Phần II

Tạo ra Mã nguồn Chất lượng Cao

Trong phần này:

- **Chương 5:** Thiết kế trong Giai đoạn Xây dựng
- **Chương 6:** Các Lớp Làm việc
- **Chương 7:** Các Thủ tục Chất lượng Cao
- **Chương 8:** Lập trình Phòng thủ (Defensive Programming)
- **Chương 9:** Quy trình Lập trình Bằng Pseudocode (Pseudo code Programming Process)

Chương 5

Thiết kế trong Giai đoạn Xây dựng

Mục lục

Chủ đề liên quan

- *Kiến trúc phần mềm*: Mục 3.5
- *Các lớp làm việc*: Chương 6
- *Đặc điểm của các thủ tục chất lượng cao*: Chương 7
- *Lập trình phòng thủ (Defensive Programming)*: Chương 8
- *Refactoring*: Chương 24
- *Ảnh hưởng của kích thước chương trình tới giai đoạn xây dựng*: Chương 27

Một số người có thể cho rằng thiết kế không thực sự là một hoạt động trong xây dựng, nhưng ở các dự án nhỏ, nhiều hoạt động thường được xem như là xây dựng, bao gồm cả thiết kế. Ở một số dự án lớn hơn, một kiến trúc phần mềm mang tính chính thức có thể chỉ đề cập tới các vấn đề ở cấp độ hệ thống, và nhiều công việc thiết kế có chủ ý được để lại cho giai đoạn xây dựng. Ở những dự án lớn khác, thiết kế có thể được dự kiến đủ chi tiết để việc lập trình trở nên cơ học, nhưng thực tế hiếm khi như vậy—lập trình viên thường phải tự thiết kế một phần chương trình, dù chính thức hay không chính thức.

Ở các dự án nhỏ, phần lớn công việc thiết kế diễn ra khi lập trình viên đang ngồi trước bàn phím. “Thiết kế” có thể chỉ đơn giản là viết ra giao diện một class bằng pseudocode trước khi viết các chi tiết, hoặc có thể là vẽ sơ đồ cho vài mối quan hệ class trước khi lập trình. Đôi khi, việc hỏi ý kiến lập trình viên khác về lựa chọn pattern (mẫu thiết kế) cũng là một hình thức thiết kế. Dù thực hiện cách nào, các dự án nhỏ vẫn hưởng lợi từ việc thiết kế cẩn thận giống như các dự án lớn, và việc xác định thiết kế như một hoạt động độc lập sẽ tối đa hóa lợi ích thu nhận được từ nó.

Thiết kế là một chủ đề rất rộng, vì vậy chương này chỉ đề cập tới một số khía cạnh nhất định. Phần lớn việc thiết kế class hoặc thủ tục tốt sẽ được xác định bởi kiến trúc hệ thống, do đó hãy đảm bảo rằng kiến trúc tiền đề nêu ở Mục 3.5 đã được hoàn thiện. Thậm chí nhiều công việc thiết kế hơn nữa sẽ được thực hiện ở cấp độ từng class và thủ tục riêng lẻ, được mô tả trong Chương 6 “Các Lớp Làm việc” và Chương 7 “Các Thủ tục Chất lượng Cao”. Nếu bạn đã quen thuộc với các chủ đề thiết kế phần mềm, bạn có thể chỉ cần tham khảo qua các điểm chính ở các mục về thách thức thiết kế (5.1) và các heuristic quan trọng (5.3).

5.1 Thách thức trong Thiết kế

"Thiết kế phần mềm" là việc hình thành, sáng tạo hoặc phát minh ra một sơ đồ để chuyển đổi một yêu cầu phần mềm thành phần mềm khả thi. Thiết kế là hoạt động kết nối yêu cầu với hoạt động lập trình và gỡ lỗi (debugging). Một thiết kế cấp cao tốt sẽ cung cấp cấu trúc đủ an toàn để chứa nhiều thiết kế chi tiết hơn ở các cấp thấp. Thiết kế tốt hữu ích cho các dự án nhỏ và không thể thiếu đối với các dự án lớn.

Tuy nhiên, thiết kế cũng đặt ra nhiều thách thức, được trình bày dưới đây.

Thiết kế là một "Vấn đề nan giải" (Wicked Problem)

Horst Rittel và Melvin Webber đã định nghĩa "vấn đề nan giải" (wicked problem) là vấn đề chỉ có thể được xác định rõ ràng khi đã giải xong nó hoặc đã giải được một phần của nó (1973). Nghịch lý này có nghĩa là bạn phải "giải" vấn đề một lần để hiểu rõ nó, rồi lại giải tiếp lần nữa để có được giải pháp hoàn chỉnh. Quá trình này được coi là kinh điển trong phát triển phần mềm suốt nhiều thập kỷ (Peters và Tripp, 1976).

Một ví dụ điển hình về vấn đề nan giải là thiết kế cây cầu Tacoma Narrows nguyên bản. Khi xây dựng, yếu tố chính được xem xét chỉ là đảm bảo cầu đủ chắc chắn để chịu tải trọng dự kiến. Tuy nhiên, trong trường hợp cầu Tacoma Narrows, gió đã tạo nên một rung động ngang bất ngờ. Đến một ngày gió lớn năm 1940, sự cộng hưởng gia tăng đến mức cầu sập hoàn toàn. Chỉ sau khi cây cầu sụp đổ, các kỹ sư mới nhận ra rằng tính khí động học cần được cân nhắc kỹ hơn—bài học chỉ rút ra khi đã "giải xong" bài toán một lần.

Đây là điểm khác biệt chính giữa các chương trình bạn phát triển khi còn đi học và khi làm việc chuyên nghiệp. Các bài tập lập trình ở trường hầu như không bao giờ đối mặt với những vấn đề mang tính "nan giải" như vậy—thường lập trình viên chỉ tiến thẳng từ đầu đến cuối. Trong thực tế, việc yêu cầu thiết kế, rồi thay đổi yêu cầu sau khi lập trình viên đã hoàn thành thiết kế, thậm chí lại đổi tiếp trước khi nộp sản phẩm, chính là thực tại thường xuyên trong lập trình chuyên nghiệp.

Quá trình Thiết kế thường không Gọn gàng (Dù kết quả có thể gọn gàng)

Thiết kế phần mềm đã hoàn thiện nên trông mạch lạc và sạch sẽ, nhưng quá trình tạo ra nó thường không hề như vậy.

Nguyên nhân là vì trong quá trình thiết kế, bạn sẽ mắc nhiều sai lầm và thử nghiệm các hướng đi không đúng—lỗi là điều tất yếu, và thực tế việc phạm lỗi trong thiết kế lại rẻ hơn rất nhiều so với khi phạm cùng lỗi ở giai đoạn hoàn thiện mã nguồn. Giải pháp tốt thường chỉ khác biệt rất tinh tế so với giải pháp chưa tốt.

Thiết kế là việc Cân nhắc và Ưu tiên

Trong thế giới lý tưởng, hệ thống nào cũng chạy tức thì, không sử dụng bộ nhớ, không cần băng thông mạng, không có lỗi, và tốn chi phí bằng 0 để xây dựng. Tuy nhiên, ở thế giới thực, nhiệm vụ của nhà thiết kế là cân nhắc giữa các đặc điểm mâu thuẫn và xác định điểm cân bằng hợp lý. Nếu tốc độ phản hồi quan trọng hơn thời gian phát triển, nhà thiết kế sẽ lựa chọn giải pháp khác so với trường hợp ưu tiên rút ngắn thời gian phát triển.

Thiết kế bao gồm việc Đặt ra các Ràng buộc

Mục tiêu của thiết kế không chỉ là tạo ra các khả năng mà còn đặt ra các giới hạn. Nếu có vô hạn thời gian, tài nguyên, không gian, thì các công trình vật lý có thể rất phức tạp, lộn xộn. Phần mềm cũng vậy nếu không tự đặt ra các giới hạn một cách có chủ ý—chính các giới hạn về tài nguyên đã buộc giải pháp phải đơn giản hóa, từ đó thường dẫn đến giải pháp tốt hơn. Thiết kế phần mềm cũng nên hướng tới điều tương tự.

Thiết kế là Quá trình Không xác định (Nondeterministic)

Nếu bạn cho ba người cùng thiết kế một chương trình, họ có thể sẽ đưa ra ba giải pháp rất khác nhau, và cả ba đều có thể hoàn toàn chấp nhận được. Có rất nhiều cách để thiết kế phần mềm.

Thiết kế là Quá trình Heuristic

Do thiết kế mang tính không xác định, các kỹ thuật thiết kế thường là heuristic—các "kinh nghiệm", "nguyên tắc định hướng" hoặc "những cái nên thử vì đôi khi hiệu quả"—chứ không phải là quy trình lặp lại có thể bảo đảm kết quả như nhau. Thiết kế gắn với thí nghiệm và thử - sai. Một công cụ hoặc kỹ thuật phù hợp cho dự án này chưa chắc phù hợp cho dự án khác.

Thiết kế là Quá trình Phát sinh (Emergent)

Cách mô tả súc tích nhất về các đặc điểm này là: thiết kế là một quá trình phát sinh (emergent). Thiết kế không nảy sinh đầy đủ ngay từ đầu; nó liên tục được phát triển và cải tiến thông qua các buổi đánh giá thiết kế, trao đổi không chính thức, kinh nghiệm lập trình và sửa đổi mã nguồn.

Hầu hết mọi hệ thống đều trải qua nhiều thay đổi thiết kế trong giai đoạn phát triển đầu tiên, và thường sẽ thay đổi nhiều hơn nữa khi được mở rộng ở các phiên bản về sau. Mức độ thay đổi là có lợi hay không phụ thuộc vào bản chất phần mềm được xây dựng.

5.2 Các Khái niệm Thiết kế Cốt lõi

> Thiết kế tốt dựa trên sự hiểu biết vững chắc về một số khái niệm trọng yếu. Phần này sẽ đề cập tới vai trò của độ phức tạp, các đặc tính mong muốn của thiết kế, cũng như các cấp độ thiết kế.

Đã kiểm tra và hiệu chỉnh thuật ngữ, cấu trúc và đảm bảo bản dịch chuẩn xác, học thuật, súc tích và nhất quán xuyên suốt đoạn trích.

Ngoài thiết kế, xem thêm

Mục 34.1, “Chinh phục Sự phức tạp”

Khó khăn ngẫu nhiên (Accidental Difficulties) và cơ bản (Essential Difficulties)

Brooks cho rằng phát triển phần mềm trở nên khó khăn bởi hai loại vấn đề khác nhau — khó khăn cơ bản (essential) và khó khăn ngẫu nhiên (accidental). Khi nhắc đến hai thuật ngữ này, Brooks dựa trên truyền thống triết học bắt nguồn từ Aristotle. Trong triết học, thuộc tính cơ bản là những thuộc tính mà một vật phải có để được xem là chính vật đó. Ví dụ, một chiếc ô tô phải có động cơ, bánh xe và cửa; nếu thiếu bất kỳ thuộc tính cơ bản nào, nó sẽ không thực sự là một chiếc ô tô.

Thuộc tính ngẫu nhiên là những thuộc tính mà một vật tình cờ có được, những thứ không thật sự quyết định bản chất của vật đó. Một chiếc ô tô có thể dùng động cơ V8, động cơ tăng áp 4 xy-lanh, hay loại động cơ khác và vẫn là ô tô. Xe có thể có hai cửa hoặc bốn; có bánh xe nhỏ hoặc lớn — tất cả đều là thuộc tính ngẫu nhiên. Có thể xem thuộc tính ngẫu nhiên là các yếu tố phụ, tùy chọn hoặc xuất hiện ngẫu nhiên.

> **Tham khảo chéo**: Khó khăn ngẫu nhiên nổi bật hơn ở giai đoạn đầu phát triển phần mềm so với các giai đoạn sau. Ví dụ, các khó khăn ngẫu nhiên liên quan đến cú pháp rườm rà đã được loại bỏ nhờ sự phát triển từ ngôn ngữ Assembly sang ngôn ngữ thế hệ ba (third-generation languages) và từ đó giảm dần tầm quan trọng. Khó khăn ngẫu nhiên do máy tính không có chế độ tương tác đã được giải quyết khi hệ điều hành chia sẻ thời gian (time-share operating systems) thay thế hệ thống xử lý theo lô (batch-mode systems). Các môi trường lập trình tích hợp (Integrated Programming Environments) tiếp tục loại bỏ sự kém hiệu quả khi các công cụ làm việc không đồng bộ với nhau. (Xem thêm Mục 4.3, “Vị trí của bạn trên Sóng Công nghệ”)

Brooks lập luận rằng tiến bộ trong việc xử lý các khó khăn cơ bản còn lại của phần mềm diễn ra chậm hơn. Nguyên nhân là vì thực chất phát triển phần mềm là quá trình xử lý các chi tiết của một tập hợp các khái niệm phức tạp, ràng buộc lẫn nhau. Khó khăn cơ bản xuất phát từ việc phải giao tiếp với thế giới thực vốn phức tạp và không trật tự; nhận diện đầy đủ và chính xác các quan hệ phụ thuộc, trường hợp ngoại lệ; thiết kế giải pháp không chỉ đúng tương đối mà phải chính xác tuyệt đối, v.v. Ngay cả khi chúng ta có thể phát minh ra một ngôn ngữ lập trình sử dụng chính thuật ngữ của bài toán thực tế, việc lập trình vẫn sẽ khó bởi bản chất việc xác định chính xác cách thức hoạt động của thế giới thực vốn đã rất phức tạp. Khi phần mềm giải quyết các vấn đề thực ngày càng lớn, tương tác giữa các thực thể thực tế cũng trở nên phức tạp hơn, từ đó làm tăng khó khăn cơ bản trong giải pháp phần mềm.

Gốc rễ của tất cả những khó khăn cơ bản này chính là sự phức tạp — cả ngẫu nhiên và cơ bản.

Tầm quan trọng của quản lý sự phức tạp

> Khi các khảo sát dự án phần mềm báo cáo nguyên nhân thất bại, hiếm khi các nguyên nhân kỹ thuật được nhắc đến là nguyên nhân chính. Dự án thường thất bại do yêu cầu không rõ ràng, kế hoạch kém hoặc quản lý yếu. Tuy nhiên, khi dự án thất bại do yếu tố kỹ thuật, nguyên nhân chủ yếu thường là do sự phức tạp không kiểm soát — phần mềm trở nên phức tạp đến mức không ai thực sự hiểu hết tác động của việc thay đổi mã ở một khu vực lên các khu vực khác, khiến tiến độ bị đình trệ.

Quản lý sự phức tạp chính là chủ đề kỹ thuật quan trọng nhất trong phát triển phần mềm. Theo quan điểm của tôi, quản lý sự phức tạp chính là Yếu tố kỹ thuật cốt lõi trong phát triển phần mềm.

Ý chính

Sự phức tạp không phải là một tính chất mới của phát triển phần mềm. Nhà tiên phong tính toán Edsger Dijkstra từng chỉ ra rằng lập trình viên là nghề duy nhất mà một trí óc đơn lẻ phải bao quát phạm vi từ một bit đến hàng trăm megabyte — tỉ lệ lên tới 1 trên 10^9, hay chênh lệch 9 bậc độ lớn (Dijkstra, 1989). Tỉ lệ khổng lồ này thật đáng kinh ngạc. Dijkstra diễn đạt như sau: “So với số lượng tầng ý nghĩa đó, lý thuyết toán học bình quân gần như phẳng. Việc tạo ra các tầng khái niệm sâu sắc đòi hỏi một thử thách trí tuệ hoàn toàn mới mà chưa từng có tiền lệ trong lịch sử.” Và ngày nay, phần mềm còn phức tạp hơn nhiều; tỉ lệ này có thể đã lên tới 1 trên 10^{15}.

> Một dấu hiệu cho thấy bạn đã bị ngập trong sự phức tạp là khi bạn kiên trì áp dụng một phương pháp hiển nhiên không liên quan, ít nhất là đối với người ngoài quan sát. “Nó giống như một người không rành kỹ thuật, khi xe

> hơi hỏng lại đổ nước vào ắc quy và đổ gạt tàn ra ngoài.”
> — *P. J. Plauger*

Ở cấp độ kiến trúc phần mềm, sự phức tạp của vấn đề được giảm bằng cách phân chia hệ thống thành các phân hệ con (subsystems). Con người dễ tiếp cận nhiều phần thông tin đơn giản hơn là một khối thông tin phức tạp. Mục tiêu của mọi kỹ thuật thiết kế phần mềm là chia nhỏ vấn đề phức tạp thành những phần đơn giản. Các phân hệ càng độc lập, bạn càng dễ tập trung giải quyết một phần nhỏ tại một thời điểm. Đối tượng (object) được định nghĩa cẩn thận sẽ phân tách các mối quan tâm, cho phép bạn tập trung từng vấn đề một. Các package phục vụ mục đích tương tự ở cấp độ trừu tượng cao hơn. Việc giữ các routine (thủ tục) ngắn giúp giảm tải công việc tinh thần. Việc lập trình dựa trên miền vấn đề thay vì các chi tiết cài đặt mức thấp, và làm việc ở lớp trừu tượng cao nhất, cũng giúp giảm tải cho trí óc.

Kết luận là, các lập trình viên biết chú ý đến giới hạn cố hữu của con người sẽ viết code dễ hiểu hơn cho cả bản thân và người khác, đồng thời giảm thiểu lỗi.

Làm thế nào để đối phó với sự phức tạp

Các thiết kế đắt đỏ và kém hiệu quả thường bắt nguồn từ ba nguồn gốc:

- Đưa ra giải pháp phức tạp cho một vấn đề đơn giản
- Đưa ra giải pháp đơn giản nhưng sai cho một vấn đề phức tạp
- Đưa ra giải pháp phức tạp, không thích hợp đối với vấn đề phức tạp

Như Dijkstra chỉ ra, phần mềm hiện đại vốn dĩ đã phức tạp; dù cố gắng đến mức nào, cuối cùng bạn vẫn phải đối mặt với một mức độ phức tạp gắn liền với bản chất của bài toán thực tế. Điều này gợi ý hai hướng tiếp cận trong quản lý sự phức tạp:

1. Giảm thiểu lượng phức tạp cơ bản mà bất kỳ ai phải xử lý tại một thời điểm.
2. Ngăn chặn sự gia tăng không cần thiết của phức tạp ngẫu nhiên.

Ý chính

Khi đã hiểu rằng mọi mục tiêu kỹ thuật khác đều xếp sau quản lý sự phức tạp, các cân nhắc về thiết kế sẽ trở nên rõ ràng hơn.

Các đặc trưng mong muốn của thiết kế

> Khi tôi đang làm việc với một vấn đề, tôi không nghĩ đến cái đẹp. Tôi chỉ nghĩ làm sao giải quyết được vấn đề. Nhưng khi xong, nếu giải pháp không đẹp, tôi biết nó sai.
> — *R. Buckminster Fuller*

Một thiết kế có chất lượng cao cần hội tụ một số đặc điểm chung. Nếu đạt được tất cả những mục tiêu này, thiết kế của bạn sẽ rất tốt. Tuy nhiên, một số mục tiêu có thể mâu thuẫn nhau; thách thức khi thiết kế là phải cân bằng giữa các mục tiêu cạnh tranh.

Một số đặc trưng của chất lượng thiết kế cũng là đặc trưng của chương trình tốt: độ tin cậy, hiệu năng, v.v. Những đặc trưng khác liên quan đến bản thân thiết kế.

> **Tham khảo chéo**: Các đặc điểm sau liên quan đến thuộc tính chất lượng phần mềm chung. Xem chi tiết tại Mục 20.1, “Các Đặc điểm của Chất lượng Phần mềm”.

Danh sách các đặc trưng thiết kế nội tại:

- **Độ phức tạp tối thiểu**: Mục tiêu hàng đầu khi thiết kế là giảm tối đa sự phức tạp, vì những lý do đã đề cập. Tránh thiết kế “thông minh” (clever) — các thiết kế quá thông minh thường khó hiểu. Thay vào đó, hãy hướng tới thiết kế đơn giản, dễ hiểu. Nếu thiết kế của bạn không cho phép bỏ qua phần lớn chương trình khi tập trung vào một bộ phận cụ thể, thì thiết kế đó chưa đáp ứng đúng vai trò.

- **Dễ bảo trì**: Thiết kế cần hướng tới người bảo trì (maintenance programmer). Thường xuyên tưởng tượng những câu hỏi mà người bảo trì sẽ đặt ra cho đoạn mã bạn viết. Hãy xem họ như đối tượng sử dụng, từ đó thiết kế hệ thống dễ hiểu, tự giải thích.

- **Kết ghép lỏng lẻo (Loose coupling)**: Các phần của chương trình nên hạn chế kết nối lẫn nhau ở mức tối thiểu. Áp dụng các nguyên tắc của abstraction (trừu tượng hóa) trong interface lớp, encapsulation (đóng gói) và information hiding (che giấu thông tin) để thiết kế các lớp với càng ít liên kết càng tốt. Liên kết tối thiểu sẽ giảm bớt khối lượng công việc khi tích hợp, kiểm thử và bảo trì.

- **Khả năng mở rộng (Extensibility)**: Hệ thống có thể được mở rộng mà không gây phá vỡ cấu trúc nền tảng. Việc thay đổi một phần trong hệ thống không ảnh hưởng đến các phần còn lại; những thay đổi thường gặp phải gây ít tác động

nhất.

- **Tái sử dụng (Reusability)**: Thiết kế hệ thống sao cho các thành phần có thể dùng lại ở hệ thống khác.
- **High fan-in**: Hiểu đơn giản là có nhiều lớp sử dụng một lớp cho trước.

(Phần bản dịch kết thúc tại đây, theo đúng giới hạn văn bản cung cấp. Nếu bạn cần dịch tiếp, vui lòng gửi phần còn lại.)

