu nó và thể hiện nó trong một hình thức mới. Thứ hai, kỹ thuật chuyển tiếp được áp dụng, thực hiện và tích hợp bất kỳ yêu cầu mới, do đó dẫn đến một hệ thống mới và nâng cao. Nó cung cấp một cơ chế cho phép nhà duy trì làm việc thông qua mã và hiểu mục đích của nó, sản xuất tài liệu và sửa đổi mã theo cách tăng cường bảo trì của nó, đồng thời hỗ trợ đặc tả lúc trước của một hệ thống từ source code có sẵn.

**Exercise 7.2 Thực hiện khôi phục đặc tả và thiết kế trên tất cả hay các phần của hệ thống phần mềm mà bạn không quen (hệ thống nên có ít nhất 2K dòng code độ lớn).**

**o Những kỹ thuật bạn dùng nhận diện đặc tả và thiết kế là gì và tại sao?**

**o Những hình thức thể hiện bạn xem là phù hợp cho những tác vụ này là gì? Chỉ ra lý do.**

**o Bài học kinh nghiệm mà bạn đã học được trong các công việc này là gì?**

****

Các nguyên lý phân tích

Trên hai thập kỉ qua, người ta đã xây dựng ra một số phương pháp phân tích và đặc tả phần mềm. Những người nghiên cứu đã xác định ra các vấn đề và nguyên nhân của chúng, và đã xây dựng ra các qui tắc và thủ tục để vượt qua chúng. Mỗi phương pháp đều có kí pháp và quan điểm riêng. Tuy nhiên, tất cả các phương pháp này đều có quan hệ với một tập hợp các nguyên lý cơ bản:

Miền thông tin của vấn đề phải được biểu diễn lại và hiểu rõ.

Các mô hình mô tả cho thông tin, chức năng và hành vi hệ thống cần phải được xây dựng.

Các mô hình (và vấn đề) phải được phân hoạch theo cách để lộ ra các chi tiết theo kiểu phân tầng (hay cấp bậc).

Tiến trình phân tích phải đi từ thông tin bản chất hướng tới chi tiết cài đặt. Bằng cách áp dụng những nguyên lý này, người phân tích tiếp cận tới vấn đề một cách hệ thống.

Miền thông tin cần được xem xét sao cho người ta có thể hiểu rõ chức năng một cách đầy đủ. Các mô hình được dùng để cho việc trao đổi thông tin được dễ dàng theo một cách ngắn gọn. Việc phân hoạch vấn đề được sử dụng để làm giảm độ phức tạp. Những cách nhìn nhận cả từ góc độ bản chất và góc độ cài đặt về phần mềm đều cần thiết để bao hàm được các ràng buộc logic do yêu cầu xử lý áp đặt nên cùng các ràng buộc vật lý do các phần tử hệ thống khác áp đặt nên.

Đặc tả yêu cầu

Tài liệu xác định yêu cầu là mô tả hướng khách hàng và được viết bởi ngôn ngữ của khách hàng. Khi đó có thể dùng ngôn ngữ tự nhiên và các khái niệm trừu tượng. Tài liệu dặc tả yêu cầu (đặc tả chức năng) là mô tả hướng người phát triển, là cơ sở của hợp đồng làm phần mềm. Nó không được phép mơ hồ, nếu không sẽ dẫn đến sự hiểu nhầm bởi khách hàng hoặc người phát triển. Với một yêu cầu mơ hồ thì người phát triển sẽ thực hiện nó một cách rẻ nhất còn khách hàng thì không muốn vậy. Do đó khách hàng có thể đòi hỏi sửa đổi chức năng phần mềm khi nó đã gần hoàn thiện khiến cho chi phí tăng và chậm thời điểm bàn giao. Chi phí cho sửa các sai sót trong phát biểu yêu cầu là rất lớn, đặc biệt là khi các sai sót này được phát hiện khi đã bắt đầu xây dựng hệ thống. Theo một số thống kê thì 85% mã phải viết lại do thay đổi yêu cầu và 12% lỗi phát hiện trong 3 năm đầu sử dụng là do đặc tả yêu cầu không chính xác. Do đó, việc đặc tả chính xác yêu cầu là mối quan tâm được đặt lên hàng đầu. Có hai phương pháp đặc tả là