5.2 Các Khái Niệm Thiết Kế Chính

Fan-out Thấp đến Trung Bình

Fan-out thấp đến trung bình nghĩa là một lớp sử dụng một số lượng nhỏ đến vừa phải các lớp khác. **Fan-out cao** (thường là hơn bảy) cho thấy rằng một lớp sử dụng một số lượng lớn các lớp khác và do đó có thể trở nên quá phức tạp. Các nhà nghiên cứu đã phát hiện rằng nguyên tắc **fan-out thấp** mang lại lợi ích bất kể xét về số lượng routine (thủ tục) được gọi từ bên trong một routine hay bên trong một lớp (Card and Glass 1990; Basili, Briand, và Melo 1996).

Tính Di động (Portability)

Portability (Tính di động) là việc thiết kế hệ thống sao cho bạn có thể dễ dàng chuyển nó sang một môi trường khác.

Tính Gọn Nhẹ (Leanness)

Leanness (Tính gọn nhẹ) là thiết kế hệ thống sao cho nó không có các thành phần thừa (Wirth 1995, McConnell 1997). Voltaire từng nói rằng, một cuốn sách hoàn thiện không phải khi không thể thêm gì vào, mà là khi không thể loại bỏ gì thêm. Trong phần mềm, điều này đặc biệt đúng vì mã dư thừa cần được phát triển, rà soát, kiểm thử, và phải cân nhắc khi sửa đổi những phần mã khác. Các phiên bản tương lai của phần mềm cũng phải duy trì khả năng tương thích ngược với phần mã dư thừa này. Câu hỏi nguy hiểm là: "Việc này đơn giản, vậy bổ sung vào thì có hại gì không?"

Phân Tầng (Stratification)

Stratification (Phân tầng) nghĩa là cố gắng giữ các cấp độ phân rã được tách bạch, giúp bạn có thể quan sát hệ thống ở bất kỳ cấp độ nào mà vẫn có được góc nhìn nhất quán. Thiết kế hệ thống sao cho bạn có thể xem xét từng cấp độ mà không cần đi sâu vào các cấp khác.

> **Chú thích:** Ví dụ, khi bạn đang phát triển một hệ thống hiện đại phải sử dụng nhiều phần mã cũ được thiết kế kém, hãy xây dựng một lớp mới có trách nhiệm giao tiếp với mã cũ đó. Thiết kế lớp này sao cho nó che giấu những điểm yếu của mã cũ, cung cấp cho các lớp mới một bộ dịch vụ nhất quán. Sau đó, phần còn lại của hệ thống nên sử dụng các lớp này thay vì làm việc trực tiếp với mã cũ. Những hiệu quả tích cực của thiết kế phân tầng trong trường hợp này là: (1) cô lập được sự lộn xộn của mã chất lượng thấp và (2) nếu có cơ hội loại bỏ hoặc refactor (tái cấu trúc) mã cũ, bạn chỉ cần thay đổi lớp giao tiếp.

Kỹ Thuật Chuẩn (Standard Techniques)

Càng sử dụng nhiều thành phần đặc biệt, hệ thống càng gây khó khăn cho ai đó khi tiếp cận lần đầu. Hãy cố gắng đem lại cảm giác quen thuộc cho toàn hệ thống bằng cách sử dụng các phương pháp và kỹ thuật phổ biến, chuẩn hóa.

> **Chú thích:** Một hình thức chuẩn hóa đặc biệt hữu ích là sử dụng design pattern (mẫu thiết kế), đề cập tại phần "Look for Common Design Patterns" trong Mục 5.3.

---

82 Chương 5: Thiết Kế Trong Lập Trình

Các Cấp Độ Thiết Kế (Levels of Design)

Thiết kế phần mềm cần thực hiện ở nhiều mức độ chi tiết khác nhau. Một số kỹ thuật thiết kế áp dụng ở mọi cấp độ, trong khi một số khác chỉ phù hợp với một hoặc hai cấp độ nhất định. Hình 5-2 minh họa các cấp thiết kế chính:

1. Hệ thống phần mềm
2. Phân chia thành các subsystem/package (phân hệ/gói)
3. Phân chia thành các class (lớp) trong từng package
4. Phân chia data (dữ liệu) và routine (thủ tục) trong mỗi class
5. Thiết kế nội bộ routine

Figure 5-2 The levels of design in a program The system (1) is first organized into sub-systems (2) The subsystems are further divided into classes (3), and the classes are divided into routines and data (4) The inside of each routine is also designed (5)

Cấp 1: Hệ Thống Phần Mềm

Cấp độ đầu tiên là toàn bộ hệ thống. Một số lập trình viên thường nhảy thẳng từ thiết kế hệ thống sang thiết kế class, nhưng thường có lợi khi cân nhắc trước các kết hợp ở mức cao hơn như subsystem (phân hệ) hoặc package (gói).

Cấp 2: Phân Chia Thành Subsystem Hoặc Package

Sản phẩm chính của thiết kế ở cấp độ này là xác định tất cả các subsystem quan trọng. Các subsystem này có thể khá lớn như database (cơ sở dữ liệu), user interface (giao diện người dùng), business rules (quy tắc nghiệp vụ), command interpreter (bộ phiên dịch lệnh), report engine (bộ tạo báo cáo), v.v. Hoạt động thiết kế chủ yếu là quyết định cách phân chia chương trình thành các subsystem chính và xác định quy tắc sử dụng lẫn nhau giữa chúng. Phân chia ở cấp này thường cần thiết cho bất kỳ dự án nào kéo dài hơn vài tuần. Trong mỗi subsystem, có thể dùng các phương pháp thiết kế khác nhau để phù hợp nhất với từng phần.

> **Lưu ý:** Đặc biệt quan trọng ở cấp độ này là các quy tắc về giao tiếp giữa các subsystem. Nếu tất cả đều có thể trao đổi thoải mái, lợi ích phân tách sẽ mất đi. Hãy làm cho mỗi subsystem có ý nghĩa bằng cách giới hạn giao tiếp.

Giả sử bạn định nghĩa một hệ thống với sáu subsystem như minh họa dưới đây:

Figure 5-3 An example of a system with six subsystems
User Interface Graphics
Application
Data Storage
Level Classes
Business Enterprise-Level
Rules Tools

Nếu không có quy tắc cụ thể, **định luật thứ hai của nhiệt động lực học** sẽ vận hành và entropy của hệ thống sẽ gia tăng. Điều này có nghĩa là giao tiếp giữa các subsystem sẽ trở nên không kiểm soát, như sau:

Figure 5-4 An example of what happens with no restrictions on intersubsystem communications
User Interface Graphics
Application
Data Storage
Level Classes
Business Enterprise-Level
Rules Tools

Như có thể thấy, mỗi subsystem sẽ giao tiếp trực tiếp với mọi subsystem khác, dẫn đến những câu hỏi quan trọng sau:

- Một lập trình viên cần hiểu bao nhiêu phần khác nhau của hệ thống để thay đổi một chi tiết trong subsystem graphics (đồ họa)?
- Điều gì xảy ra nếu muốn tái sử dụng các business rules trong một hệ thống khác?
- Nếu muốn thay thế giao diện người dùng (ví dụ thử nghiệm bằng giao diện dòng lệnh), bạn sẽ làm thế nào?
- Nếu muốn chuyển data storage (lưu trữ dữ liệu) lên một máy chủ từ xa thì sao?

Bạn có thể hình dung các liên kết giữa các subsystem như ống dẫn nước. Nếu muốn rút một subsystem ra, thì sẽ phải tháo rời một số "ống" kết nối. Càng nhiều ống cần ngắt và nối lại, việc thay đổi càng phức tạp. Hãy thiết kế kiến trúc để khi cần di chuyển một subsystem ta chỉ cần thay đổi một vài liên kết rất rõ ràng, và dễ dàng nối lại.

Với sự cân nhắc trước, tất cả các vấn đề trên có thể được giải quyết mà không tốn nhiều công sức. Hãy chỉ cho phép giao tiếp giữa các subsystem khi "thật sự cần thiết" và phải có lý do hợp lý. Nếu còn phân vân, nên giới hạn giao tiếp trước; nếu cần mở rộng, việc nới lỏng quy tắc về sau sẽ dễ dàng hơn nhiều so với việc giảm bớt giao tiếp sau khi đã có hàng trăm lời gọi lẫn nhau giữa các subsystem. Hình dưới đây minh họa việc áp dụng một số nguyên tắc giao tiếp sẽ thay đổi hệ thống ra sao:

Figure 5-5 With a few communication rules, you can simplify subsystem interactions significantly
User Interface Graphics
Application
Data Storage
Level Classes
Business Enterprise-Level
Rules Tools

Nguyên Tắc Kết Nối

Cố gắng giữ cho các kết nối giữa các subsystem đơn giản và dễ bảo trì. Kiểu kết nối đơn giản nhất là một subsystem gọi routine của subsystem khác. Phức tạp hơn là khi một subsystem chứa class từ subsystem khác. Phức tạp nhất là các class của subsystem này kế thừa từ subsystem kia.

Nguyên tắc chung: sơ đồ hệ thống cấp cao như Hình 5-5 nên là một **đồ thị không có chu trình** (acyclic graph). Điều này có nghĩa là chương trình không nên có mối quan hệ vòng tròn, ví dụ Class A dùng Class B, Class B dùng Class C, và Class C lại dùng Class A.

Đối với các chương trình lớn, thiết kế ở cấp subsystem rất quan trọng. Nếu bạn nghĩ chương trình của mình đủ nhỏ để bỏ qua cấp này, thì ít nhất hãy ra quyết định này một cách có cân nhắc.

Các Subsystem Phổ Biến

Một số loại subsystem thường xuyên xuất hiện ở nhiều hệ thống khác nhau. Dưới đây là một số ví dụ tiêu biểu:

- **Business rules**: Các quy tắc nghiệp vụ là tập hợp các luật, quy định, chính sách và thủ tục được mã hoá vào hệ thống vi tính. Ví dụ, hệ thống tính lương có thể mã hoá các quy tắc của IRS (Sở Thuế vụ Hoa Kỳ) liên quan đến số lượng khấu trừ được phép, mức thuế dự kiến, các quy tắc từ hợp đồng công đoàn (về làm thêm giờ, nghỉ lễ...), hoặc các quy định tính phí bảo hiểm xe.
- **User interface**: Xây dựng riêng một subsystem cho giao diện người dùng để khi muốn phát triển hoặc thay đổi giao diện, không cần ảnh hưởng đến các phần khác của chương trình. Thông thường, subsystem giao diện người dùng sẽ quản lý nhiều subsystem hoặc class nhỏ khác như GUI (giao diện đồ hoạ), giao diện dòng lệnh, menu, quản lý cửa sổ, hệ thống trợ giúp, v.v.
- **Database access**: Tạo một lớp (hoặc subsystem) che giấu những chi tiết về thao tác với cơ sở dữ liệu, giúp phần lớn chương trình không phải xử lý trực tiếp các cấu trúc dữ liệu ở mức thấp mà chỉ cần làm việc với dữ liệu ở mức nghiệp vụ. Thiết kế này giúp giảm độ phức tạp tổng thể, tập trung điều khiển thao tác dữ liệu ở một nơi duy nhất, giảm nguy cơ lỗi sai, và dễ dàng thay đổi thiết kế database mà không ảnh hưởng đến các phần khác của chương trình.
- **System dependencies**: Gom các phụ thuộc vào hệ điều hành vào một subsystem, cũng giống như cách xử lý với phụ thuộc phần cứng, để kiểm soát tốt hơn sự phụ thuộc này trong toàn hệ thống.

Chú thích bổ sung:

- Đã bỏ qua hoặc chỉnh sửa các lỗi định dạng và diễn đạt để bảo đảm bản dịch rõ ràng, mạch lạc.
- Các thuật ngữ như class, routine, subsystem, package, design pattern giữ nguyên và được chú thích khi cần thiết theo quy ước dịch thuật khoa học máy tính.

Nếu sau này bạn muốn chuyển chương trình của mình sang Mac OS hoặc Linux, tất cả những gì bạn cần thay đổi chỉ là *interface subsystem* (hệ con giao tiếp). Một *interface subsystem* có thể quá lớn để bạn tự mình triển khai, nhưng các hệ con như vậy có sẵn rộng rãi trong nhiều thư viện mã nguồn thương mại.

Cấp độ 3: Phân chia thành các lớp (Classes)

Tham khảo thêm: Để tìm hiểu sâu hơn về thiết kế cơ sở dữ liệu ở mức này, xem *Agile Database Techniques* (Ambler 2003).

Thiết kế ở cấp độ này bao gồm việc xác định tất cả các *class* trong hệ thống. Ví dụ, một *database-interface subsystem* (hệ con giao tiếp cơ sở dữ liệu) có thể được phân chia tiếp thành các *data access class* (lớp truy xuất dữ liệu), *persistence framework class* (lớp khung bền vững) cũng như *database metadata* (siêu dữ liệu cơ sở dữ liệu). Hình 5-2, Cấp độ 3, minh họa cách một hệ con ở Cấp độ 2 có thể được chia nhỏ thành các lớp, đồng thời gợi ý rằng ba hệ con còn lại ở Cấp độ 2 cũng được phân rã thành các lớp.

Chi tiết về cách mỗi *class* tương tác với phần còn lại của hệ thống cũng được xác định song song với quá trình định nghĩa các *class*. Đặc biệt, *interface* (giao diện) của *class* sẽ được xác định.

Nhìn chung, hoạt động thiết kế chính ở cấp độ này là đảm bảo tất cả các *subsystem* (hệ con) đã được phân rã đủ mức chi tiết để bạn có thể hiện thực các phần của chúng như những *class* riêng biệt.

Tham khảo chéo: Để biết chi tiết về việc tạo các *class* chất lượng cao, xem Chương 6 “Working Classes”.

Việc chia các hệ con thành lớp (class) là điều cần thiết đối với bất kỳ dự án nào kéo dài hơn vài ngày. Nếu dự án lớn, sự phân chia này sẽ rất rõ ràng so với việc phân tách chương trình ở Cấp độ 2. Nếu dự án rất nhỏ, bạn có thể chuyển trực tiếp từ cái nhìn tổng thể ở Cấp độ 1 sang cái nhìn theo lớp ở Cấp độ 3.

Sự khác biệt giữa Class và Object

Một khái niệm then chốt trong thiết kế hướng đối tượng là sự phân biệt giữa *class* và *object*. Một *object* là bất kỳ thực thể cụ thể nào tồn tại trong chương trình tại thời điểm chạy. Một *class* là thành phần tĩnh bạn thấy trong danh sách chương trình. *Object* là thành phần động với các giá trị, thuộc tính cụ thể bạn thấy khi chạy chương trình.

Ví dụ, bạn có thể khai báo một *class* `Person` chứa các thuộc tính như `name`, `age`, `gender`, v.v. Khi chương trình chạy, bạn sẽ có các *object* như `nancy`, `hank`, `diane`, `tony`... – tức là các instance cụ thể của *class*. Nếu bạn đã quen với thuật ngữ trong cơ sở dữ liệu, thì nó giống với sự khác biệt giữa "schema" (lược đồ) và "instance" (thể hiện). Có thể ví *class* như khuôn bánh và *object* là chiếc bánh. Cuốn sách này sử dụng các thuật ngữ này một cách không hoàn toàn chính thức và thường đề cập đến *class* và *object* gần như thay thế cho nhau.

Cấp độ 4: Phân chia thành Routine

Thiết kế ở cấp này bao gồm việc phân chia mỗi *class* thành các *routine* (thủ tục/hàm). *Class interface* (giao diện lớp) được định nghĩa ở Cấp độ 3 sẽ xác định một số *routine*. Thiết kế ở Cấp độ 4 sẽ chi tiết hóa các *private routine* (thủ tục/hàm riêng) của lớp. Khi bạn kiểm tra các chi tiết của *routine* bên trong lớp, sẽ thấy nhiều *routine* đơn giản, nhưng có một số *routine* bao gồm các *routine* được tổ chức theo thứ bậc, đòi hỏi phải thiết kế sâu hơn.

Việc định nghĩa đầy đủ các *routine* của *class* thường dẫn đến hiểu rõ hơn về *class interface*, và do đó sinh ra các thay đổi tương ứng đối với giao diện – tức là các thay đổi ở Cấp độ 3.

Mức độ phân rã và thiết kế này thường được giao cho từng lập trình viên tự thực hiện, và cần thiết cho bất kỳ dự án nào kéo dài hơn vài giờ. Việc này không nhất thiết phải được thực hiện một cách chính thức, nhưng tối thiểu cần được hình dung trong đầu.

Cấp độ 5: Thiết kế Nội bộ Routine

Tham khảo chéo: Để biết chi tiết về tạo các *routine* chất lượng cao, tham khảo Chương 7, "High-Quality Routines," và Chương 8, "Defensive Programming."

Thiết kế ở mức *routine* bao gồm việc xác định chức năng chi tiết của từng *routine* riêng lẻ. Thiết kế nội bộ của *routine* thường được giao cho từng lập trình viên phụ trách *routine* đó. Quá trình này bao gồm các hoạt động như viết *pseudocode* (mã giả), tra cứu thuật toán trong sách tham khảo, quyết định cách tổ chức các đoạn mã trong *routine* và viết mã bằng ngôn ngữ lập trình. Ở cấp độ này, thiết kế luôn được thực hiện, dù đôi khi chỉ là vô thức và chất lượng kém, thay vì có chủ đích và chất lượng cao. Trong Hình 5-2, thiết kế ở cấp độ này được đánh dấu với số 5.

5.3 Các Khối Xây Dựng Thiết Kế: Heuristic

Các nhà phát triển phần mềm thường thích có các câu trả lời rõ ràng: "Làm A, B, và C, thì X, Y, Z sẽ luôn xảy ra." Chúng ta tự hào khi học được các bước thuần thục tạo ra kết quả mong muốn, và cảm thấy khó chịu khi các hướng dẫn không hoạt động như quảng cáo.

Nhu cầu về hành vi xác định như vậy là hoàn toàn phù hợp đối với việc lập trình chi tiết, bởi sự cẩn trọng đến từng chi tiết đó quyết định thành bại của chương trình. Tuy nhiên, thiết kế phần mềm là một câu chuyện hoàn toàn khác.

Vì thiết kế là không định hướng (nondeterministic), việc áp dụng thành thạo một tập hợp hiệu quả các *heuristic* (quy tắc thực nghiệm/hữu ích) là hoạt động cốt lõi trong thiết kế phần mềm tốt. Các phần sau đây mô tả một số *heuristic*—cách tiếp cận để xem xét một thiết kế, đôi khi sẽ đem lại những ý tưởng thiết kế tốt. Bạn có thể xem *heuristic* như là kim chỉ nam cho quá trình "thử và sai".

Chắc chắn bạn đã từng bắt gặp một số *heuristic* này trước đây. Vì vậy, các mục tiếp theo sẽ mô tả từng *heuristic* dựa trên Mệnh lệnh Kỹ thuật Chủ đạo của Phần mềm: *quản lý độ phức tạp*.

Tìm các Đối tượng Thực tế

> Đừng hỏi trước hệ thống làm gì; hãy hỏi NÓ làm việc đó cho ai!
> —Bertrand Meyer

Phương pháp phổ biến nhất để xác định các phương án thiết kế là cách tiếp cận *object-oriented* (hướng đối tượng) "theo sách giáo khoa", tập trung vào việc nhận diện các *real-world object* (đối tượng thực tế) và *synthetic object* (đối tượng tổng hợp).

Các bước thiết kế với đối tượng bao gồm:

- **Nhận diện các object và thuộc tính của chúng** (method và data).
- **Xác định các thao tác có thể thực hiện trên mỗi object**.
- **Xác định những gì mỗi object được phép làm đối với các object khác**.
- **Xác định bộ phận nào của mỗi object sẽ được nhìn thấy bởi các object khác—phần nào là *public*, phần nào là *private*.
- **Định nghĩa *public interface* (giao diện công khai) của mỗi object**.

Các bước này không nhất thiết phải thực hiện theo thứ tự, và thường được lặp đi lặp lại. Việc lặp lại là quan trọng. Dưới đây là tóm tắt từng bước.

Nhận diện các object và thuộc tính của chúng

Chương trình máy tính thường dựa trên các thực thể ngoài đời thực. Ví dụ, bạn có thể xây dựng hệ thống quản lý thời gian và hóa đơn dựa trên các nhân viên, khách hàng, thẻ chấm công và hóa đơn thực tế. Hình 5-6 minh họa góc nhìn hướng đối tượng về hệ thống hóa đơn như vậy.

```
Client
Employee
name
name billingAddress
title accountBalance
billingRate currentBillingAmount
GetHoursForMonth() EnterPayment()

1 billingEmployee 1 1 clientToBill
clientToBill
bills
* * *
Timecard Bill
hours billDate
date * 0 1
BillForClient()
projectCode billingRecords
```

Hình 5-6: Hệ thống hóa đơn này được cấu thành từ bốn đối tượng chính. Các đối tượng này đã được đơn giản hóa cho ví dụ này.

Việc nhận diện thuộc tính của các object cũng đơn giản như việc nhận diện chính các object. Mỗi object có những đặc trưng liên quan đến chương trình máy tính. Ví dụ, trong hệ thống chấm công, một object `employee` có thuộc tính `name`, `title`, `billingRate`. Một object `client` có thuộc tính `name`, `billingAddress`, `accountBalance`. Một object `bill` có `billingAmount`, `clientName`, `billingDate` và các thuộc tính khác.

Trong một hệ thống giao diện người dùng đồ họa, các object thường bao gồm `window`, `dialog box`, `button`, `font`, và `drawing tool`. Việc khảo sát sâu hơn miền vấn đề có thể đưa ra lựa chọn tốt hơn cho các object phần mềm so với việc ánh xạ một-một với các đối tượng thực tế, nhưng bắt đầu từ các object thực tế là hợp lý.

Xác định các thao tác có thể thực hiện trên mỗi object

Có nhiều thao tác có thể thực hiện trên mỗi object. Trong hệ thống hóa đơn minh họa ở Hình 5-6, một object `employee` có thể thay đổi `title` hoặc `billing rate`; một object `client` có thể thay đổi `name` hoặc `billingAddress`, v.v.