Đặc tả phi hình thức: là cách đặc tả bằng ngôn ngữ tự nhiên

Đặc tả hình thức: là cách đặc tả bằng các ngôn ngữ nhân tạo (ngôn ngữ đặc tả), các công thức và biểu đồ

Đặc tả phi hình thức (ngôn ngữ tự nhiên) thuận tiện cho việc xác định yêu cầu nhưng nhiều khi không thích hợp với đặc tả yêu cầu vì:

Không phải lúc nào người đọc và người viết đặc tả bằng ngôn ngữ tự nhiên cũng hiều các từ như nhau.

Ngôn ngữ tự nhiên quá mềm dẻo do đó các yêu cầu liên quan đến nhau có thể được biểu diễn bằng các hình thức hoàn toàn khác nhau và người phát triển không nhận ra các mối liên quan này.

Các yêu cầu khó được phân hoạch một cách hữu hiệu do đó hiệu quả của việc đổi thay chỉ có thể xác định được bằng cách kiểm tra tất cả các yêu cầu chứ không phải một nhóm các yêu cầu liên quan.

Các ngôn ngữ đặc tả (đặc tả hình thức) khắc phục được các hạn chế trên, tuy nhiên đa số khách hàng lại không thông thạo các ngôn ngữ này. Thêm nữa mỗi ngôn ngữ đặc tả hình thức thường chỉ phục vụ cho một nhóm lĩnh vực riêng biệt và việc đặc tả hình thức là một công việc tốn kém thời gian.

Một cách tiếp cận là bên cạnh các đặc tả hình thức người ta viết các chú giải bằng ngôn ngữ tự nhiên để giúp khách hành dễ hiểu.

**Exercise 7.3 Một ngân hàng có substantial investment trong hệ thống phần mềm viết bằng Cobol ít nhất 1 triệu dòng code và chạy trên 20 năm. Nó được dùng cơ bản mỗi ngày để thực thi thao tác khác nhau như quản lý tài khoản khách hàng và loans. Sau vài năm cập nhật, cả dự định và không hoạch định – hệ thống trở nên quá đắt tiền để bảo trì. Kết quả là, ngân hàng muốn vài lời khuyên ở các bước tiếp để làm.**

**o Giả sử bạn được thuê làm việc như nhân viên tư vấn bảo trì.**

**o Bạn sẽ cho Ngân hàng lời khuyên gì?**

**o Chỉ ra lý do cho bất kỳ đề nghị mà bạn đã thực hiện**

Một hệ thống lớn bao giờ cũng cần rất nhiều thời gian, nhân lực, cũng như gặp nhiều rủi ro trong quá trình bảo trì. Do đó, chắc chắn hệ thống đó phải được bảo trì định kỳ để giảm rủi ro và tài nguyên hao tốn. Nhưng sau một thời gian, dù được bảo trì định kì thì hệ thống cũng sẽ trở nên nặng nề, cồng kềnh và việc bảo trì càng ngày càng mất nhiều thời gian, công sức cũng như tiền bạc, nhiều rủi ro hơn. Thực sự thì em không biết Cobol là ngôn ngữ có ưu nhược điểm thế nào, và chưa tưởng tượng ra được khoảng 1tr dòng code là nhiều hay ít cho một phần mềm có tính năng tương tự khi viết bằng các ngôn ngữ khác mà em biết. Nhưng nếu được đề xuất, em nghĩ có thể thay thế bằng một hệ thống khác, có sức chịu tải hơn, dữ liệu có thể được convert lại để phù hợp với cấu trúc hệ thống mới. Em hoàn toàn tán đồng nếu ai đó bảo ý kiến thế là rất phí phạm, nhưng em nghĩ, hao tốn dành cho việc bảo trì hệ thống cũ cũng ngang ngửa hoặc hơn cho việc đầu tư hệ thống mới