Xác định quyền tương tác giữa các object

Bước này đúng như tên gọi. Hai hình thức tổ chức tổng quát mà các object có thể thực hiện với nhau là *containment* (bao hàm) và *inheritance* (kế thừa). Object nào có thể chứa object nào? Object nào có thể kế thừa từ object nào? Trong Hình 5-6, một object `timecard` có thể chứa một `employee` và một `client`, và một `bill` có thể chứa một hoặc nhiều `timecard`. Ngoài ra, một `bill` có thể xác định một `client` đã được tính phí, và một `client` có thể thanh toán cho một hóa đơn. Hệ thống phức tạp hơn sẽ bao gồm nhiều mối quan hệ tương tác hơn nữa.

Xác định các phần của object được hiển thị ra bên ngoài

Một trong những quyết định thiết kế then chốt là nhận diện những phần nào của một object nên *public* và phần nào nên *private*. Quyết định này cần được đưa ra đối với cả *data* lẫn *method*.

Định nghĩa interface của từng object

Định nghĩa các *interface* chính thức, theo cú pháp ngôn ngữ lập trình, cho mỗi object. Dữ liệu và phương thức mỗi object cung cấp cho các object khác được gọi là *public interface* (giao diện công khai). Các phần của object cung cấp cho object dẫn xuất thông qua *inheritance* được gọi là *protected interface* (giao diện bảo vệ). Hãy xem xét cả hai loại interface này.

Sau khi hoàn thành các bước để đạt tới tổ chức hệ thống hướng đối tượng ở mức cao nhất, bạn sẽ phải lặp lại theo hai hướng: cải thiện tổ chức hệ thống tổng thể (các *class*), đồng thời lặp lại việc phân tích từng *class* đã xác định, nhằm phát triển thiết kế chi tiết hơn nữa cho từng *class*.

Hình thành các Abstraction nhất quán

Abstraction (trừu tượng hóa) là khả năng làm việc với một khái niệm đồng thời bỏ qua an toàn một số chi tiết của nó — xử lý những chi tiết khác nhau ở các cấp độ khác nhau. Bất cứ khi nào bạn làm việc với một tập hợp, bạn đang làm việc với một *abstraction*. Nếu bạn gọi một vật là "nhà" thay vì mô tả chi tiết nó là sự kết hợp của kính, gỗ, đinh..., tức là bạn đã sử dụng abstraction.

Lớp Cơ Sở và Khái Niệm Trừu Tượng

Lớp cơ sở (base class) là các trừu tượng cho phép bạn tập trung vào các thuộc tính chung của một tập hợp các lớp dẫn xuất (derived class) và bỏ qua các chi tiết cụ thể của từng lớp khi bạn thao tác trên lớp cơ sở. Một interface (giao diện) lớp tốt là một trừu tượng giúp bạn chỉ cần quan tâm đến interface mà không cần bận tâm đến cách thức hoạt động bên trong của lớp đó. Interface của một routine (thủ tục/chương trình con) được thiết kế tốt cũng mang lại lợi ích tương tự ở mức chi tiết thấp hơn, và interface của một package (gói)/subsystem (hệ thống con) mang lại lợi ích này ở mức chi tiết cao hơn.

Từ góc độ độ phức tạp, lợi ích chính của trừu tượng hóa (abstraction) là cho phép bạn bỏ qua các chi tiết không liên quan. Hầu hết các đối tượng trong thế giới thực đều là các trừu tượng ở một mức độ nào đó. Như đã đề cập, một ngôi nhà là trừu tượng của cửa sổ, cửa ra vào (door), tường, hệ thống dây điện (wiring), ống nước (plumbing), lớp cách nhiệt (insulation) và cách chúng được tổ chức. Cánh cửa lại là trừu tượng của một sự sắp xếp vật liệu hình chữ nhật với bản lề (hinge) và tay nắm cửa (doorknob). Bản thân tay nắm cửa là trừu tượng của việc tạo hình từ đồng thau (brass), niken (nickel), sắt (iron) hoặc thép (steel).

Con người sử dụng trừu tượng hóa liên tục. Nếu mỗi lần sử dụng cửa bạn phải quan tâm tới từng sợi gỗ, từng phân tử vec-ni (varnish molecule), từng phân tử thép, chắc hẳn bạn khó có thể ra/vào nhà mỗi ngày. Như thể hiện ở Hình 5-7, trừu tượng hóa giúp chúng ta nhìn nhận các khái niệm phức tạp một cách đơn giản hơn.

Hình 5-7: Trừu tượng hóa cho phép bạn có cái nhìn đơn giản hơn về một khái niệm phức tạp.

Lạm Dụng Chi Tiết Kỹ Thuật Khi Thiết Kế Phần Mềm

Đôi khi các nhà phát triển phần mềm lại xây dựng hệ thống ở mức quá chi tiết (mức sợi gỗ, phân tử vec-ni, phân tử thép). Điều này khiến hệ thống trở nên phức tạp không cần thiết và khó quản lý về mặt trí tuệ. Khi lập trình viên không cung cấp các trừu tượng lập trình lớn hơn, đôi khi chính bản thân hệ thống cũng gặp khó khăn trong việc "vào cửa".

Các lập trình viên giỏi xây dựng các trừu tượng ở cấp interface routine, interface class và interface package — tương tự như tay nắm cửa, cánh cửa, và ngôi nhà — hỗ trợ lập trình nhanh hơn, an toàn hơn.

Đóng Gói Chi Tiết Triển Khai (Encapsulation)

Đóng gói (encapsulation) phát triển tiếp từ trừu tượng hóa. Trừu tượng hóa cho phép bạn nhìn vào một đối tượng ở mức chi tiết tổng quát; đóng gói bổ sung rằng, bạn KHÔNG được phép nhìn vào đối tượng ở bất kỳ mức chi tiết nào khác.

Tiếp nối với ví dụ về vật liệu xây nhà: đóng gói ngụ ý rằng bạn chỉ được phép nhìn bao quát bên ngoài ngôi nhà mà không thể nhìn gần đến mức nhận biết chi tiết từng cửa. Bạn được biết có một cái cửa, biết nó mở hay đóng, nhưng không được biết cửa làm từ gỗ, sợi thủy tinh, thép, hay vật liệu khác, và càng không được nhìn từng sợi gỗ riêng lẻ.

Hình 5-8: Đóng gói giúp quản lý độ phức tạp một cách hiệu quả bằng việc cấm bạn nhìn vào các chi tiết không liên quan.

"Những gì bạn nhìn thấy là tất cả những gì bạn có được!"

Thừa Kế (Inheritance) — Khi Thừa Kế Đơn Giản Hóa Thiết Kế

Khi thiết kế hệ thống phần mềm, bạn thường gặp các đối tượng rất giống nhau, chỉ khác vài điểm nhỏ. Ví dụ, trong hệ thống kế toán, có thể có cả nhân viên toàn thời gian và bán thời gian (employee). Hầu hết dữ liệu & thuộc tính của hai loại này giống nhau, chỉ có một số điểm khác biệt. Trong lập trình hướng đối tượng (OOP), bạn có thể định nghĩa một kiểu nhân viên tổng quát (general employee), sau đó định nghĩa nhân viên toàn thời gian và bán thời gian là dẫn xuất của nhân viên tổng quát, chỉ khác biệt ở một số điểm.

Khi thao tác với một employee mà không quan tâm kiểu cụ thể, các thao tác sẽ xử lý như thể chỉ là employee tổng quát. Nếu thao tác phụ thuộc bạn là nhân viên full-time hay part-time, chương trình sẽ xử lý khác biệt.

Việc xác định điểm giống và khác nhau giữa các đối tượng như vậy gọi là *thừa kế* (inheritance), vì các loại employee cụ thể sẽ thừa kế đặc điểm từ employee tổng quát.

Lợi ích của thừa kế là hỗ trợ tốt cho trừu tượng hóa. Trừu tượng hóa cho phép xử lý đối tượng ở các mức độ chi tiết khác nhau — như cánh cửa là tập hợp của phân tử ở một mức, sợi gỗ ở mức khác, và một công cụ chống trộm ở một mức cao hơn. Gỗ có những đặc tính riêng — như có thể cắt bằng cưa, dán bằng keo gỗ — và các loại gỗ khác nhau sẽ vừa có đặc tính chung vừa có đặc tính riêng.

Thừa kế đơn giản hóa lập trình vì bạn chỉ cần viết một routine tổng quát cho các thuộc tính chung của cửa, và các routine cụ thể cho các loại cửa khác nhau. Một số thao tác, như `Open()` hoặc `Close()`, có thể áp dụng cho bất kỳ loại cửa nào (cửa đặc, cửa trong nhà, cửa ngoài nhà, cửa lưới, cửa kính trượt...) mà không cần biết loại cửa là gì cho tới lúc chạy chương trình (runtime). Khả năng ngôn ngữ hỗ trợ các thao tác như vậy gọi là *đa hình* (polymorphism). Các ngôn ngữ hướng đối tượng như C++, Java và các phiên bản sau của Microsoft Visual Basic đều hỗ trợ kế thừa (inheritance) và đa hình (polymorphism).

> **Lưu ý:** Thừa kế là một trong những công cụ mạnh mẽ nhất của lập trình hướng đối tượng, nhưng cũng có thể gây nhiều tác hại nếu lạm dụng hoặc dùng không đúng cách.

Che Giấu Thông Tin (Information Hiding)

Che giấu thông tin (information hiding) là nền tảng của cả thiết kế cấu trúc (structured design) và thiết kế hướng đối tượng (object-oriented design). Trong thiết kế cấu trúc, khái niệm "hộp đen" (black box) xuất phát từ information hiding. Trong thiết kế hướng đối tượng, nó sinh ra các khái niệm đóng gói (encapsulation), tính mô-đun (modularity), cũng như gắn liền với trừu tượng hóa.

Information hiding là một trong những ý tưởng nền tảng của phát triển phần mềm. Khái niệm này lần đầu tiên được David Parnas công bố trong bài báo “On the Criteria to Be Used in Decomposing Systems Into Modules” (1972). Information hiding được đặc trưng bằng ý tưởng về “bí mật” — những quyết định về thiết kế và triển khai mà lập trình viên giấu trong một nơi duy nhất đối với phần còn lại của hệ thống.

Fred Brooks, trong ấn bản kỷ niệm 20 năm của *The Mythical Man Month*, đã kết luận rằng việc ông từng phê phán information hiding là sai. "Parnas đã đúng, tôi sai về information hiding," ông khẳng định (Brooks 1995). Barry Boehm báo cáo rằng information hiding là kỹ thuật mạnh mẽ trong việc loại trừ việc phải làm lại sản phẩm (eliminating rework), đặc biệt hiệu quả trong môi trường thay đổi liên tục (incremental, high-change environments) (Boehm 1987).

Information hiding là một *heuristic* mạnh mẽ đối với Yếu tố Kỹ thuật Chính của Phần Mềm (Software’s Primary Technical Imperative), vì từ tên gọi đến mọi chi tiết, nó đều nhấn mạnh việc ẩn giấu sự phức tạp.

Bí mật và Quyền Riêng Tư

Trong information hiding, mỗi class (hoặc package, routine) đặc trưng bởi các quyết định thiết kế/thi công mà nó che giấu với các class khác. Bí mật này có thể là khu vực hay thay đổi, định dạng một tệp, cách triển khai kiểu dữ liệu, hoặc vùng cần được cách ly nhằm hạn chế lỗi lan rộng. Nhiệm vụ của class là giữ kín thông tin này, bảo vệ quyền riêng tư của chính nó. Những thay đổi nhỏ có thể ảnh hưởng đến một số routine trong cùng một class, nhưng không nên ảnh hưởng ra ngoài interface của class.

> **Châm ngôn:** “Hãy hướng tới interface class đầy đủ nhưng tối giản.” — Scott Meyers

Một nhiệm vụ then chốt khi thiết kế class là quyết định tính năng nào nên công khai ra bên ngoài (public) và tính năng nào nên giữ kín (private). Một class có thể dùng 25 routine nhưng chỉ cung cấp ra ngoài 5 routine, dùng một số kiểu dữ liệu nhưng không hề để lộ cách triển khai chúng. Khía cạnh này còn gọi là “visibility” (tính hiển thị), tức là đặc điểm nào của class “hiển thị” ra ngoài.

Interface của class nên để lộ càng ít thông tin về cấu trúc nội bộ càng tốt. Giống như tảng băng: chỉ 1/8 nổi trên mặt nước, phần còn lại ẩn bên dưới.

> **Hình 5-9:** Interface của một class tốt như phần nổi của tảng băng, phần lớn chi tiết bên trong sẽ được che giấu.

Việc thiết kế interface class là một quá trình lặp lại, tương tự các khía cạnh thiết kế khác. Nếu lần đầu chưa được, hãy thử lại cho tới khi giao diện ổn định. Nếu nó không ổn định, bạn cần cân nhắc cách tiếp cận khác.

Ví Dụ về Che Giấu Thông Tin

Giả sử bạn có một chương trình trong đó mỗi object đều phải có một ID duy nhất, lưu trong biến thành viên `id`. Một cách thiết kế là sử dụng số nguyên cho các giá trị ID và lưu giá trị ID cao nhất hiện tại trong biến toàn cục `g_maxId`. Khi khởi tạo object mới, có thể trong constructor, bạn chỉ việc dùng lệnh:

```
id = ++g_maxId;
```

Điều này đảm bảo ID duy nhất và giảm thiểu mã ở mỗi nơi object được tạo ra.

Thách Thức

> Điều gì có thể xảy ra với giải pháp này?
>
> Rất nhiều điều có thể xảy ra không như ý muốn...

(Phần tiếp theo sẽ tiếp tục phân tích các rủi ro với phương pháp này).

Và nếu chương trình của bạn có tính đa luồng, phương pháp này sẽ không đảm bảo an toàn luồng (thread-safe)

Cách tạo ra các ID mới là một quyết định thiết kế mà bạn nên che giấu. Nếu bạn sử dụng biểu thức `++g_maxId` xuyên suốt chương trình, bạn đang để lộ cách mà một ID mới được tạo ra, đó chỉ đơn giản là tăng giá trị của `g_maxId`. Thay vào đó, nếu bạn sử dụng lệnh `id = NewId()` trong toàn bộ chương trình, bạn đã ẩn đi thông tin về cách tạo các ID mới. Bên trong thủ tục `NewId()`, bạn vẫn có thể chỉ có một dòng mã, chẳng hạn như:

```
return (++g_maxId);
```

hoặc tương đương, nhưng nếu sau này bạn quyết định dành riêng một số dải ID cho các mục đích đặc biệt hoặc tái sử dụng các ID cũ, bạn chỉ cần thay đổi trong thủ tục `NewId()`, mà không cần tác động tới hàng chục hoặc hàng trăm lệnh `id = NewId()` khác trong toàn bộ chương trình. Dù các thay đổi trong `NewId()` có phức tạp đến đâu, chúng cũng sẽ không ảnh hưởng tới các phần khác của chương trình.

Giả sử bạn phát hiện ra cần thay đổi kiểu của ID từ số nguyên (integer) sang chuỗi (string). Nếu bạn khai báo các biến như `int id` rải rác trong toàn bộ chương trình, việc sử dụng thủ tục `NewId()` sẽ không còn hữu ích. Bạn vẫn phải sửa đổi hàng chục, thậm chí hàng trăm chỗ trong mã nguồn.

Một bí mật bổ sung cần được che giấu đó là kiểu dữ liệu của ID. Việc để lộ ID là số nguyên khuyến khích lập trình viên thực hiện các phép toán số nguyên như >, <, = trên nó. Trong C++, bạn có thể sử dụng một khai báo `typedef` đơn giản để định nghĩa ID thuộc kiểu `IdType` — một kiểu do người dùng định nghĩa, thực chất ánh xạ tới `int` — thay vì khai báo trực tiếp kiểu `int`. Ngoài ra, trong C++ và các ngôn ngữ khác, bạn cũng có thể tạo một lớp `IdType` đơn giản.

Một lần nữa, việc ẩn giấu quyết định thiết kế có thể tạo ra sự khác biệt lớn về phạm vi mã nguồn bị ảnh hưởng khi có nhu cầu thay đổi.

Tính hữu ích của Information hiding (ẩn giấu thông tin)

Information hiding có ích ở mọi cấp độ thiết kế, từ việc sử dụng các hằng số có tên thay cho các giá trị cố định (literal), cho tới việc định nghĩa kiểu dữ liệu, thiết kế lớp (class design), thiết kế thủ tục (routine design) và thiết kế các miền con (subsystem design).

ĐIỂM MẤU CHỐT

Hai loại bí mật trong information hiding

Các bí mật cần ẩn dấu trong information hiding thường thuộc hai nhóm chính:

- **Ẩn giấu độ phức tạp:** để bạn không phải xử lý mọi chi tiết phức tạp trừ khi thực sự quan tâm tới nó;
- **Ẩn giấu nguồn thay đổi:** để khi xảy ra thay đổi, phạm vi ảnh hưởng được giới hạn.

Nguồn gốc của độ phức tạp bao gồm các kiểu dữ liệu phức tạp, cấu trúc tệp, kiểm tra boolean, các thuật toán phức tạp, v.v. Một danh sách đầy đủ hơn về nguồn thay đổi sẽ được trình bày trong các phần sau của chương.

Rào cản đối với Information hiding

> **Chú thích:** Một số nội dung trong phần này được trích dẫn và điều chỉnh từ "Designing Software for Ease of Extension and Contraction" (Parnas, 1979).

Trong một số trường hợp, information hiding thực sự không thực hiện được, nhưng phần lớn các rào cản là do thói quen tư duy đã hình thành từ việc sử dụng các kỹ thuật khác.

Phân tán thông tin quá mức

Một rào cản phổ biến là việc thông tin bị phân tán quá nhiều trong hệ thống. Ví dụ, bạn có thể đã "hard-code" giá trị 100 nhiều chỗ trong hệ thống. Việc sử dụng giá trị cố định như vậy khiến các tham chiếu trở nên phân tán. Tốt hơn hết, bạn nên ẩn

thông tin này vào một chỗ duy nhất, ví dụ hằng số `MAX_EMPLOYEES`, để khi cần thay đổi chỉ phải sửa ở một nơi.

Một ví dụ khác của việc phân tán thông tin là khi giao tiếp với người dùng được sen kẽ khắp hệ thống. Nếu sau này phương thức tương tác thay đổi, ví dụ chuyển từ giao diện GUI (Graphical User Interface) sang giao diện dòng lệnh, hầu như toàn bộ mã nguồn phải thay đổi. Sẽ hợp lý hơn khi gom toàn bộ giao tiếp người dùng vào một lớp, gói (package) hoặc miền con nhất định; khi đó, bạn chỉ cần sửa ở một nơi mà không ảnh hưởng tới phần còn lại của hệ thống.

Truy cập dữ liệu toàn cục

Một ví dụ khác là phần tử dữ liệu toàn cục, chẳng hạn một mảng nhân viên tối đa 1000 phần tử được truy cập ở nhiều nơi trong chương trình. Nếu chương trình truy cập dữ liệu toàn cục này trực tiếp, thông tin về việc dữ liệu đó là mảng và có tối đa 1000 phần tử sẽ bị rải rác khắp chương trình. Nếu bạn giới hạn việc truy cập dữ liệu này thông qua các thủ tục truy cập (access routines), thì chỉ các thủ tục truy cập mới biết chi tiết về cách cài đặt dữ liệu.

Phụ thuộc vòng tròn

Một rào cản tinh vi hơn đối với information hiding là sự phụ thuộc vòng tròn, ví dụ khi một thủ tục trong lớp A gọi một thủ tục trong lớp B, và ngược lại một thủ tục trong lớp B cũng gọi lại thủ tục trong lớp A.

Nên tránh các vòng lặp phụ thuộc như vậy. Chúng khiến việc kiểm thử hệ thống trở nên khó khăn, vì bạn không thể kiểm thử lớp A hoặc B cho đến khi ít nhất một phần của lớp còn lại đã sẵn sàng.

Nhầm lẫn giữa dữ liệu lớp và dữ liệu toàn cục

Nếu bạn là một lập trình viên cẩn trọng, có thể bạn muốn tránh biến toàn cục do các vấn đề phát sinh, và do đó cũng tránh cả dữ liệu lớp. Tuy nhiên, dữ liệu lớp ít rủi ro hơn dữ liệu toàn cục nhiều.

Dữ liệu toàn cục thường gặp hai vấn đề: các thủ tục thao tác lên nó mà không biết các thủ tục khác cũng đang thao tác, hoặc biết rằng có các thủ tục khác thao tác nhưng không biết chính xác chúng làm gì. Dữ liệu lớp không gặp hai vấn đề này, vì quyền truy cập dữ liệu được giới hạn cho một nhóm nhỏ các thủ tục trong một lớp duy nhất; và các thủ tục này đều biết các thủ tục khác cùng thao tác lên dữ liệu đó là gì.

> Lưu ý: Lập luận này giả định hệ thống của bạn được thiết kế với các lớp nhỏ, hợp lý. Nếu chương trình dùng các lớp quá lớn với hàng chục thủ tục, ranh giới giữa dữ liệu lớp và dữ liệu toàn cục sẽ bị lu mờ, dẫn tới các rủi ro tương tự như khi dùng dữ liệu toàn cục.

Lo ngại về hiệu năng (performance) khi ẩn thông tin

Một rào cản cuối cùng là lo ngại về việc giảm hiệu năng cả ở mức kiến trúc lẫn mã nguồn khi áp dụng information hiding.

- Ở mức kiến trúc, lo lắng về hiệu năng là không cần thiết vì việc thiết kế hệ thống theo hướng che giấu thông tin không hề mâu thuẫn với việc thiết kế hướng hiệu năng. Nếu bạn cân nhắc cả hai mục tiêu, bạn hoàn toàn có thể đạt được cả hai.
- Ở mức mã nguồn, nhiều người lo rằng việc truy cập dữ liệu gián tiếp qua nhiều đối tượng, gọi thủ tục... sẽ làm giảm hiệu năng thực thi. Tuy nhiên, lo lắng này là không cần thiết ở giai đoạn thiết kế/nguyên mẫu. Hãy tập trung xây dựng một thiết kế hướng mô-đun (modular) tốt. Khi đã có sản phẩm đo được hiệu năng và xác định được các điểm nghẽn (bottleneck), bạn hoàn toàn có thể tối ưu hóa các lớp hoặc thủ tục riêng biệt mà không ảnh hưởng đến toàn hệ thống.

Giá trị của Information hiding

Information hiding là một trong số ít các kỹ thuật lý thuyết đã chứng minh được giá trị thực tiễn một cách rõ ràng trong nhiều năm qua (Boehm, 1987a). Các chương trình lớn sử dụng information hiding được phát hiện là dễ sửa đổi hơn gấp bốn lần so với các chương trình không sử dụng phương pháp này (Korson và Vaishnavi, 1986). Hơn nữa, information hiding còn là nền tảng của cả thiết kế cấu trúc (structured design) và thiết kế hướng đối tượng (object-oriented design).

Information hiding có sức mạnh truyền cảm hứng thiết kế hiệu quả một cách đặc biệt. Thiết kế hướng đối tượng truyền cảm hứng bằng cách mô hình hóa thế giới thực thành các đối tượng, tuy nhiên nó không giúp bạn tự động tránh việc khai báo ID là kiểu int thay vì IdType. Một nhà thiết kế hướng đối tượng thường sẽ tự hỏi: "Có nên coi ID là một đối tượng?" Dựa trên các chuẩn mã hóa của dự án, câu trả lời "Có" có thể đồng nghĩa với việc lập trình viên phải viết constructor, destructor, operator copy, operator gán, thêm chú thích và chịu sự kiểm soát cấu hình. Hầu hết lập trình viên khi đó sẽ nghĩ: “Không, không đáng để tạo cả một lớp chỉ cho một ID, tôi sẽ dùng luôn kiểu int.”

Lưu ý điều gì đã xảy ra: Một phương án thiết kế hữu ích, đơn giản là che giấu kiểu dữ liệu của ID, đã không được cân nhắc. Nếu thay vào đó, người thiết kế tự hỏi "Điều gì về ID cần được che giấu?" thì rất có thể đã quyết định che giấu kiểu dữ liệu phía sau một khai báo đơn giản, thay thế `int` bằng `IdType`.

Sự khác biệt giữa thiết kế hướng đối tượng và information hiding trong ví dụ này không phải là sự đối lập về quy tắc mà là sự khác biệt về cách tư duy – việc nghĩ tới information hiding sẽ truyền cảm hứng và ủng hộ các quyết định thiết kế mà tư duy hướng đối tượng không bao quát hết.

Information hiding cũng hữu ích khi thiết kế giao diện public của một lớp. Khoảng cách giữa lý thuyết và thực tiễn trong thiết kế lớp là rất lớn, và với nhiều lập trình viên, quyết định về những gì nên xuất hiện trong public interface của lớp phần lớn chỉ dựa vào mức độ tiện lợi, và thường dẫn đến việc để lộ quá nhiều thông tin của lớp. Nhiều lập trình viên sẵn sàng để lộ tất cả dữ liệu private của lớp thay vì viết thêm vài dòng mã để bảo vệ bí mật của lớp.

Câu hỏi "Lớp này cần phải che giấu điều gì?" là cốt lõi của vấn đề thiết kế giao diện lớp. Nếu bạn có thể đưa một hàm hoặc dữ liệu vào public interface mà không làm lộ các bí mật, hãy làm vậy; nếu không, đừng.

Việc tự hỏi điều gì cần được che giấu sẽ ủng hộ những quyết định thiết kế tốt ở mọi cấp độ. Nó khuyến khích sử dụng các hằng số đặt tên thay cho các giá trị cố định ở mức mã nguồn; giúp đặt tên tốt cho các hàm và tham số trong lớp; hỗ trợ quyết định về cách phân rã, kết nối các lớp và miền con ở cấp độ hệ thống.

Đã kiểm tra lại các thuật ngữ chuyên ngành, đảm bảo tính nhất quán trong toàn văn bản.

Ý Chính

Xác Định Những Khu Vực Có Khả Năng Thay Đổi

> **Đọc Thêm**: Một nghiên cứu về các nhà thiết kế xuất sắc cho thấy một thuộc tính chung mà họ có là khả năng dự đoán trước các thay đổi (Glass, 1995). Việc thích ứng với các thay đổi là một trong những khía cạnh thách thức nhất của thiết kế chương trình tốt. Mục tiêu là cô lập các khu vực không ổn định để tác động của một thay đổi sẽ chỉ giới hạn trong một routine (thuật toán), class hoặc package (gói phần mềm).

Phương pháp được mô tả trong phần này được điều chỉnh từ "Designing Software for Ease of Extension and Contraction" (Parnas, 1979). Dưới đây là các bước bạn nên thực hiện để chuẩn bị cho những biến động như vậy:

1. **Xác định các thành phần có khả năng thay đổi**. Nếu tài liệu yêu cầu (requirements) được xây dựng tốt, chúng sẽ liệt kê các thay đổi tiềm ẩn và khả năng xảy ra của từng thay đổi. Trong trường hợp như vậy, việc xác định các thay đổi tiềm năng trở nên đơn giản. Nếu tài liệu không đề cập tới khả năng thay đổi, hãy xem thảo luận bên dưới về các khu vực thường xuyên thay đổi trong bất kỳ dự án nào.
2. **Tách riêng các thành phần dễ thay đổi**. Cô lập từng thành phần dễ biến động đã xác định ở bước 1 vào class riêng, hoặc gộp vào class với những thành phần khác cũng dễ thay đổi cùng lúc.
3. **Cô lập các thành phần dễ xảy ra thay đổi**. Thiết kế các giao diện giữa các class sao cho không bị phụ thuộc vào các thay đổi tiềm năng. Thiết kế giao diện sao cho các thay đổi bị giới hạn bên trong class, còn bên ngoài vẫn không bị ảnh hưởng. Bất kỳ class nào sử dụng class đã thay đổi cũng không được biết rằng thay đổi đã xảy ra. Giao diện của class cần bảo vệ các "bí mật" bên trong của nó.

Một số khu vực thường xuyên có khả năng thay đổi

> **Tham khảo chéo**: Một trong những kỹ thuật mạnh mẽ để dự đoán thay đổi là sử dụng phương pháp table-driven (phương pháp điều khiển dựa trên bảng). Xem chi tiết ở Chương 18, "Table-Driven Methods".

- **Business rules (Quy tắc nghiệp vụ):** Đây thường là nguồn gốc của các thay đổi phần mềm liên tục, ví dụ: Quốc hội thay đổi cấu trúc thuế, công đoàn thương lượng lại hợp đồng, hoặc công ty bảo hiểm thay đổi bảng giá. Nếu bạn tuân thủ nguyên tắc information hiding (giấu thông tin), logic dựa trên các quy tắc này sẽ không bị rải rác trong chương trình mà sẽ được ẩn trong một điểm duy nhất của hệ thống cho đến khi cần thay đổi.

- **Hardware dependencies (Phụ thuộc phần cứng):** Bao gồm giao tiếp với màn hình, máy in, bàn phím, chuột, ổ đĩa, loa, thiết bị kết nối,... Hãy cô lập các phụ thuộc này trong subsystem (hệ thống con) hoặc class riêng biệt. Điều này sẽ giúp ích khi chuyển phần mềm sang môi trường phần cứng mới, hoặc khi phát triển phần mềm cho phần cứng chưa ổn định. Bạn có thể viết phần mềm mô phỏng tương tác phần cứng, để subsystem phụ trách giao tiếp phần cứng sử dụng mô phỏng này cho đến khi phần cứng sẵn sàng, sau đó chỉ cần thay thế subsystem giao diện phần cứng.

- **Input và output (Nhập xuất):** Ở cấp thiết kế cao hơn so với giao diện phần cứng thô sơ, nhập/xuất là khu vực dễ biến đổi. Nếu ứng dụng tạo file dữ liệu riêng, format các file có thể thay đổi khi ứng dụng phát triển. Định dạng nhập/xuất cho người dùng cũng sẽ thay đổi - vị trí các trường trên màn hình, số trường, thứ tự các trường, v.v. Nên xem xét tất cả giao diện bên ngoài để dự đoán thay đổi.

- **Nonstandard language features (Tính năng ngôn ngữ không chuẩn):** Nhiều trình biên dịch cung cấp các extension (mở rộng) tiện lợi nhưng không đồng nhất. Việc sử dụng các extension này có hai mặt: chúng có thể không sẵn có ở môi trường khác (phần cứng khác, nhà cung cấp khác, hoặc phiên bản mới của cùng nhà cung cấp). Nếu sử dụng extension

không chuẩn, hãy ẩn chúng trong class riêng để dễ thay thế khi chuyển môi trường mới. Tương tự, các library chưa chắc dùng được ở mọi môi trường - hãy ẩn chúng sau interface (giao diện) tùy chỉnh.

- **Difficult design and construction areas (Khu vực thiết kế và xây dựng phức tạp):** Nên cô lập các khu vực này vì có thể thiết kế chưa tốt và bạn buộc phải làm lại. Bằng cách tách biệt, bạn giảm tác động tiêu cực lên toàn hệ thống.

- **Status variables (Biến trạng thái):** Thường thay đổi nhiều hơn các dữ liệu khác. Ví dụ, ban đầu khai báo biến trạng thái lỗi dưới dạng boolean rồi sau thấy nên dùng kiểu liệt kê (enumerated type) như ErrorType_None, ErrorType_Warning, ErrorType_Fatal.

  Để tăng tính linh hoạt và dễ đọc:

  - Không dùng biến boolean cho trạng thái, mà dùng kiểu liệt kê. Việc thêm trạng thái mới sẽ chỉ cần biên dịch lại thay vì sửa nhiều dòng code.
  - Sử dụng routine truy cập thay vì kiểm tra biến trực tiếp. Điều này cho phép thực hiện kiểm tra phức tạp hơn mà không làm code trở nên rối rắm.

- **Data-size constraints (Giới hạn kích thước dữ liệu):** Khi khai báo một mảng kích thước 100, bạn đã công bố thông tin không cần thiết. Thực hiện information hiding không nhất thiết phải tạo class mới, chỉ đơn giản là sử dụng hằng số có tên (như MAX_EMPLOYEES thay cho 100).

Dự Báo Các Mức Độ Thay Đổi Khác Nhau

> **Tham khảo chéo**: Phương pháp dự báo thay đổi trong phần này không phải là thiết kế vượt trước hoặc lập trình trước. Để biết thêm, xem "A program contains code that seems like it might be needed someday" ở Mục 24.2.

Một kỹ thuật tốt để xác định khu vực dễ thay đổi là xác lập tập con nhỏ nhất của chương trình mà người dùng cần – đây là core (lõi) của hệ thống, ít có khả năng thay đổi. Tiếp theo, xác định các phần bổ sung tối thiểu cho hệ thống, kể cả những thay đổi có vẻ rất nhỏ. Khi xem xét thay đổi về chức năng, đừng quên cân nhắc cả những thay đổi về chất lượng, ví dụ: đảm bảo thread-safe (an toàn đa luồng), chuẩn bị cho việc bản địa hóa/làm đa ngôn ngữ,... Những khu vực này được coi là cải tiến tiềm năng và nên được thiết kế tuân thủ nguyên tắc information hiding.

Bằng cách xác định core trước tiên, bạn dễ thấy thành phần nào chỉ là phần bổ sung, và nhờ đó có thể giấu đi mọi thay đổi bổ sung đằng sau cấu trúc phù hợp.

Giữ Kết Nối Lỏng Lẻo (Loose Coupling)

Coupling mô tả mức độ liên hệ chặt chẽ giữa một class hay routine với các class hoặc routine khác. Mục tiêu là tạo ra các class và routine với kết nối nhỏ, trực tiếp, rõ ràng và linh hoạt – hay còn gọi là "loose coupling" (kết nối lỏng lẻo). Khái niệm này áp dụng cho cả class và routine, trong phần này sẽ dùng từ "module" để chỉ cả hai loại trên.

Sự kết nối giữa các module nên đủ lỏng để một module dễ được dùng bởi các module khác. Ví dụ, các toa tàu mô hình được nối bằng móc ngược nhau – kết nối rất dễ dàng. Nếu thay vì vậy phải vặn ốc, nối dây, hay chỉ có thể ghép một số loại xe nhất định với nhau, việc kết nối sẽ phức tạp hơn rất nhiều. Kết nối giữa các module phần mềm nên đơn giản y như vậy. Nên tạo các module ít phụ thuộc vào nhau, càng cô lập càng tốt (giống quan hệ giữa cộng sự kinh doanh chứ không phải như cặp song sinh dính liền).

Ví dụ:

- Một routine như `sin()` là loosely coupled vì chỉ cần một giá trị đầu vào (góc).
- Một routine như `InitVars(var1, var2, var3, ..., varN)` có coupling chặt hơn vì cần biết thông tin nội bộ về nhiều biến.
- Hai class cùng dùng chung global data (dữ liệu toàn cục) thậm chí còn bị coupling chặt hơn nữa.

Tiêu chí đánh giá Coupling

Dưới đây là một số tiêu chí để đánh giá mức độ coupling giữa các module:

- **Size (Kích thước):** Số lượng kết nối giữa các module. Coupling nhỏ là lý tưởng vì càng ít kết nối, việc tích hợp module với các module khác càng dễ dàng. Routine chỉ có một tham số sẽ coupling lỏng hơn routine với sáu tham số. Một class với bốn phương thức public rõ ràng sẽ coupling lỏng hơn một class có tới 37 phương thức public.

- **Visibility (Tính hiển thị):** Chỉ mức độ dễ nhận ra của kết nối giữa hai module.

(Bản dịch kết thúc ở điểm này theo yêu cầu nội dung gốc)

Dịch sang tiếng Việt (Phong cách học thuật, trang trọng, giữ nguyên đoạn mã, định dạng markdown)

> **Lưu ý:** Văn bản gốc không có đoạn mã, chỉ là diễn giải lý thuyết. Các thuật ngữ chuyên ngành sẽ được giữ nguyên tiếng Anh với chú thích khi xuất hiện lần đầu.

Sự Liên kết Modul (Coupling) & Tính Linh Hoạt (Flexibility)

Cách tiếp cận này tương tự như quảng bá sản phẩm; bạn sẽ nhận được nhiều sự đánh giá tích cực nếu có thể làm rõ các mối liên kết của mình một cách minh bạch nhất có thể. Việc truyền dữ liệu thông qua danh sách tham số là một cách tạo ra liên kết rõ ràng và do đó là tốt. Ngược lại, việc sửa đổi dữ liệu toàn cục (global data) để một module khác có thể sử dụng dữ liệu đó là một liên kết khó nhận biết và do đó không được khuyến khích. Việc tài liệu hóa các liên kết dựa trên dữ liệu toàn cục sẽ giúp chúng trở nên minh bạch hơn, mặc dù chỉ cải thiện một phần nhỏ.

Tính Linh Hoạt

Tính linh hoạt (flexibility) đề cập đến mức độ dễ dàng trong việc thay đổi các kết nối giữa các module. Lý tưởng nhất, bạn sẽ mong muốn có một kết nối tương tự như cổng USB trên máy tính thay vì dây trần và máy hàn. Tính linh hoạt phần nào là hệ quả của các đặc điểm liên kết khác, nhưng cũng có một số khác biệt riêng.

Giả sử bạn có một routine dùng để tra cứu số ngày nghỉ phép mà một nhân viên nhận được mỗi năm, dựa trên ngày tuyển dụng và phân loại công việc (job classification). Đặt tên routine này là `LookupVacationBenefit()`. Giả sử trong một module khác bạn có một đối tượng nhân viên (employee object) chứa ngày tuyển dụng và phân loại công việc, cùng các thông tin khác, và module này truyền đối tượng đó cho `LookupVacationBenefit()`. Theo các tiêu chí khác, hai module này sẽ được xem là liên kết lỏng lẻo (loosely coupled). Kết nối giữa hai module thông qua employee là rõ ràng, chỉ có một kết nối.

Tuy nhiên, giả sử bạn cần sử dụng module `LookupVacationBenefit()` từ một module thứ ba không có đối tượng employee mà chỉ có ngày tuyển dụng và phân loại công việc. Đột ngột, `LookupVacationBenefit()` trở nên kém thân thiện, không sẵn sàng hợp tác với module mới này.

Để module thứ ba sử dụng được `LookupVacationBenefit()`, nó buộc phải hiểu về lớp Employee (Employee class). Có thể giả lập (dummy up) một đối tượng employee chỉ với hai thuộc tính cần thiết, nhưng điều này lại đòi hỏi phải biết nội tại của `LookupVacationBenefit()`, cụ thể là biết trước những trường nào được sử dụng. Đây là giải pháp 'vá víu' (kludge) và thiếu thẩm mỹ. Phương án khác là sửa đổi `LookupVacationBenefit()` để nó nhận trực tiếp ngày tuyển dụng và phân loại công việc thay vì một đối tượng employee. Dù chọn phương án nào, module ban đầu hóa ra kém linh hoạt hơn so với nhận định ban đầu.

Câu chuyện có cái kết có hậu nếu một module "kém thân thiện" chịu thay đổi để trở nên linh hoạt hơn—trong trường hợp này là thông qua việc thay đổi tham số nhận vào là ngày tuyển dụng và phân loại công việc thay vì đối tượng employee.

Tóm lại, module càng dễ được các module khác gọi tới thì càng liên kết lỏng lẻo (loosely coupled), và điều này là tốt bởi nó gia tăng tính linh hoạt cũng như khả năng bảo trì. Khi xây dựng cấu trúc hệ thống, hãy chia nhỏ chương trình dựa trên tiêu chí tối thiểu hóa sự liên kết. Nếu ví việc lập trình là việc tách gỗ, bạn nên cố gắng tách dọc theo thớ gỗ.

Các Loại Liên Kết (Kinds of Coupling)

Dưới đây là các loại liên kết phổ biến bạn có thể gặp:

- **Liên kết thông qua tham số dữ liệu đơn giản (Simple-data-parameter coupling):** Hai module được xem là liên kết kiểu này nếu tất cả dữ liệu truyền giữa chúng đều là kiểu dữ liệu nguyên thủy (primitive data types) và tất cả dữ liệu này được truyền qua danh sách tham số. Đây là loại liên kết bình thường và được chấp nhận.
- **Liên kết đối tượng đơn giản (Simple-object coupling):** Một module được xem là liên kết đối tượng đơn giản với một đối tượng nếu nó khởi tạo (instantiate) đối tượng đó. Loại liên kết này là phù hợp.
- **Liên kết thông qua tham số đối tượng (Object-parameter coupling):** Hai module được liên kết với nhau kiểu này nếu Object1 yêu cầu Object2 truyền cho mình một Object3. Liên kết này chặt chẽ hơn việc Object1 chỉ yêu cầu Object2 truyền dữ liệu kiểu nguyên thủy bởi vì Object2 cần phải biết về Object3.
- **Liên kết ngữ nghĩa (Semantic coupling):** Đây là loại liên kết "ẩn sâu" nhất, xảy ra khi một module sử dụng không phải thành phần cú pháp (syntactic element) mà là tri thức về ý nghĩa, hoạt động bên trong (semantic knowledge) của một module khác. Một số ví dụ:
    - Module1 truyền cho Module2 một 'control flag', thông báo Module2 nên thực hiện gì. Cách làm này đòi hỏi Module1 phải giả định về nội bộ của Module2, cụ thể là cách mà Module2 xử lý 'control flag'. Nếu Module2 định

nghĩa một kiểu dữ liệu cụ thể cho 'control flag' (kiểu liệt kê—enumerated type—hoặc đối tượng), cách dùng này có thể chấp nhận được.
- Module2 sử dụng dữ liệu toàn cục sau khi dữ liệu này đã bị sửa đổi bởi Module1. Cách làm này yêu cầu Module2 phải giả định rằng Module1 đã sửa đổi dữ liệu phù hợp với nhu cầu của nó và rằng Module1 đã được gọi đúng thời điểm.
- Giao diện của Module1 quy định routine `Module1_Initialize()` cần được gọi trước khi routine `Module1_Routine()` được gọi. Tuy nhiên, Module2 biết rằng `Module1_Routine()` sẽ tự động gọi `Module1_Initialize()`, nên chỉ khởi tạo Module1 rồi gọi `Module1_Routine()` mà bỏ qua bước gọi `Module1_Initialize()`.
- Module1 truyền một đối tượng cho Module2, và vì biết rằng Module2 chỉ sử dụng ba trên bảy phương thức của đối tượng nên Module1 chỉ khởi tạo dữ liệu cần thiết cho ba phương thức đó.
- Module1 truyền `BaseObject` cho Module2. Vì Module2 biết rằng thực tế Module1 truyền `DerivedObject`, nên Module2 ép kiểu (cast) từ `BaseObject` sang `DerivedObject` và gọi các phương thức đặc thù của `DerivedObject`.

Liên kết ngữ nghĩa (semantic coupling) rất nguy hiểm, bởi vì việc thay đổi mã nguồn trong module được dùng có thể khiến module sử dụng gặp lỗi mà trình biên dịch hoàn toàn không phát hiện được. Khi các mã dạng này bị lỗi, lỗi thường rất khó xác định, dường như không liên quan trực tiếp tới thay đổi vừa thực hiện ở module được dùng, biến việc gỡ lỗi thành nhiệm vụ gần như bất khả thi.

Mục tiêu của liên kết lỏng lẻo là cho phép một module cung cấp tầng trừu tượng bổ sung; một khi bạn đã viết nó, bạn có thể yên tâm sử dụng mà không cần quan tâm tới chi tiết bên trong. Điều này giúp giảm độ phức tạp tổng thể của chương trình và cho phép bạn chỉ tập trung vào một vấn đề tại một thời điểm. Nếu sử dụng một module đòi hỏi bạn phải chú ý đến nhiều thứ cùng lúc—ví dụ như hiểu nội bộ vận hành, sửa đổi dữ liệu toàn cục hoặc chức năng không rõ ràng—thì sức mạnh trừu tượng của module bị mất, và khả năng kiểm soát độ phức tạp của hệ thống bị suy yếu hoặc loại bỏ.

Lớp (class) và routine trước hết là công cụ trí tuệ giúp giảm độ phức tạp. Nếu chúng không khiến công việc của bạn đơn giản hơn, tức là chúng chưa hoàn thành nhiệm vụ của mình.

> **KEY POINT:**
> Lớp và routine là công cụ chủ yếu giúp giảm độ phức tạp cho nhà phát triển phần mềm. Nếu chúng không giúp bạn làm việc đơn giản hơn, hãy xem lại thiết kế.

Tìm hiểu về Các Design Pattern Phổ biến

Design pattern (Mẫu thiết kế) cung cấp nhân lõi của các giải pháp đã được xây dựng sẵn, có thể áp dụng để giải quyết nhiều vấn đề phổ biến trong phần mềm. Một số vấn đề phần mềm cần các giải pháp sáng tạo dựa trên nguyên lý nền tảng, nhưng phần lớn các vấn đề đều khá giống những vấn đề đã từng tồn tại, từ đó có thể giải quyết bằng các giải pháp quen thuộc, tức là các pattern.

Các pattern thường gặp bao gồm:

- Adapter,
- Bridge,
- Decorator,
- Facade,
- Factory Method,
- Observer,
- Singleton,
- Strategy,
- Template Method.

Cuốn sách *Design Patterns* của Erich Gamma, Richard Helm, Ralph Johnson, và John Vlissides (1995) là nguồn tài liệu mô tả đầy đủ về các pattern này.

Lợi ích của Việc Sử dụng Design Pattern

- **Pattern giúp giảm độ phức tạp thông qua trừu tượng hóa sẵn có:**
  Nếu bạn nói, "Đoạn mã này sử dụng Factory Method để khởi tạo các instance của lớp dẫn xuất," các lập trình viên khác sẽ lập tức hiểu rằng mã của bạn có mối quan hệ phức tạp đặc trưng cho Factory Method.

  Factory Method là một pattern cho phép tạo instance của bất kỳ lớp dẫn xuất nào từ một lớp cơ sở cụ thể, mà không cần quản lý các lớp dẫn xuất ở bất kỳ đâu ngoài Factory Method. Để tìm hiểu sâu hơn về Factory Method, tham khảo mục "Replace Constructor with Factory Method" trong sách *Refactoring* (Fowler 1999).

  Bạn không cần trình bày từng dòng mã, người khác vẫn nắm được cách tiếp cận thiết kế của bạn.

- **Pattern giúp giảm lỗi bằng cách chuẩn hóa chi tiết các giải pháp:**
  Vấn đề thiết kế phần mềm thường xuất hiện các tiểu tiết chỉ bộc lộ sau khi vấn đề đã được giải quyết vài lần. Vì pattern đại diện cho Chuẩn hóa cách giải quyết vấn đề, chúng rất "giàu kinh nghiệm", đúc kết từ nhiều thử—sai trong quá trình phát triển phần mềm.

  Sử dụng một design pattern tương tự như sử dụng thư viện mã thay vì tự viết từ đầu. Chắc hẳn ai cũng từng tự cài thuật toán Quicksort, nhưng khả năng rất cao phiên bản tự viết sẽ không hoàn chỉnh ngay lần đầu tiên. Cũng như vậy, nhiều vấn đề thiết kế phần mềm giống những vấn đề từng gặp, do đó tốt nhất nên sử dụng giải pháp đã được xây dựng sẵn thay vì tạo ra giải pháp hoàn toàn mới.

- **Pattern cung cấp giá trị heuristic (gợi ý) bằng việc đề xuất phương án thiết kế thay thế:**
  Nhà thiết kế am hiểu các pattern dễ dàng cân nhắc, tự hỏi "Pattern nào phù hợp với vấn đề thiết kế hiện tại?" Việc so sánh các lựa chọn quen thuộc này dễ dàng hơn rất nhiều so với việc phải xây dựng giải pháp hoàn toàn mới. Hơn thế, mã nguồn dựa trên các pattern quen thuộc cũng dễ đọc, dễ hiểu hơn so với mã tùy biến.

- **Pattern giúp trao đổi về thiết kế ở cấp độ cao hơn, nhanh hơn:**
  Ngoài lợi ích quản lý độ phức tạp, design pattern còn tăng tốc quá trình thảo luận thiết kế bằng cách cho phép thảo luận ở mức trừu tượng cao hơn. Nếu bạn nói, "Tôi chưa quyết định nên sử dụng Creator hay Factory Method trong trường hợp này," bạn đã truyền đạt rất nhiều chỉ với một câu nói—đương nhiên với điều kiện người nghe cũng hiểu các pattern được đề cập. Nếu không, việc đào sâu vào chi tiết mã cho từng pattern để so sánh và lựa chọn sẽ tốn thời gian hơn nhiều.

Nếu bạn chưa quen với các design pattern, Bảng 5-1 sau đây sẽ tổng hợp một số pattern phổ biến nhằm khơi gợi sự quan tâm của bạn.

Bảng 5-1: Các Design Pattern Phổ biến

Pattern	Mô tả
Abstract Factory	Hỗ trợ việc tạo ra tập hợp các đối tượng liên quan bằng cách chỉ định loại tập hợp, nhưng không chỉ rõ từng loại đối tượng cụ thể
Adapter	Chuyển đổi (convert) giao diện (interface) của một lớp sang một giao diện khác

---

Ghi chú: Bản dịch đã hiệu chỉnh các lỗi đánh máy và ngắt dòng không hợp lý từ văn bản gốc để đảm bảo tính liền mạch và dễ đọc.
Kiểm tra lần cuối: Thuật ngữ được giữ nguyên, định dạng và chú thích đầy đủ, không dịch đoạn mã (không có mã), diễn đạt trang trọng, mạch lạc, chuẩn học thuật.

Các Mẫu Thiết Kế Phổ Biến

- **Composite**
  Bao gồm một đối tượng chứa các đối tượng bổ sung cùng kiểu với chính nó, nhờ đó mã phía khách hàng chỉ cần tương tác với đối tượng ở cấp cao nhất mà không phải quan tâm đến các đối tượng chi tiết bên trong.

- **Decorator**
  Gắn thêm trách nhiệm cho một đối tượng một cách động mà không cần tạo các lớp con riêng biệt cho từng cấu hình trách nhiệm có thể xảy ra.

- **Facade**
  Cung cấp một giao diện nhất quán cho phần mã vốn dĩ không có một giao diện thống nhất.

- **Factory Method**
  Tạo ra các đối tượng từ các lớp dẫn xuất của một lớp cơ sở cụ thể mà không cần phải quản lý riêng từng lớp dẫn xuất ở bất cứ đâu ngoài Factory Method.

- **Iterator**
  Là một đối tượng phục vụ (server object) cung cấp khả năng truy cập tuần tự tới từng phần tử trong một tập hợp.

- **Observer**
  Đảm bảo đồng bộ giữa nhiều đối tượng bằng cách giao trách nhiệm thông báo thay đổi của bất kỳ thành viên nào trong tập hợp cho một đối tượng trung gian.

- **Singleton**
  Cung cấp truy cập toàn cục tới một lớp mà chỉ tồn tại duy nhất một thể hiện (instance).

- **Strategy**
  Định nghĩa một tập hợp các thuật toán hoặc hành vi có thể thay đổi lẫn nhau một cách linh hoạt trong quá trình thực thi.

- **Template Method**
  Định nghĩa cấu trúc tổng thể của một thuật toán nhưng để lại một số chi tiết triển khai cho các lớp con.

Giá trị của Mẫu Thiết Kế

Nếu bạn chưa từng thấy các mẫu thiết kế (design pattern) trước đây, phản xạ đầu tiên của bạn khi đọc các mô tả ở Bảng 5-1 có thể là: “Dĩ nhiên, tôi đã biết hầu hết các ý tưởng này rồi.” Chính phản ứng đó là lý do lớn khiến các mẫu thiết kế trở nên quý giá. Các mẫu này quen thuộc với phần lớn lập trình viên có kinh nghiệm, và việc đặt tên dễ nhận biết cho chúng giúp nâng cao hiệu quả và uy lực trong quá trình giao tiếp về mặt thiết kế.

5.3 Các Khối Xây Dựng Của Thiết Kế: Heuristic

Cảnh Báo Khi Áp Dụng Mẫu Thiết Kế

Một cái bẫy tiềm ẩn khi sử dụng mẫu thiết kế là “bắt” mã phải tuân theo mẫu. Đôi khi, việc điều chỉnh mã một chút theo một mẫu được công nhận sẽ giúp mã dễ hiểu hơn. Tuy nhiên, nếu phải điều chỉnh quá nhiều để mã trông giống một mẫu tiêu chuẩn, điều này có thể làm tăng độ phức tạp.

Một nguy cơ khác là “feature-itis”: sử dụng mẫu thiết kế chỉ vì muốn thử nghiệm, thay vì vì nó thật sự phù hợp với giải pháp thiết kế.

Tóm lại, các mẫu thiết kế là công cụ mạnh mẽ để kiểm soát độ phức tạp. Bạn có thể tham khảo các mô tả chi tiết hơn trong nhiều đầu sách giá trị được liệt kê ở cuối chương này.

Các Heuristic Khác

Các phần trước đã mô tả những heuristic chủ đạo trong thiết kế phần mềm. Dưới đây là một số heuristic khác tuy không phải lúc nào cũng hữu ích nhưng vẫn đáng được nhắc đến.

Hướng Đến Tính Kết Chặt (Strong Cohesion)

Tính kết chặt (cohesion) xuất phát từ thiết kế có cấu trúc và thường được thảo luận cùng với độ kết nối (coupling). Cohesion đề cập đến mức độ các phương thức trong một lớp hoặc khối mã trong một hàm hỗ trợ cho một mục đích trung tâm – tức mức độ tập trung của lớp. Các lớp chứa chức năng liên quan mật thiết được mô tả là có tính kết chặt mạnh. Mục tiêu heuristic là hướng đến sự kết chặt mạnh nhất có thể. Đây là công cụ hữu ích để quản lý độ phức tạp, vì khi khối mã trong một lớp tập trung phục vụ một mục đích duy nhất, não bộ của bạn sẽ dễ ghi nhớ mọi thứ mà mã thực hiện.

Việc suy nghĩ về cohesion ở mức độ hàm đã là một heuristic hữu dụng trong nhiều thập kỷ và đến nay vẫn còn nguyên giá trị. Ở cấp lớp, heuristic này phần lớn đã được thay thế bởi heuristic tổng quát hơn về trừu tượng hóa (abstraction), vốn được đề cập ở phần trước của chương này và trong Chương 6. Tuy nhiên, abstraction cũng hữu dụng ở cấp độ hàm, song ở mức cân bằng hơn với cohesion tại lớp chi tiết đó.

Xây Dựng Hệ Thống Phân Cấp (Hierarchies)

Hệ thống phân cấp (hierarchy) là một cấu trúc thông tin theo tầng, trong đó đại diện tổng quát hay trừu tượng nhất của khái niệm nằm ở đỉnh, với các phần thể hiện ngày càng chi tiết và chuyên biệt ở các tầng dưới. Trong phát triển phần mềm, có thể thấy hierarchy trong các hệ phân cấp lớp (class hierarchy), cũng như hierarchy về lời gọi hàm (routine-calling hierarchy).

Các hệ thống phân cấp đã là công cụ quan trọng để kiểm soát các tập thông tin phức tạp trong ít nhất 2.000 năm. Aristotle sử dụng hierarchy để tổ chức vương quốc động vật. Con người thường dùng dàn bài (outline) để tổ chức thông tin phức tạp (như cấu trúc cuốn sách này). Nghiên cứu cho thấy con người nói chung cảm thấy tự nhiên khi tổ chức thông tin phức tạp dưới dạng phân cấp. Khi họ vẽ một đối tượng phức tạp như ngôi nhà, họ cũng vẽ một cách phân cấp: Đầu tiên là phác thảo tổng quát, rồi đến cửa sổ, cửa ra vào, sau đó mới tới các chi tiết nhỏ. Họ không vẽ từng viên gạch, từng tấm ngói hay từng chiếc đinh một (Simon 1996).

Phân cấp là công cụ hữu hiệu để đạt được Yêu cầu Kỹ thuật Cốt lõi của Phần mềm, vì nhờ đó bạn chỉ cần tập trung vào mức chi tiết mà bạn quan tâm tại thời điểm hiện tại. Các chi tiết khác không mất đi, chỉ được đẩy xuống tầng khác để bạn có thể xem xét khi cần thiết thay vì luôn phải để ý tới tất cả cùng lúc.

Chuẩn Hóa Hợp Đồng Lớp (Class Contracts)

> Tham khảo thêm về hợp đồng: xem “Sử dụng assert để tài liệu hóa và kiểm thử điều kiện trước (precondition) và điều kiện sau (postcondition)” ở Mục 8.2.

Ở mức chi tiết hơn, việc coi mỗi giao diện lớp như một hợp đồng (contract) với phần còn lại của chương trình giúp bạn có nhiều hiểu biết sâu sắc. Thông thường, hợp đồng sẽ là: "Nếu bạn cung cấp dữ liệu x, y, z với đặc điểm a, b, c như đã cam kết, thì tôi hứa sẽ thực hiện các thao tác 1, 2, 3 trong giới hạn 8, 9, 10." Cam kết của phía khách hàng lớp (client) gọi là precondition, còn cam kết của đối tượng với client là postcondition.

Hợp đồng giúp giảm thiểu độ phức tạp nhờ về lý thuyết, đối tượng có thể bỏ qua các hành vi ngoài phạm vi hợp đồng. Tuy nhiên, trên thực tế, vấn đề này phức tạp hơn nhiều.

Phân Công Trách Nhiệm (Assign Responsibilities)

Một heuristic khác là cân nhắc cách phân công các trách nhiệm cho từng đối tượng. Việc đặt câu hỏi mỗi đối tượng nên chịu trách nhiệm về điều gì cũng giống như hỏi về thông tin nào cần được che giấu, nhưng cách phân tích này thường cho ra câu trả lời tổng quát hơn, điều này giúp heuristic có giá trị đặc biệt.

Thiết Kế Hướng Tới Kiểm Thử (Design for Test)

Một cách tư duy có thể dẫn tới các phát hiện thiết kế thú vị là đặt câu hỏi hệ thống sẽ trông như thế nào nếu bạn thiết kế nhằm mục đích dễ kiểm thử. Ví dụ: Ta có cần tách biệt giao diện người dùng khỏi phần mã còn lại để có thể kiểm thử độc lập? Có cần tổ chức các phân hệ sao cho giảm thiểu phụ thuộc giữa chúng? Việc thiết kế hướng tới kiểm thử thường dẫn đến các giao diện lớp được chuẩn hóa hơn, điều này nhìn chung là lợi ích.

Tránh Thất Bại (Avoid Failure)

Giáo sư kỹ thuật xây dựng Henry Petroski đã viết cuốn sách thú vị "Design Paradigms: Case Histories of Error and Judgment in Engineering" (Petroski 1994), ghi lại lịch sử những thất bại trong thiết kế cầu. Ông cho rằng nhiều vụ thất bại ấn tượng xảy ra do chỉ tập trung vào các thành công trước đó mà không cân nhắc đủ các kịch bản thất bại tiềm tàng. Ông kết luận rằng các thất bại như cầu Tacoma Narrows đã có thể tránh được nếu các kỹ sư đã đặc biệt chú ý tới các nguyên nhân gây hỏng, thay vì chỉ sao chép thuộc tính của những thiết kế thành công khác.

Những sự cố an ninh nổi bật của nhiều hệ thống nổi tiếng trong một vài năm qua đồng thuận với quan điểm rằng chúng ta cần áp dụng các quan sát về thất bại thiết kế của Petroski vào lĩnh vực phần mềm.

Lựa Chọn Thời Điểm Ràng Buộc Giá Trị Một Cách Có Ý Thức (Choose Binding Time Consciously)

> Tham khảo thêm về thời điểm ràng buộc (binding time): xem Mục 10.6, "Binding Time".

Binding time (thời điểm ràng buộc) là thời điểm một giá trị cụ thể được gán cho một biến. Mã ràng buộc giá trị sớm thường đơn giản hơn nhưng cũng kém linh hoạt hơn. Đôi khi, bạn có thể có được ý tưởng thiết kế mới từ việc đặt những câu hỏi như: Điều gì xảy ra nếu ràng buộc giá trị này sớm hơn? Nếu ràng buộc giá trị trễ hơn thì sao? Nếu tôi khởi tạo bảng dữ liệu này ngay trong mã? Nếu tôi lấy giá trị của biến từ người dùng tại thời điểm chạy?

Tạo Các Trung Tâm Kiểm Soát (Make Central Points of Control)

P.
J. Plauger cho rằng mối quan tâm lớn nhất của ông là "Nguyên tắc Một Nơi Đúng – nên có duy nhất một nơi để tìm kiếm bất kỳ đoạn mã không tầm thường nào, và một nơi duy nhất khi cần thay đổi bảo trì hợp lý" (Plauger 1993). Việc kiểm soát có thể được tập trung ở các lớp, hàm, macro tiền xử lý (preprocessor), tệp #include, thậm chí một hằng số có tên cũng là ví dụ về một trung tâm kiểm soát.

Lợi ích về mặt giảm độ phức tạp là: bạn phải tìm kiếm ở càng ít nơi, thì việc thay đổi trở nên dễ dàng và an toàn hơn.

Cân Nhắc Dùng Phép Thử-Thô Bạo (Brute Force)

> Khi còn phân vân, hãy dùng brute force
> — Butler Lampson

Một heuristic mạnh mẽ khác là giải pháp brute force (thử-thô bạo: duyệt hết/gồm hết/so sánh cạn kiệt). Đừng đánh giá thấp nó. Một giải pháp brute force nhưng hoạt động còn hơn một giải pháp tinh tế mà không hoạt động được. Có thể mất rất lâu để một giải pháp tinh tế hoạt động đúng. Khi mô tả lịch sử các thuật toán tìm kiếm, Donald Knuth chỉ ra rằng dù thuật toán tìm kiếm nhị phân đầu tiên được công bố năm 1946, phải mất 16 năm nữa mới có người phát triển thuật toán tìm kiếm chính xác cho mọi cỡ danh sách (Knuth 1998). Thuật toán tìm kiếm nhị phân tuy đẹp, nhưng một giải pháp brute force – ví dụ duyệt tuần tự – thường là đủ.

Vẽ Sơ Đồ (Draw a Diagram)

Sơ đồ là một heuristic mạnh mẽ khác. Một bức tranh giá trị 1000 từ – theo một nghĩa nào đó. Bạn thực sự muốn lược bỏ hầu hết trong 1000 từ đó, vì mục đích của sơ đồ là thể hiện vấn đề ở mức trừu tượng cao hơn. Đôi khi bạn cần giải quyết vấn đề

chi tiết, song nhiều lúc bạn chỉ muốn làm việc ở mức tổng thể hơn.

Thiết Kế Hướng Đến Tính Module (Keep Your Design Modular)

Mục tiêu của modularity (tính module hóa) là làm cho mỗi hàm hoặc lớp như một "hộp đen" (black box): bạn biết những gì đầu vào và đầu ra, nhưng không cần biết điều gì diễn ra bên trong. Một black box có giao diện đơn giản đến mức và chức năng được xác định rõ ràng đến nỗi với bất kỳ đầu vào cụ thể nào bạn đều có thể dự đoán chính xác đầu ra tương ứng.

Khái niệm modularity có liên quan mật thiết đến information hiding (che giấu thông tin), encapsulation (đóng gói), và nhiều heuristic thiết kế khác.

Tóm tắt các Heuristic Thiết kế

> Điều rất đáng lo ngại là cùng một lập trình viên hoàn toàn có thể thực hiện cùng một nhiệm vụ theo hai hoặc ba cách khác nhau, đôi khi một cách vô thức, nhưng cũng rất thường là để thay đổi, hoặc để tạo ra sự biến hóa tinh tế.
> — A. R. Brown và W. A. Sampson

Tóm tắt các heuristic thiết kế chủ đạo:

- **Tìm kiếm các đối tượng trong thế giới thực**
- **Hình thành các trừu tượng nhất quán**
- **Đóng gói chi tiết triển khai**
- **Kế thừa khi có thể**
- **Ẩn các bí mật (Information Hiding)**
- **Xác định các khu vực có khả năng thay đổi**
- **Giữ cho coupling (liên kết) lỏng lẻo**
- **Tìm kiếm các mẫu thiết kế phổ biến (Design Patterns)**

Một số heuristic sau đây cũng thường hữu ích:

- **Hướng tới tính cohesion (kết dính) mạnh**
- **Xây dựng các hệ phân cấp**
- **Chính thức hóa các hợp đồng lớp (Class Contracts)**
- **Phân công trách nhiệm**
- **Thiết kế dễ kiểm thử**
- **Tránh thất bại**
- **Chọn thời điểm binding (ràng buộc) một cách chủ động**
- **Tạo các điểm kiểm soát trung tâm**
- **Cân nhắc sử dụng phương pháp brute force**
- **Vẽ sơ đồ**
- **Giữ cho thiết kế có tính module**

Hướng dẫn sử dụng các heuristic

Phương pháp tiếp cận thiết kế trong phần mềm có thể học hỏi từ các lĩnh vực thiết kế khác. Một trong những cuốn sách gốc về heuristic trong giải quyết vấn đề là *How to Solve It* của G. Polya (1957). Phương pháp tổng quát của Polya tập trung vào giải quyết vấn đề trong toán học. Hình 5-10 tóm tắt cách tiếp cận này (dẫn lại từ tác giả với một số nhấn mạnh):

1. **Hiểu vấn đề**
    - Bạn phải hiểu được vấn đề.
    - Cái chưa biết là gì? Dữ liệu là gì? Điều kiện ra sao? Có thể thỏa mãn điều kiện không? Điều kiện có đủ để xác định cái chưa biết không, hay không đủ, dư thừa hoặc mâu thuẫn?
    - Vẽ hình minh họa. Đưa ra ký hiệu phù hợp. Phân biệt các phần của điều kiện. Bạn có thể ghi lại chúng không?
2. **Lập kế hoạch**
    - Tìm mối liên hệ giữa dữ liệu và cái chưa biết.
    - Nếu không tìm được kết nối trung gian, bạn có thể cần xem xét các vấn đề phụ trợ.
    - Bạn nên xây dựng được một kế hoạch giải quyết vấn đề.
    - Đã từng gặp vấn đề này chưa? Gặp ở dạng khác chưa? Biết vấn đề liên quan nào không? Có định lý nào hữu ích không?
    - Nhìn vào cái chưa biết và nghĩ đến một vấn đề quen thuộc có đặc điểm tương tự.
    - Có thể sử dụng kết quả, phương pháp hoặc cần thêm đối tượng phụ nào?
    - Có thể phát biểu lại vấn đề không? Hoặc phát biểu theo cách khác? Xem lại các định nghĩa.

- Nếu không giải được vấn đề được đưa ra, hãy thử giải vấn đề liên quan trước.
- Có thể tưởng tượng vấn đề liên quan dễ tiếp cận hơn không? Tổng quát hơn? Đặc biệt hơn? Tương tự hơn? Có thể giải một phần không?
- Giữ lại một phần điều kiện, bỏ phần khác; khi đó cái chưa biết bị ràng buộc tới mức nào, nó có thể biến đổi ra sao?
- Có thể rút ra điều gì hữu ích từ dữ liệu? Có thể nghĩ ra dữ liệu khác phù hợp không?
- Có thể thay đổi cái chưa biết hoặc dữ liệu, hoặc cả hai khi cần thiết, để hai thứ gần nhau hơn?
- Đã sử dụng hết dữ liệu, sử dụng toàn bộ điều kiện, và các khái niệm thiết yếu chưa?

3. **Thực hiện kế hoạch**

   - Triển khai kế hoạch đã đề ra.
   - Kiểm tra mỗi bước của kế hoạch. Có rõ ràng là bước này đúng không? Có thể chứng minh được không?

4. **Xem xét lại**

   - Xem xét nghiệm lại kết quả giải quyết vấn đề.
   - Có thể kiểm tra kết quả không? Có thể kiểm tra lập luận? Có thể chứng minh kết quả theo cách khác? Có thể nhận ra ngay kết quả không?
   - Có thể áp dụng kết quả hoặc phương pháp này cho vấn đề khác không?

Hình 5-10: G. Polya đã phát triển một phương pháp giải quyết vấn đề trong toán học, cũng có ích trong việc giải quyết vấn đề trong thiết kế phần mềm (*Polya 1957*).

Một số chỉ dẫn hiệu quả khi ứng dụng heuristic thiết kế

Một trong những chỉ dẫn hiệu quả nhất là **đừng bị mắc kẹt với chỉ một phương pháp**. Nếu vẽ sơ đồ thiết kế bằng UML (Unified Modeling Language) không hiệu quả, hãy thử trình bày bằng tiếng Anh. Viết một chương trình kiểm thử nhỏ. Hãy thử tiếp cận hoàn toàn khác biệt. Nghĩ đến giải pháp brute-force (phép thử toàn bộ). Cứ tiếp tục phác thảo, ghi chú bằng bút chì—bộ não sẽ theo sau ý tưởng. Nếu mọi nỗ lực đều không thành công, hãy tạm thời bỏ qua vấn đề, đi bộ hoặc nghĩ tới thứ khác trước khi quay lại. Đôi khi… **việc tạm rời khỏi vấn đề giúp tìm ra giải pháp nhanh hơn chỉ bằng sự kiên nhẫn đơn thuần.**

Bạn không cần giải quyết toàn bộ vấn đề thiết kế một lần. Nếu bị mắc kẹt, hãy nhận ra rằng vẫn còn điểm chưa đủ thông tin để quyết định. Không cần phải cố gắng giải quyết 20% cuối cùng của thiết kế nếu như lần lặp sau có thể hoàn thiện nó dễ dàng hơn. Tránh ra quyết định tồi dựa trên kinh nghiệm hạn chế khi có thể trì hoãn để đưa ra quyết định tốt hơn dựa trên kinh nghiệm phong phú hơn. Một số người cảm thấy không thoải mái nếu sau một vòng thiết kế chưa có kết luận, nhưng sau khi đã trải nghiệm vài lần thiết kế mà chưa giải quyết hết vấn đề sớm, sẽ thấy việc đó trở nên tự nhiên hơn (*Zahniser 1992, Beck 2000*).

5.4 Thực hành Thiết kế

Phần trước tập trung vào các heuristic liên quan đến thuộc tính thiết kế – những gì bạn mong muốn thiết kế hoàn chỉnh sẽ đạt được. Phần này trình bày về các heuristic thực hành thiết kế, *các bước* bạn có thể thực hiện để thường đạt được kết quả tốt.

Lặp lại (Iterate)

Bạn có thể từng trải nghiệm rằng sau khi lập trình xong, bạn nhận ra nhiều điều quý giá và ước gì được viết lại chương trình đó với hết các hiểu biết vừa đạt được. Hiện tượng đó cũng xuất hiện trong quá trình thiết kế, nhưng các chu kỳ thiết kế ngắn hơn và ảnh hưởng về sau lớn hơn, do đó bạn có thể "lướt qua" vòng lặp thiết kế nhiều lần mà vẫn hiệu quả.

Thiết kế là một quá trình có tính lặp. Bạn thường không đi từ điểm A thẳng đến điểm B mà luôn quay về điểm A rồi lại tiến đến B.

> **Lưu ý then chốt**: Khi bạn thực hiện các phương án thiết kế và thử các cách tiếp cận khác nhau, bạn sẽ quan sát ở cả cấp cao lẫn thấp. Góc nhìn tổng thể (high-level) từ việc giải quyết các vấn đề lớn sẽ giúp bạn thấy được bức tranh chung để định hướng cho chi tiết. Ngược lại, các chi tiết (low-level) tạo nền tảng vững chắc cho các quyết định ở tầng cao. Sự tương tác giữa hai cấp độ tạo nên cấu trúc ổn định hơn là cách thiết kế chỉ từ trên xuống hoặc chỉ từ dưới lên.

Nhiều lập trình viên – thậm chí nhiều người nói chung – gặp khó khăn khi chuyển đổi giữa các cấp độ chi tiết khác nhau. Việc chuyển ý niệm từ góc nhìn tổng thể xuống chi tiết và ngược lại đòi hỏi khả năng trí tuệ và sự linh hoạt, nhưng đó là điều thiết yếu để tạo ra thiết kế hiệu quả. Tham khảo thêm *Conceptual Blockbusting* (*Adams 2001*), được giới thiệu ở phần "Tài nguyên bổ sung" cuối chương, để rèn luyện tư duy linh hoạt này.

> **Tham khảo chéo**: Refactoring (Tái cấu trúc mã) là cách an toàn để thử các phương án thay thế trong mã nguồn. Xem thêm ở Chương 24, “Refactoring”.

Khi bạn vừa có bản thiết kế đầu tiên tạm gọi là "tốt đủ dùng", **đừng dừng lại**! Nỗ lực thứ hai gần như luôn tốt hơn lần đầu, và bạn sẽ học được nhiều hơn từng lần tiếp cận để cải thiện tổng thể thiết kế. Cũng như Thomas Edison đã thử hàng nghìn vật liệu để làm dây tóc bóng đèn mà vẫn kiên trì: *"Tôi đã phát hiện ra một nghìn thứ không hiệu quả."* Đôi khi, giải quyết vấn đề bằng một hướng sẽ đem lại các ý tưởng giúp bạn phát triển thêm giải pháp khác còn tốt hơn.

Chia để trị (Divide and Conquer)

Như Edsger Dijkstra đã đề cập, không ai có thể nhớ hết mọi chi tiết của một chương trình phức tạp, điều này cũng đúng với thiết kế. Hãy chia chương trình thành các khu vực riêng biệt, giải quyết từng phần riêng lẻ. Nếu gặp bế tắc ở một phần, hãy lặp lại quá trình!

Kỹ thuật **incremental refinement** (tinh chỉnh dần dần) là công cụ mạnh để kiểm soát phức tạp. Theo khuyến nghị của Polya trong giải quyết vấn đề toán học: "Hiểu vấn đề, xây dựng kế hoạch, triển khai và sau đó đánh giá lại kết quả."

Tiếp cận thiết kế Top-Down và Bottom-Up

"Top-down" (từ trên xuống) và "bottom-up" (từ dưới lên) tuy nghe có vẻ cũ kỹ, nhưng vẫn hữu ích trong thiết kế phần mềm hướng đối tượng.

- **Thiết kế Top-down** khởi đầu ở mức độ trừu tượng cao: định nghĩa các lớp cơ bản (base classes) hoặc các thành phần tổng quát. Trong quá trình phát triển thiết kế, bạn bổ sung chi tiết, xác định các lớp dẫn xuất (derived classes), các lớp phối hợp (collaborating classes) và các thành phần chi tiết khác.
- **Thiết kế Bottom-up** bắt đầu từ các đối tượng cụ thể (specifics) rồi xây dựng dần lên các cấu trúc tổng hợp (aggregations) và các lớp cơ sở (base classes) dựa trên các đối tượng đó.

Một số người cho rằng nên bắt đầu từ khái quát (generalities) rồi phát triển dần tới cụ thể, số khác lại cho rằng phải xử lý chi tiết quan trọng trước thì mới rút ra được nguyên tắc tổng quát.

Lập luận cho Top-down

Nguyên tắc chỉ đạo của thiết kế Top-down là **bộ não con người chỉ có thể tập trung vào một lượng chi tiết nhất định** tại mỗi thời điểm. Nếu bạn bắt đầu từ các lớp tổng quát rồi phân rã thành các lớp chuyên biệt hơn từng bước, thì bạn không bị quá tải bởi quá nhiều chi tiết cùng lúc.

Quy trình chia để trị là lặp lại theo hai nghĩa: Thứ nhất, bạn không chỉ dừng lại sau một lần phân rã mà lặp lại ở nhiều cấp độ. Thứ hai, bạn cũng không dừng lại ngay ở phương án đầu tiên mà cần tiếp tục phát triển, đánh giá các cách phân tách khác nhau.

> **Lưu ý về lỗi**: Một số lỗi đánh máy và định dạng đã được chỉnh lại trong quá trình dịch để đảm bảo bản dịch rõ ràng, mạch lạc và chính xác với ngữ cảnh chuyên ngành.

Lựa chọn phương án và phân rã chương trình

Bạn đưa ra một lựa chọn và quan sát kết quả. Sau đó, bạn bắt đầu lại và phân rã theo một cách khác để xem phương án đó có hiệu quả hơn hay không. Sau một số lần thử, bạn sẽ có ý tưởng rõ ràng về phương án nào phù hợp và lý do tại sao.

Bạn nên phân rã chương trình đến mức nào? Hãy tiếp tục phân rã cho đến khi bạn cảm thấy việc lập trình ở cấp tiếp theo dễ dàng hơn là tiếp tục phân rã. Làm việc cho đến khi bạn bắt đầu cảm thấy sốt ruột vì thiết kế có vẻ quá hiển nhiên và đơn giản. Khi đó, bạn đã hoàn thành. Nếu mọi thứ chưa rõ ràng, hãy tiếp tục làm việc. Nếu phương án giải quyết vẫn còn thậm chí hơi phức tạp với bạn ở thời điểm này, thì nó sẽ còn khó hơn nhiều đối với bất kỳ ai làm việc sau này.

Lập luận ủng hộ phương pháp Bottom-Up (Bottom-Up Design)

Đôi khi, phương pháp top-down (từ trên xuống) quá trừu tượng khiến bạn khó bắt đầu. Nếu bạn cần tiếp cận với những khía cạnh cụ thể hơn, hãy thử cách thiết kế bottom-up (từ dưới lên). Tự hỏi bản thân: "Tôi biết hệ thống này cần thực hiện những gì?" Chắc chắn, bạn có thể trả lời câu hỏi đó. Bạn có thể xác định một số trách nhiệm ở mức thấp mà bạn có thể gán cho các class cụ thể.

Ví dụ, bạn có thể biết rằng hệ thống cần định dạng một báo cáo nhất định, tính toán dữ liệu cho báo cáo đó, căn giữa tiêu đề, hiển thị báo cáo trên màn hình, in báo cáo trên máy in, v.v. Sau khi bạn xác định được một số trách nhiệm ở mức thấp, bạn thường sẽ cảm thấy đủ tự tin để quay lại xem xét tổng thể một lần nữa.

Trong một số trường hợp khác, các đặc trưng chính của vấn đề thiết kế được quyết định từ phía dưới. Có thể bạn phải giao tiếp với các thiết bị phần cứng mà yêu cầu giao diện của chúng quyết định phần lớn thiết kế của bạn.

Những điều cần lưu ý khi thực hiện phương pháp bottom-up:

- Tự hỏi bản thân bạn biết rằng hệ thống cần thực hiện những gì.
- Xác định các đối tượng cụ thể và trách nhiệm từ câu hỏi đó.
- Xác định các đối tượng chung và nhóm chúng lại bằng cách tổ chức thành subsystem (hệ thống phụ), package (gói), cấu thành trong đối tượng, hoặc kế thừa (inheritance), tùy từng trường hợp.
- Tiếp tục lên cấp tiếp theo, hoặc quay lại phần tổng quát và thử tiếp cận từ trên xuống.

Thực tế về hai phương pháp

Sự khác biệt chính giữa chiến lược top-down (phân rã) và bottom-up (tổng hợp) là một bên là chiến lược phân tách còn bên kia là chiến lược kết hợp. Một bên bắt đầu từ vấn đề tổng thể và chia nhỏ thành các phần có thể quản lý được; bên còn lại bắt đầu từ các phần nhỏ và xây dựng thành một giải pháp tổng thể. Cả hai phương pháp đều có điểm mạnh và điểm yếu mà bạn cần cân nhắc khi áp dụng vào vấn đề thiết kế.

Ưu điểm của phương pháp top-down

- Dễ thực hiện: Con người có xu hướng giỏi trong việc chia nhỏ một khái niệm lớn thành các thành phần nhỏ hơn, và lập trình viên đặc biệt giỏi trong việc này.
- Có thể trì hoãn các chi tiết triển khai: Vì các hệ thống thường bị thay đổi bởi những chi tiết triển khai (ví dụ, thay đổi cấu trúc file hoặc định dạng báo cáo), nên sẽ hữu ích nếu sớm biết rằng nên ẩn các chi tiết ấy ở các class cấp thấp trong hệ thống phân cấp.

Ưu điểm của phương pháp bottom-up

- Thường giúp xác định sớm các chức năng tiện ích cần thiết, từ đó tạo ra một thiết kế gọn gàng, hợp lý.
- Nếu đã có các hệ thống tương tự, phương pháp bottom-up giúp bạn bắt đầu thiết kế hệ thống mới bằng cách xem xét các thành phần đã có và tự hỏi "Có thể tái sử dụng những gì?".

Nhược điểm của phương pháp bottom-up

- Khó sử dụng độc lập: Phần lớn mọi người làm tốt hơn ở việc chia nhỏ một ý tưởng lớn thành các ý nhỏ, hơn là kết hợp các ý nhỏ lại thành một ý lớn. Điều này giống như việc tự lắp ráp: "Tôi tưởng đã xong, sao hộp vẫn còn nhiều bộ phận vậy?"
- Đôi khi bạn không thể xây dựng một chương trình dựa trên các thành phần đã có sẵn. Bạn không thể xây một chiếc máy bay bằng gạch, và có thể bạn sẽ phải làm việc từ trên xuống trước khi biết mình cần những loại bộ phận gì ở phía dưới.

Tổng kết

- Top-down (từ trên xuống) thường bắt đầu đơn giản, nhưng đôi khi các biến động phức tạp ở cấp thấp sẽ ảnh hưởng trở lại cấp cao, khiến mọi thứ phức tạp hơn mức cần thiết.
- Bottom-up (từ dưới lên) thường bắt đầu phức tạp, nhưng xác định sớm độ phức tạp đó sẽ giúp thiết kế các class cấp cao tốt hơn — nếu như độ phức tạp ấy không làm hỏng toàn bộ hệ thống ngay từ đầu.

Kết luận, hai phương pháp top-down và bottom-up không phải là các chiến lược cạnh tranh – chúng bổ trợ lẫn nhau. Thiết kế là một quá trình heuristic (dựa trên thử nghiệm và kinh nghiệm), nghĩa là không có giải pháp nào đảm bảo thành công tuyệt đối. Thiết kế mang tính thử và sai. Hãy thử nhiều phương pháp cho đến khi bạn tìm ra cái phù hợp nhất.

Prototyping thực nghiệm (Experimental Prototyping)

Đôi khi, bạn không thể biết một thiết kế có khả thi hay không cho tới khi bạn hiểu rõ hơn một số chi tiết triển khai. Có thể bạn sẽ không biết kiểu tổ chức database (cơ sở dữ liệu) nào đó có hiệu quả hay không cho đến khi xác định nó có đáp ứng mục tiêu hiệu năng hay không. Bạn cũng có thể chưa biết một thiết kế subsystem (hệ thống con) có khả thi hay không cho đến khi chọn được các thư viện GUI (Graphical User Interface) cụ thể sẽ sử dụng. Đây là ví dụ về đặc tính khó lường (wickedness) của thiết kế phần mềm — bạn không thể xác định đầy đủ bài toán thiết kế cho đến khi đã giải quyết ít nhất một phần của nó.

Một kỹ thuật chung để giải quyết các câu hỏi này với chi phí thấp là thử nghiệm prototyping (làm nguyên mẫu thử nghiệm). Từ "prototyping" có ý nghĩa khác nhau đối với mỗi người (McConnell 1996). Trong ngữ cảnh này, prototyping nghĩa là viết

một lượng mã tối thiểu không sử dụng lại chỉ để trả lời một câu hỏi thiết kế cụ thể.

Prototyping hoạt động kém hiệu quả khi các lập trình viên không tuân thủ kỷ luật viết lượng mã tối thiểu cần thiết để trả lời một câu hỏi. Giả sử câu hỏi thiết kế là, "Liệu framework cơ sở dữ liệu mà chúng ta chọn có đáp ứng được lưu lượng giao dịch cần thiết không?" Bạn không cần viết mã sản xuất để trả lời câu hỏi đó. Bạn thậm chí không cần biết chi tiết về database. Chỉ cần biết những yếu tố chính như số lượng bảng, số lượng bản ghi trong các bảng, v.v. Từ đó, bạn có thể viết mã thử nghiệm rất đơn giản với các bảng như Table1, Table2 và các cột Column1, Column2, điền dữ liệu ngẫu nhiên và thực hiện kiểm tra hiệu suất.

Prototyping cũng không hiệu quả khi câu hỏi thiết kế không đủ cụ thể. Một câu hỏi như "Liệu framework cơ sở dữ liệu này có hoạt động không?" không đủ định hướng cho việc thử nghiệm. Ngược lại, câu hỏi như "Liệu framework cơ sở dữ liệu này có hỗ trợ 1.000 giao dịch mỗi giây với các giả định X, Y và Z không?" sẽ tạo nền tảng vững chắc hơn cho việc prototyping.

Rủi ro cuối cùng của prototyping là khi lập trình viên không coi mã thử nghiệm là mã sẽ bị loại bỏ. Tôi nhận thấy rằng hầu như không ai có thể viết đúng lượng mã tối thiểu nếu họ tin rằng mã đó sẽ được đưa vào hệ thống sản xuất. Họ sẽ triển khai luôn hệ thống thay vì chỉ thử nghiệm. Khi bạn xác định tư tưởng rằng, sau khi trả lời được câu hỏi, mã sẽ bị loại bỏ, bạn đã giảm thiểu rủi ro này. Một cách để tránh vấn đề đó là tạo prototype trên công nghệ khác với công nghệ sản xuất. Chẳng hạn, bạn có thể thử nghiệm thiết kế Java trong Python hoặc mô phỏng giao diện người dùng bằng Microsoft PowerPoint. Nếu bạn tạo prototype sử dụng chính công nghệ sản xuất, nên quy ước tên class hoặc package cho mã prototype được bắt đầu bằng "prototype", ít nhất điều này cũng giúp lập trình viên cân nhắc kỹ hơn trước khi mở rộng mã thử nghiệm (Stephens 2003).

Sử dụng một cách có kỷ luật, prototyping là công cụ hữu hiệu chống lại sự không xác định của thiết kế. Sử dụng mà thiếu kỷ luật, prototyping lại làm phát sinh những rắc rối mới.

Thiết kế hợp tác (Collaborative Design)

> Tham khảo thêm: Xem Chương 21, "Collaborative Construction" về phát triển hợp tác.

Trong thiết kế, hai cái đầu luôn tốt hơn một, dù được tổ chức chính thức hay không chính thức. Việc hợp tác có thể xuất hiện dưới nhiều hình thức:

- Bạn ra chỗ đồng nghiệp và trao đổi ý tưởng một cách không chính thức.
- Bạn và đồng nghiệp cùng ngồi trong phòng họp, vẽ sơ đồ các phương án thiết kế lên bảng trắng.
- Bạn và đồng nghiệp cùng ngồi trước bàn phím và thực hiện thiết kế chi tiết trong ngôn ngữ lập trình đang sử dụng – nghĩa là có thể dùng pair programming (lập trình cặp), được mô tả ở Chương 21.
- Bạn lên lịch họp để trình bày ý tưởng thiết kế của mình với một hoặc nhiều đồng nghiệp.
- Bạn lên kế hoạch kiểm tra chính thức theo quy trình được mô tả trong Chương 21.
- Nếu không có ai để kiểm tra thiết kế, bạn có thể tự làm một bản thiết kế, cất đi một tuần rồi quay lại kiểm tra. Bạn sẽ quên bớt chi tiết đủ để tự đánh giá chính xác hơn.
- Bạn nhờ ai đó ở ngoài công ty giúp đỡ: gửi câu hỏi lên các diễn đàn hoặc nhóm chuyên môn.

Nếu mục tiêu là đảm bảo chất lượng, tôi thường đề xuất phương pháp kiểm tra chính thức nhất – formal inspection (kiểm tra chính thức) – như đã phân tích ở Chương 21. Tuy nhiên, nếu mục tiêu là thúc đẩy sáng tạo và gia tăng số lượng phương án thiết kế, chứ không chỉ nhằm phát hiện lỗi, thì các phương pháp thiên về sự linh hoạt sẽ hiệu quả hơn. Khi bạn đã chốt được phương án thiết kế, chuyển sang kiểm tra chính thức có thể phù hợp, tùy vào đặc thù dự án.

Bao nhiêu thiết kế là đủ? (How Much Design Is Enough?)

Chúng ta thường cố gắng giải quyết vấn đề bằng cách vội vàng đi qua quá trình thiết kế để dành đủ thời gian ở cuối dự án, nhằm phát hiện các lỗi phát sinh do bỏ qua công đoạn thiết kế.

Đôi khi, chỉ qua vài phác thảo sơ bộ về kiến trúc là đã bắt đầu lập trình. Ở những trường hợp khác, nhóm lại thực hiện thiết kế ở mức chi tiết cao đến mức việc lập trình chỉ đơn thuần là thao tác cơ học. Vậy nên thực hiện thiết kế đến đâu trước khi bắt đầu code?

Một câu hỏi liên quan là mức độ chính thức của bản thiết kế: Bạn có cần các bản thiết kế chi tiết, trau chuốt không?

Ghi chú:

- Đã loại bỏ các ký hiệu đánh máy lỗi trong bản gốc để bản dịch mạch lạc, dễ đọc.
- Các đoạn code không xuất hiện trong đoạn này nên không cần giữ định dạng code.
- Thuật ngữ giữ nguyên và chú thích lần đầu, đảm bảo nhất quán trong toàn văn.

Yếu Tố Ảnh Hưởng Đến Phương Pháp Thiết Kế

Kinh nghiệm của nhóm, tuổi thọ dự kiến của hệ thống, mức độ tin cậy mong muốn, quy mô dự án và quy mô nhóm phát triển đều cần được xem xét. Bảng 5-2 tóm tắt cách từng yếu tố này ảnh hưởng đến phương pháp thiết kế.

Bảng 5-2: Độ Chính Xác Cần Thiết Trong Thiết Kế và Mức Độ Hình Thức

Yếu Tố	Mức Độ Chi Tiết Cần Thiết Trong Thiết Kế Trước Khi Ghi Tài Liệu	Độ Hình Thức Trong Thi Công
Nhóm thiết kế/thi công có kinh nghiệm sâu trong lĩnh vực ứng dụng	Thấp	Thấp
Nhóm có kinh nghiệm thiết kế/thi công nhưng thiếu kinh nghiệm trong lĩnh vực ứng dụng	Vừa	Vừa
Nhóm thiết kế/thi công thiếu kinh nghiệm	Vừa đến Cao	Thấp đến Vừa
Nhóm thiết kế/thi công có mức độ luân chuyển nhân sự vừa đến cao	Vừa	—
Ứng dụng mang tính an toàn (safety-critical)	Cao	Cao
Ứng dụng mang tính nhiệm vụ (mission-critical)	Vừa	Vừa đến Cao
Dự án nhỏ	Thấp	Thấp
Dự án lớn	Vừa	Vừa
Phần mềm dự kiến có tuổi thọ ngắn (vài tuần hoặc vài tháng)	Thấp	Thấp
Phần mềm dự kiến có tuổi thọ dài (vài tháng hoặc nhiều năm)	Vừa	Vừa

Nhiều yếu tố trong số này có thể đồng thời áp dụng cho một dự án cụ thể, và trong một số trường hợp các yếu tố này có thể đưa ra các lời khuyên mâu thuẫn. Ví dụ, bạn có thể có một nhóm giàu kinh nghiệm đang làm việc trên phần mềm an toàn (safety-critical). Trong trường hợp đó, bạn nên ưu tiên mức độ chi tiết và hình thức cao hơn trong thiết kế. Khi đó, bạn cần cân nhắc tầm quan trọng của từng yếu tố và đưa ra quyết định xem yếu tố nào là quan trọng nhất.

Nếu mức độ chi tiết của thiết kế được để cho từng cá nhân quyết định thì, khi thiết kế đi đến cấp độ của một tác vụ mà bạn đã thực hiện trước đó hoặc chỉ là chỉnh sửa, mở rộng đơn giản cho tác vụ đó, bạn có thể dừng thiết kế để bắt đầu lập trình.

5.4 Thực Hành Thiết Kế

Nếu bạn không thể quyết định nên đi sâu vào thiết kế ở mức nào trước khi bắt đầu lập trình, nên thiên về chi tiết nhiều hơn. Những sai lầm lớn nhất trong thiết kế thường phát sinh từ những trường hợp mà bạn nghĩ rằng mình đã đi đủ sâu, nhưng thực tế lại chưa đủ sâu để nhận ra những thách thức bổ sung trong thiết kế. Nói cách khác, các vấn đề lớn trong thiết kế thường không xuất phát từ các khu vực mà bạn biết là khó và đã thiết kế tỉ mỉ cho chúng, mà từ các khu vực tưởng chừng như dễ dàng và bạn không thiết kế cho chúng chút nào. Tôi hiếm khi gặp dự án gặp khó khăn vì làm quá nhiều công việc thiết kế.

> **Tôi chưa từng gặp ai muốn đọc 17.000 trang tài liệu thiết kế, nếu có, tôi sẽ loại người đó ra khỏi quỹ gen.**
> — Joseph Costello

Ngược lại, đôi khi tôi đã thấy các dự án chịu thiệt hại do có quá nhiều tài liệu thiết kế. Định luật Gresham nói rằng "hoạt động lập trình làm giảm đi hoạt động phi lập trình" (Simon 1965). Việc vội vã hoàn thiện tài liệu thiết kế khi chưa cần thiết là một ví dụ điển hình cho định luật này. Tôi thà rằng 80% công sức thiết kế được dùng để tạo ra nhiều phương án thiết kế khác nhau và chỉ 20% cho việc hoàn thiện tài liệu, còn hơn là 20% cho thiết kế và 80% cho làm đẹp tài liệu của các phương án thiết kế tầm thường.

Ghi Lại Công Việc Thiết Kế Của Bạn

Phương pháp truyền thống để lưu lại công việc thiết kế là soạn thảo tài liệu thiết kế hình thức. Tuy nhiên, bạn có thể ghi lại thiết kế bằng nhiều cách khác, phù hợp với các dự án nhỏ, dự án không chính thức, hoặc các dự án cần một phương pháp nhẹ nhàng để lưu vết thiết kế:

- **Chèn tài liệu thiết kế vào chính mã nguồn**
  Ghi lại các quyết định/cốt lõi thiết kế trong phần chú thích (comment) của mã nguồn, thường là ở phần đầu tệp hoặc lớp (class header). Khi sử dụng công cụ trích xuất tài liệu như JavaDoc, tài liệu thiết kế sẽ sẵn có cho lập trình viên làm việc với mã nguồn đó và tăng khả năng tài liệu luôn được cập nhật hợp lý.
- **Ghi lại thảo luận và quyết định thiết kế trên Wiki**
  Tiến hành các cuộc thảo luận bằng văn bản trên Wiki của dự án giúp lưu lại các trao đổi và quyết định thiết kế, mặc dù tốn công nhập liệu thay vì thảo luận miệng. Wiki cũng cho phép đính kèm hình ảnh, liên kết, tài liệu tham khảo hỗ trợ quyết định, rất hữu ích khi nhóm phát triển phân tán địa lý.

- **Viết tóm tắt qua email**
  Sau thảo luận thiết kế, hãy chỉ định một thành viên ghi tóm tắt về buổi thảo luận—đặc biệt là các quyết định được đưa ra—và gửi cho nhóm. Lưu lại email này trong thư mục công cộng của dự án.
- **Sử dụng máy ảnh kỹ thuật số**
  Việc tạo hình vẽ thiết kế bằng các công cụ vẽ chuyên dụng có thể gây nhàm chán. Tuy nhiên, bạn không nhất thiết phải chọn giữa "lưu thiết kế bằng ký hiệu format đẹp, hình thức" và "không có tài liệu thiết kế". Chụp ảnh bảng trắng bằng máy ảnh số rồi nhúng ảnh vào tài liệu có thể mang lại 80% hiệu quả với chỉ 1% công sức so với dùng công cụ vẽ.
- **Lưu các bảng giấy flip chart thiết kế**
  Không có luật nào buộc tài liệu thiết kế phải nằm trên khổ giấy chuẩn. Nếu bạn vẽ thiết kế trên giấy flip chart cỡ lớn, hãy lưu trữ hoặc dán trực tiếp quanh khu vực làm việc để tiện tham khảo và cập nhật.
- **Sử dụng thẻ CRC (Class, Responsibility, Collaborator)**
  Một phương pháp truyền thống khác là dùng thẻ ghi chú (index card). Trên mỗi thẻ, ghi tên lớp (class), trách nhiệm (responsibility) và cộng tác viên (collaborator). Nhóm sẽ cùng thao tác với thẻ cho đến khi thống nhất thiết kế, sau đó lưu lại các thẻ này để tham khảo. Thẻ rẻ tiền, dễ sử dụng, dễ tương tác.
- **Tạo sơ đồ UML (Unified Modeling Language) ở mức độ chi tiết phù hợp**
  UML do Object Management Group định nghĩa, là một kỹ thuật phổ biến để trình bày các khái niệm thiết kế. UML bao gồm nhiều hình thức biểu diễn cấu trúc/phép chiếu các thực thể và mối quan hệ thiết kế. Bạn có thể dùng các phiên bản không chính thức để thử nghiệm phương án thiết kế, bắt đầu với phác thảo đơn giản và bổ sung chi tiết sau khi xác định giải pháp cuối cùng. UML hỗ trợ hiểu biết chung và tăng tốc quá trình cân nhắc lựa chọn thiết kế trong nhóm.

Các kỹ thuật này có thể kết hợp linh hoạt, bạn có thể "mix and match" với từng dự án hoặc từng module trong dự án.

5.5 Bình Luận Về Các Phương Pháp Tiếp Cận Phổ Biến

Lịch sử thiết kế phần mềm luôn có sự xuất hiện của các nhà truyền đạo với những quan điểm cực đoan. Khi tôi xuất bản phiên bản đầu tiên của *Code Complete* vào đầu thập niên 1990, nhiều người khuyến nghị phải làm chi tiết mọi khía cạnh của thiết kế trước khi lập trình, một lời khuyên không thực tế.

> **Những người cổ súy cho thiết kế phần mềm như một hoạt động kỷ luật thường khiến chúng ta cảm thấy tội lỗi vì không đủ cấu trúc hay không đủ hướng đối tượng để đạt được "niết bàn số hóa" mà họ yêu cầu. Nhưng thực ra, phần lớn chúng ta đều là những nhà thiết kế phần mềm tốt hơn nhiều so với họ thừa nhận.**
> —P.J. Plauger

Đến giữa những năm 2000, nhiều chuyên gia lại cho rằng không cần thiết kế gì trước khi lập trình; "Big Design Up Front (BDUF) là xấu; bạn càng ít thiết kế trước càng tốt". Trong vòng mười năm, con lắc đã di chuyển từ "thiết kế mọi thứ" sang "không thiết kế gì cả". Tuy nhiên, giải pháp không phải là BDUF hay không thiết kế gì, mà là *Little Design Up Front* (LDUF) hoặc *Enough Design Up Front* (ENUF).

Làm sao để biết bao nhiêu là đủ? Đây là vấn đề thuộc về phán đoán; không ai có thể biết chính xác mức độ đúng. Tuy nhiên, có hai mức thiết kế luôn luôn sai: làm mọi thứ quá tỉ mỉ hoặc không làm gì cả. Hai cực đoan này đều luôn sai!

Như P.J. Plauger nói:

> **"Càng áp dụng một phương pháp thiết kế một cách cứng nhắc, bạn càng ít giải quyết được các vấn đề thực tế."**
> — P.J. Plauger

Hãy xem thiết kế như một quá trình thử và sai, linh hoạt, bất định. Đừng chỉ chấp nhận phương án đầu tiên xuất hiện trong đầu bạn. Hãy hợp tác, hướng tới sự đơn giản, tạo mẫu thử (prototype) khi cần thiết, lặp lại nhiều lần, và bạn sẽ hài lòng với thiết kế của mình.

Tài Nguyên Tham Khảo Bổ Sung

Lĩnh vực thiết kế phần mềm rất phong phú và đa dạng về tài nguyên; thách thức là xác định đâu là tài nguyên hữu ích nhất cho bạn. Dưới đây là một số gợi ý:

Thiết Kế Phần Mềm, Tổng Quan

- Weisfeld, Matt. *The Object-Oriented Thought Process*, 2nd edition, SAMS, 2004. Đây là một cuốn sách dễ tiếp cận, giới thiệu về lập trình hướng đối tượng.

Các Tài Liệu Tham Khảo Về Thiết Kế Phần Mềm

Sách về Heuristics Thiết Kế Hướng Đối Tượng

- **Riel, Arthur J.** *Object-Oriented Design Heuristics*. Reading, MA: Addison-Wesley, 1996
  Quyển sách này dễ đọc và tập trung vào thiết kế ở cấp độ class.

- **Plauger, P. J.** *Programming on Purpose: Essays on Software Design*. Englewood Cliffs, NJ: PTR Prentice Hall, 1993
  Tôi đã tiếp thu được rất nhiều mẹo về thiết kế phần mềm tốt từ quyển sách này, không thua kém bất kỳ quyển sách nào tôi từng đọc. Plauger có hiểu biết sâu rộng về nhiều tiếp cận thiết kế, mang tính thực dụng và là một tác giả xuất sắc.

- **Meyer, Bertrand.** *Object-Oriented Software Construction*, 2nd ed. New York, NY: Prentice Hall PTR, 1997
  Meyer trình bày một lập luận mạnh mẽ về lập trình hướng đối tượng "cốt lõi".

- **Raymond, Eric S.** *The Art of UNIX Programming*. Boston, MA: Addison-Wesley, 2004
  Đây là một nghiên cứu chuyên sâu về thiết kế phần mềm dưới góc nhìn của UNIX. Mục 1.6 đặc biệt cô đọng khi giải thích trong 12 trang về 17 nguyên lý thiết kế chủ chốt của UNIX.

- **Larman, Craig.** *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 2001
  Quyển này là sách nhập môn phổ biến về thiết kế hướng đối tượng trong bối cảnh Unified Process (Quy trình Thống nhất). Đồng thời, tác phẩm cũng bàn về phân tích hướng đối tượng.

Lý Thuyết Thiết Kế Phần Mềm

- **Parnas, David L., và Paul C. Clements.** "A Rational Design Process: How and Why to Fake It." *IEEE Transactions on Software Engineering* SE-12, no. 2 (February 1986): 251–57
  Bài báo kinh điển này mô tả khoảng cách giữa cách mà chương trình thực sự được thiết kế và cách mà ta ước mong nó được thiết kế. Điểm chính là không ai thực sự thực hiện một quy trình thiết kế hợp lý, có thứ tự, nhưng việc hướng tới mục tiêu đó sẽ mang lại những thiết kế tốt hơn cuối cùng.

  Tôi chưa biết có tài liệu chuyên sâu nào về *information hiding* (ẩn thông tin). Hầu hết các giáo trình kỹ thuật phần mềm chỉ đề cập ngắn gọn vấn đề này, thường trong bối cảnh các kỹ thuật hướng đối tượng. Ba bài báo của Parnas liệt kê dưới đây là những công trình tiên phong về ý tưởng này và có lẽ vẫn là nguồn tài liệu tốt nhất về chủ đề ẩn thông tin:

- **Parnas, David L.** "On the Criteria to Be Used in Decomposing Systems into Modules." *Communications of the ACM* 5, no.12 (December 1972): 1053-58

- **Parnas, David L.** "Designing Software for Ease of Extension and Contraction." *IEEE Transactions on Software Engineering* SE-5, no. 2 (March 1979): 128-38

- **Parnas, David L., Paul C. Clements, và D. M. Weiss.** "The Modular Structure of Complex Systems." *IEEE Transactions on Software Engineering* SE-11, no. 3 (March 1985): 259-66

Mẫu Thiết Kế (*Design Patterns*)

- **Gamma, Erich, et al.** *Design Patterns*. Reading, MA: Addison-Wesley, 1995
  Quyển sách của "Gang of Four" này là tác phẩm nền tảng về mẫu thiết kế.

- **Shalloway, Alan và James R. Trott.** *Design Patterns Explained*. Boston, MA: Addison-Wesley, 2002
  Đây là sách nhập môn dễ hiểu về mẫu thiết kế.

Tài Nguyên Bổ Sung

Thiết Kế Nói Chung

- **Adams, James L.** *Conceptual Blockbusting: A Guide to Better Ideas*, 4th ed. Cambridge, MA: Perseus Publishing, 2001
  Tuy không đề cập cụ thể tới thiết kế phần mềm, quyển sách này được viết để dạy thiết kế cho sinh viên kỹ thuật tại Stanford. Dù bạn không bao giờ thiết kế gì, quyển sách vẫn mang lại một thảo luận hấp dẫn về các quá trình tư duy sáng tạo, kèm nhiều bài tập giúp phát triển tư duy thiết kế hiệu quả. Đồng thời, sách cũng có một thư mục chú thích tốt về thiết kế và tư duy sáng tạo. Nếu bạn thích giải quyết vấn đề, bạn sẽ thích cuốn này.

- **Polya, G.** *How to Solve It: A New Aspect of Mathematical Method*, 2nd ed. Princeton, NJ: Princeton University Press, 1957
  Quyển sách bàn về heuristic (phép thử, phương pháp kinh nghiệm) và giải quyết vấn đề, tập trung vào toán học nhưng vẫn áp dụng được cho phát triển phần mềm. Đây là quyển đầu tiên viết về áp dụng heuristic trong giải toán, phân tích rõ sự khác biệt giữa các heuristic "lộn xộn" để khám phá giải pháp và các kỹ thuật trình bày giải pháp một cách mạch lạc

sau khi khám phá được. Đây không phải sách dễ đọc, nhưng nếu bạn quan tâm tới heuristic, sớm hay muộn bạn cũng sẽ đọc nó. Polya làm rõ rằng giải quyết vấn đề không phải một hoạt động tất định và tuân theo một phương pháp duy nhất chẳng khác gì bị xích chân. Từng có thời gian, Microsoft tặng quyển này cho tất cả lập trình viên mới.

- **Michalewicz, Zbigniew, và David B. Fogel.** *How to Solve It: Modern Heuristics*. Berlin: Springer-Verlag, 2000
  Đây là bản cập nhật, mở rộng của sách Polya, dễ đọc hơn và có cả các ví dụ không thuộc lĩnh vực toán học.

- **Simon, Herbert.** *The Sciences of the Artificial*, 3rd ed. Cambridge, MA: MIT Press, 1996
  Sách này phân biệt các khoa học liên quan đến thế giới tự nhiên (như sinh học, địa chất) và các khoa học về thế giới nhân tạo do con người tạo ra (như kinh doanh, kiến trúc, khoa học máy tính). Tác phẩm thảo luận đặc điểm của khoa học "nhân tạo", nhấn mạnh khoa học thiết kế. Giọng văn học thuật, rất đáng đọc với những ai hướng tới sự nghiệp trong phát triển phần mềm hoặc các lĩnh vực "nhân tạo" khác.

- **Glass, Robert L.** *Software Creativity*. Englewood Cliffs, NJ: Prentice Hall PTR, 1995
  Phát triển phần mềm chịu ảnh hưởng bởi lý thuyết hay thực tiễn nhiều hơn? Đó chủ yếu là sáng tạo hay tất định? Một lập trình viên cần phẩm chất trí tuệ nào? Quyển này bàn luận thú vị về bản chất phát triển phần mềm, nhấn mạnh đặc biệt tới thiết kế.

- **Petroski, Henry.** *Design Paradigms: Case Histories of Error and Judgment in Engineering*. Cambridge: Cambridge University Press, 1994
  Sách chủ yếu dựa vào lĩnh vực kỹ thuật công trình (đặc biệt là thiết kế cầu) để giải thích quan điểm rằng thiết kế thành công phụ thuộc nhiều vào việc học hỏi từ các thất bại trước đây không kém gì thành công quá khứ.

Tiêu Chuẩn

- **IEEE Std 1016-1998**, *Recommended Practice for Software Design Descriptions*
  Tài liệu này chứa tiêu chuẩn IEEE-ANSI về mô tả thiết kế phần mềm, quy định các nội dung cần có trong một tài liệu thiết kế phần mềm.

- **IEEE Std 1471-2000**, *Recommended Practice for Architectural Description of Software Intensive Systems*
  Tài liệu hướng dẫn của IEEE-ANSI về việc xây dựng đặc tả kiến trúc phần mềm.

CHECKLIST: Thiết Kế Trong Giai Đoạn Lập Trình

Thực Hành Thiết Kế

- ❑ Bạn đã lặp lại nhiều lần, lựa chọn phương án tốt nhất trong số nhiều phương án thử thay vì áp dụng phương án đầu tiên?
- ❑ Bạn đã thử phân rã hệ thống theo nhiều cách khác nhau để xác định cách tối ưu?
- ❑ Bạn đã tiếp cận bài toán thiết kế theo cả hai hướng: từ trên xuống (top-down) và từ dưới lên (bottom-up)?
- ❑ Bạn đã tạo prototype cho các phần rủi ro hay chưa quen thuộc, chỉ viết lượng mã cần thiết tối thiểu để trả lời các câu hỏi trọng tâm?
- ❑ Thiết kế của bạn đã được người khác đánh giá, chính thức hoặc phi chính thức?
- ❑ Thiết kế đã được phát triển đến mức việc hiện thực hóa nó trở nên hiển nhiên?
- ❑ Bạn đã ghi lại công việc thiết kế bằng kỹ thuật phù hợp như Wiki, email, bảng ghi, chụp ảnh kỹ thuật số, UML, thẻ CRC, hoặc bình luận trực tiếp trong mã nguồn?

Mục Tiêu Thiết Kế

- ❑ Thiết kế đã xử lý đầy đủ các vấn đề được xác định và hoãn lại ở cấp độ kiến trúc?
- ❑ Thiết kế có được tổ chức thành các lớp (layer) hợp lý?
- ❑ Bạn có hài lòng với cách chương trình được phân chia thành các subsystem (hệ thống con), package (gói) và class (lớp)?
- ❑ Bạn có hài lòng với cách các class được phân chia thành routine (thủ tục, phương thức)?
- ❑ Các class có được thiết kế để giảm thiểu tối đa sự tương tác lẫn nhau không?
- ❑ Các class và subsystem có được thiết kế để có thể tái sử dụng trong những hệ thống khác?
- ❑ Chương trình có dễ bảo trì?
- ❑ Thiết kế có tối giản, tất cả các thành phần đều thực sự cần thiết?
- ❑ Thiết kế có áp dụng các kỹ thuật tiêu chuẩn và tránh các yếu tố kỳ lạ, khó hiểu?
- ❑ Tổng thể, thiết kế có giúp giảm thiểu cả sự phức tạp ngẫu nhiên (accidental complexity) và sự phức tạp bản chất (essential complexity) không?

Những Điểm Chính

- **Yêu cầu kỹ thuật quan trọng nhất của phần mềm là quản lý sự phức tạp.** Việc chú trọng vào sự đơn giản trong thiết kế sẽ giúp đạt được mục tiêu này.
- **Đơn giản hóa được thực hiện theo hai cách:** giảm thiểu lượng phức tạp bản chất mà lập trình viên cần xử lý cùng lúc, và ngăn chặn phức tạp ngẫu nhiên phát sinh không cần thiết.
- **Thiết kế là một quá trình heuristic.** Việc cứng nhắc áp dụng một phương pháp duy nhất sẽ làm giảm tính sáng tạo và chất lượng chương trình.
- **Thiết kế tốt có tính lặp lại;** càng thử nhiều khả năng thiết kế thì thiết kế cuối cùng càng tốt.
- **Information hiding (ẩn thông tin) là một khái niệm đặc biệt quý giá.** Việc hỏi "Nên ẩn thông tin nào?" sẽ giải quyết được nhiều vấn đề thiết kế khó khăn.
- **Có rất nhiều tài liệu hữu ích về thiết kế vượt ngoài khuôn khổ cuốn sách này.** Quan điểm trình bày ở đây chỉ là phần bề nổi của vấn đề.

Chương 6: Làm Việc Với Class

Nội Dung

Chủ đề liên quan

- Thiết kế trong quá trình lập trình: Chương 5
- Kiến trúc phần mềm: Mục 3.5
- Routine chất lượng cao: Chương 7
- Quy trình lập trình bằng Pseudocode: Chương 9
- Refactoring: Chương 24

Dẫn nhập

Vào thời kỳ đầu ngành khoa học máy tính, lập trình viên thường nghĩ về lập trình dưới dạng từng câu lệnh. Trong thập niên 1970 và 1980, cách tiếp cận chuyển sang nghĩ về chương trình dưới dạng các routine (hàm, thủ tục). Sang thế kỷ 21, lập trình viên đã bắt đầu xem việc lập trình là hoạt động liên quan tới các class (lớp).

Một class là tập hợp dữ liệu và các routine có chung một trách nhiệm gắn kết, xác định rõ ràng. Một class cũng có thể là tập hợp các routine cung cấp một nhóm dịch vụ nhất quán, dù không có dữ liệu chung giữa chúng.

Một yếu tố then chốt để trở thành một lập trình viên hiệu quả là:

> **Tối đa hóa phần chương trình mà bạn có thể an tâm bỏ qua khi làm việc trên một đoạn mã cụ thể.**
>
> Class là công cụ chủ yếu để đạt được mục tiêu đó.

Đã kiểm tra bản dịch, đảm bảo tính thuật ngữ nhất quán, rõ nghĩa và liền mạch. Nếu bạn cần dịch thêm các đoạn chi tiết về từng mục, vui lòng cung cấp nội dung cụ thể.

Nếu bạn vẫn đang làm quen với các khái niệm hướng đối tượng

Nếu bạn vẫn đang làm quen với các khái niệm hướng đối tượng (object-oriented), chương này có thể sẽ quá nâng cao. Hãy chắc chắn rằng bạn đã đọc Chương 5, "Thiết kế trong Quá trình Xây dựng" ("Design in Construction"). Sau đó, bắt đầu với Mục 6.1, "Nền tảng của Lớp: Kiểu Dữ liệu Trừu tượng (ADTs - Abstract Data Types)", và làm quen dần với các mục tiếp theo. Nếu bạn đã quen thuộc với những nguyên lý cơ bản về lớp (class), bạn có thể lướt qua Mục 6.1 và chuyển ngay sang thảo luận về giao diện lớp (class interface) ở Mục 6.2. Phần "Tài nguyên Bổ sung" ở cuối chương này cung cấp các tài liệu tham khảo ở cả trình độ cơ bản, nâng cao cũng như các nguồn tài liệu liên quan đến các ngôn ngữ lập trình cụ thể.

6.1 Nền tảng của Lớp: Kiểu Dữ liệu Trừu tượng (ADTs)

Định nghĩa Kiểu Dữ liệu Trừu tượng (ADTs)

Một kiểu dữ liệu trừu tượng (ADT - Abstract Data Type) là một tập hợp dữ liệu cùng các thao tác (operations) thực hiện trên dữ liệu đó. Các thao tác này vừa mô tả dữ liệu cho phần còn lại của chương trình, vừa cho phép các phần khác của chương trình thay đổi dữ liệu này. Từ "dữ liệu" trong cụm "kiểu dữ liệu trừu tượng" được sử dụng một cách linh hoạt. Một ADT có thể là một cửa sổ đồ họa cùng với tất cả các thao tác ảnh hưởng lên nó, một tệp (file) và các thao tác trên tệp, một bảng tỷ lệ bảo hiểm cùng các thao tác xử lý nó, hoặc những đối tượng khác.

> **Liên hệ:** Việc hiểu rõ ADT là điều kiện tiên quyết để nắm bắt lập trình hướng đối tượng (object-oriented programming). Nếu không nhận thức đúng về ADT, lập trình viên có thể tạo ra các lớp chỉ tồn tại dưới tên gọi, thực chất không khác gì những "hộp chứa" tiện lợi cho các tập hợp dữ liệu và hàm liên quan lỏng lẻo (tham khảo Mục 4.3, "Vị trí của bạn trên làn sóng công nghệ", và Mục 34.4, "Lập trình vào ngôn ngữ của bạn, không phải theo nó"). Khi đã hiểu đúng về ADT, bạn sẽ xây dựng được các lớp vừa dễ triển khai ban đầu, vừa dễ chỉnh sửa về sau.

Truyền thống, sách lập trình thường thiên về toán học khi thảo luận về ADT, với những nhận xét như "Có thể xem kiểu dữ liệu trừu tượng là một mô hình toán học cùng với một tập hợp các thao tác định nghĩa trên nó." Kiểu trình bày này khiến bạn tưởng như sẽ không bao giờ dùng ADT, ngoại trừ khi cần ru ngủ!

Các giải thích khô cứng về ADT thường bỏ lỡ ý nghĩa thực tiễn. ADT thực sự hấp dẫn vì nó cho phép bạn thao tác với các thực thể trong thế giới thực, thay vì phải làm việc với các chi tiết triển khai ở mức thấp. Thay vì thao tác với node trong một danh sách liên kết (linked list), bạn có thể thêm một ô vào bảng tính (spreadsheet), thêm kiểu cửa sổ vào danh sách các loại cửa sổ, hoặc bổ sung toa hành khách vào mô phỏng đoàn tàu. Hãy tận dụng sức mạnh làm việc ở miền bài toán, thay vì sa lầy vào các chi tiết triển khai tầm thấp!

Ví dụ về sự cần thiết của ADT

Để làm rõ, hãy xét một trường hợp mà ADT sẽ tỏ ra hữu ích. Chúng ta sẽ đi sâu hơn sau khi có ví dụ cụ thể.

Giả sử bạn đang viết một chương trình điều khiển việc xuất văn bản lên màn hình, với nhiều kiểu chữ (typeface), cỡ chữ (point size) và thuộc tính font (như in đậm - bold, in nghiêng - italic). Phần chương trình này sẽ xử lý font văn bản. Nếu dùng ADT, bạn sẽ có một nhóm các thủ tục thao tác font cùng dữ liệu mà chúng xử lý—cụ thể là danh sách tên kiểu chữ, kích thước, thuộc tính font. Chính tập hợp các thủ tục và dữ liệu liên quan đến font này tạo thành một ADT.

Nếu không sử dụng ADT, bạn sẽ tiếp cận theo kiểu tự phát (ad hoc) khi thao tác font. Ví dụ, để chuyển sang cỡ chữ 12 điểm (point), tức cao 16 pixel, bạn sẽ có đoạn code như sau:

```
currentFont size = 16
```

Nếu bạn đã xây dựng tập các hàm thư viện, mã nguồn có thể dễ hiểu hơn một chút:

```
currentFont size = PointsToPixels( 12 )
```

Hoặc bạn đặt tên thuộc tính rõ ràng hơn:

```
currentFont sizeInPixels = PointsToPixels( 12 )
```

Nhưng bạn không thể đồng thời có cả `currentFont sizeInPixels` và `currentFont sizeInPoints` vì khi cả hai tồn tại, đối tượng `currentFont` sẽ không biết sử dụng thuộc tính nào. Nếu thay đổi kích thước font ở nhiều nơi trong chương trình, bạn sẽ có những dòng code tương tự xuất hiện xuyên suốt toàn bộ chương trình.

Để đặt font sang in đậm, bạn có thể dùng phép toán logic hoặc một hằng số hệ thập lục phân `0x02` như sau:

```
currentFont attribute = currentFont attribute or 0x02
```

Nếu may mắn, bạn có thể có đoạn code "sạch" hơn, tuy nhiên tốt nhất của cách làm ad hoc chỉ là:

```
currentFont attribute = currentFont attribute or BOLD
```

Hoặc thậm chí:

```
currentFont bold = True
```

Giống như với thuộc tính kích thước, hạn chế ở đây là mã nguồn phía client (khách) phải thao tác trực tiếp lên các thành viên dữ liệu, từ đó giới hạn cách sử dụng đối tượng `currentFont`.

Nếu tiếp tục lập trình như vậy, bạn sẽ có các dòng code truy cập trực tiếp này xuất hiện ở nhiều nơi trong chương trình.

Lợi ích của việc sử dụng ADT

Vấn đề không phải là tiếp cận theo ad hoc là kém, mà là bạn có thể thay thế nó bằng phương pháp tốt hơn dẫn đến các lợi ích sau:

- **Ẩn chi tiết triển khai:** Khi ẩn thông tin về kiểu dữ liệu font, nếu cần thay đổi nội dung bên trong, bạn chỉ sửa ở một nơi mà không ảnh hưởng đến toàn bộ chương trình. Ví dụ, nếu không che giấu chi tiết bằng ADT, việc thay đổi cách biểu diễn thuộc tính `bold` sẽ đòi hỏi sửa tất cả mọi nơi sử dụng nó trong chương trình, thay vì chỉ một nơi. Ngoài ra, ẩn thông tin giúp bảo vệ phần còn lại của chương trình nếu bạn chuyển việc lưu trữ dữ liệu sang bộ nhớ ngoài hoặc thay đổi lại toàn bộ thao tác điều khiển font sang ngôn ngữ khác.
- **Thay đổi không ảnh hưởng đến toàn chương trình:** Nếu cần mở rộng kiểu font cho phép nhiều thao tác hơn (chữ in nhỏ, tên chỉ số trên - superscripts, gạch ngang - strikethrough...), bạn sẽ chỉ cần chỉnh sửa một nơi, không ảnh hưởng mã nguồn phần còn lại.
- **Giao diện rõ ràng hơn:** Đoạn code như `currentFont size = 16` khá mơ hồ vì `16` có thể là điểm hoặc pixel. Khi gom các thao tác vào ADT, bạn có thể xác định giao diện chỉ với đơn vị điểm, pixel hoặc phân biệt rõ ràng, giúp tránh nhầm lẫn.
- **Cải thiện hiệu năng dễ dàng:** Khi cần tối ưu thao tác font, bạn chỉ cần viết lại một vài hàm đã xác định, thay vì phải không biết bắt đầu từ đâu trong toàn bộ chương trình.
- **Chương trình chính xác hơn:** Thay vì phải kiểm tra tính đúng đắn của các dòng như `currentFont attribute = currentFont attribute or 0x02`, bạn chỉ cần xác minh những lệnh như `currentFont SetBoldOn()`. Số lỗi tiềm ẩn giảm xuống rõ rệt.
- **Tự động tài liệu hóa mã:** Việc thay thế `0x02` bằng `BOLD` đã giúp dễ đọc, nhưng sẽ không thể sánh với lệnh gọi thủ tục rõ nghĩa như `currentFont SetBoldOn()`.
- **Dễ dàng quản lý dữ liệu:** Không cần phải truyền `currentFont` khắp nơi hoặc chuyển nó thành biến toàn cục (global variable). Dữ liệu sẽ được đóng gói trong ADT và chỉ các thủ tục bên trong ADT mới có quyền truy xuất trực tiếp tới chúng. Các thao tác bên ngoài không cần quan tâm đến cấu trúc dữ liệu sâu bên trong.
- **Làm việc với thực thể thực tế thay vì cấu trúc thấp:** Có thể định nghĩa các thao tác liên quan đến font, giúp phần lớn chương trình làm việc trên các object "font" thực sự, thay vì thao tác trực tiếp với mảng (array), cấu trúc (structure), hay giá trị True/False.

Ví dụ về giao diện ADT cho font:

Bạn có thể định nghĩa một số hàm để xử lý font như sau:

```
currentFont SetSizeInPoints(sizeInPoints)
currentFont SetSizeInPixels(sizeInPixels)
currentFont SetBoldOn()
currentFont SetBoldOff()
currentFont SetItalicOn()
currentFont SetItalicOff()
currentFont SetTypeFace(faceName)
```

Phần mã bên trong các hàm này thường khá ngắn gọn, tương tự như cách làm tự phát (ad hoc) với font ở ví dụ trước. Tuy nhiên, điểm khác biệt là các thao tác với font đã được cô lập trong một tập thủ tục, đem lại mức độ trừu tượng tốt hơn và tăng khả năng bảo trì khi cần thay đổi các thao tác font.

Một số Ví dụ khác về ADT

Giả sử bạn viết phần mềm điều khiển hệ thống làm mát của một lò phản ứng hạt nhân. Bạn có thể coi hệ thống làm mát là một ADT và định nghĩa các thao tác sau:

```
coolingSystem GetTemperature()
coolingSystem SetCirculationRate(rate)
coolingSystem OpenValve(valveNumber)
coolingSystem CloseValve(valveNumber)
```

Môi trường cụ thể sẽ quyết định cách thực hiện từng thao tác này. Phần còn lại của chương trình chỉ cần tương tác với hệ thống làm mát thông qua các hàm này và không cần quan tâm đến chi tiết bên trong của cấu trúc dữ liệu, các hạn chế, hoặc những thay đổi ở mức triển khai nội bộ.

Bạn có thể thấy từ các ví dụ trên rằng, bạn có thể biểu diễn một stack (ngăn xếp), một list (danh sách), một queue (hàng đợi), cũng như hầu như bất kỳ kiểu dữ liệu thông

dụng nào khác, dưới dạng một ADT (Abstract Data Type - Kiểu Dữ liệu Trừu tượng).

Tư duy trừu tượng qua ADT

Điều bạn cần hỏi là: “Stack, list, hoặc queue này đại diện cho điều gì?” Nếu một stack biểu diễn tập hợp các nhân viên, hãy xử lý ADT như là các nhân viên thay vì chỉ là một stack. Nếu một list đại diện cho tập các bảng ghi thanh toán, hãy xem nó là các bảng ghi thanh toán thay vì chỉ là một danh sách. Nếu một queue đại diện cho các ô trong một bảng tính, hãy xem đó là tập hợp các ô, chứ không phải chỉ là một đối tượng chung nằm trong hàng đợi. Hãy tự cho phép mình vận dụng mức trừu tượng cao nhất có thể.

Hãy xem các đối tượng thông thường như file (tập tin) cũng là các ADT. Hầu hết các ngôn ngữ đều hỗ trợ một số kiểu dữ liệu trừu tượng mà bạn có lẽ quen thuộc nhưng có thể chưa nghĩ đến dưới góc độ ADT. Ví dụ điển hình là thao tác với file. Khi ghi dữ liệu ra đĩa, hệ điều hành giúp bạn tránh khỏi việc phải tự định vị đầu đọc/ghi tại một địa chỉ vật lý cụ thể, tự động cấp phát sector đĩa mới khi cần, và xử lý các mã lỗi khó hiểu. Hệ điều hành đã cung cấp cho bạn một tầng trừu tượng đầu tiên cùng với các ADT cho tầng này. Các ngôn ngữ lập trình bậc cao lại cung cấp một tầng trừu tượng thứ hai và các ADT cho tầng cao hơn đó. Một ngôn ngữ bậc cao sẽ "che chắn" cho bạn khỏi các chi tiết phức tạp liên quan đến việc gọi hệ điều hành và thao tác với buffer dữ liệu, cho phép bạn xem một khối không gian trên đĩa như là một “file”.

Bạn cũng có thể xây dựng nhiều tầng ADT tương tự như vậy. Nếu bạn muốn sử dụng một ADT ở một tầng cung cấp các thao tác cấp cấu trúc dữ liệu (như push và pop đối với stack), điều đó hoàn toàn hợp lý. Bạn có thể tạo ra một tầng khác ở phía trên, hoạt động ở mức giải quyết vấn đề thực tế.

Hãy biến cả những phần tử đơn giản thành ADT

Bạn không nhất thiết phải có một kiểu dữ liệu lớn và phức tạp để biện minh cho việc dùng một ADT. Một trong các ví dụ ADT được liệt kê là một bóng đèn chỉ hỗ trợ hai thao tác—bật và tắt. Bạn có thể cho rằng việc tách nhỏ thao tác "bật" và "tắt" thành các routine (thủ tục) riêng là lãng phí, nhưng ngay cả các thao tác đơn giản cũng có thể hưởng lợi từ việc sử dụng ADT. Đưa bóng đèn và các thao tác liên quan vào một ADT giúp code tự tài liệu hóa (self-documenting), dễ dàng thay đổi, cô lập các tác động của thay đổi vào các routine như `TurnLightOn()` và `TurnLightOff()`, đồng thời giảm số lượng biến dữ liệu phải chuyển tiếp.

Tham chiếu ADT mà không phụ thuộc vào phương tiện lưu trữ

Giả sử bạn có một bảng giá bảo hiểm (insurance-rates table) lớn đến mức luôn phải lưu trên đĩa. Bạn có thể sẽ gọi nó là “rate file” và xây dựng các routine truy cập như `RateFileRead()`. Tuy nhiên, tham chiếu nó dưới dạng file sẽ tiết lộ nhiều thông tin về dữ liệu hơn mức cần thiết. Nếu một ngày bạn thay đổi chương trình để bảng được lưu trong bộ nhớ thay vì trên đĩa, những đoạn code gọi nó là file sẽ trở nên sai lệch, gây nhầm lẫn. Hãy cố gắng đặt tên class và các routine truy cập độc lập với cách lưu trữ dữ liệu, và tham chiếu trực tiếp đến ADT, như “bảng giá bảo hiểm”, thay vì “file”. Điều này giúp tên class và routine trở nên như `rateTableRead()` hoặc đơn giản là `ratesRead()`.

Xử lý nhiều instance dữ liệu với ADT trong môi trường không hướng đối tượng

Ngôn ngữ lập trình hướng đối tượng cung cấp sự hỗ trợ tự động cho việc xử lý nhiều instance (thực thể) của một ADT. Nếu bạn chỉ làm việc trong các môi trường hướng đối tượng, và chưa bao giờ phải tự xử lý chi tiết thực hiện nhiều instance, bạn thật may mắn! (Bạn cũng có thể đến thẳng phần tiếp theo về “ADT và Class”).

Nếu bạn làm việc trong môi trường không hướng đối tượng, chẳng hạn như C, bạn sẽ phải tự xây dựng hỗ trợ cho nhiều instance một cách thủ công. Thông thường, điều này có nghĩa là bạn phải bổ sung các service (dịch vụ) cho ADT nhằm tạo và xóa instance, đồng thời thiết kế các dịch vụ còn lại để có thể làm việc với nhiều instance.

Giả sử ADT font ban đầu cung cấp các dịch vụ sau:

```
currentFont SetSize( sizeInPoints )
currentFont SetBoldOn()
currentFont SetBoldOff()
currentFont SetItalicOn()
currentFont SetItalicOff()
currentFont SetTypeFace( faceName )
```

Trong môi trường không hướng đối tượng, các hàm này sẽ không được gắn với class, mà thường trông như sau:

```
SetCurrentFontSize( sizeInPoints )
SetCurrentFontBoldOn()
SetCurrentFontBoldOff()
SetCurrentFontItalicOn()
SetCurrentFontItalicOff()
SetCurrentFontTypeFace( faceName )
```

Nếu bạn cần sử dụng đồng thời nhiều font, bạn sẽ phải bổ sung các service để tạo và xóa instance font, ví dụ:

```
CreateFont( fontId )
DeleteFont( fontId )
SetCurrentFont( fontId )
```

Khái niệm `fontId` được thêm vào để quản lý nhiều font trong quá trình tạo và sử dụng chúng. Đối với các thao tác còn lại, bạn có thể lựa chọn một trong ba cách để xử lý giao diện ADT:

- **Lựa chọn 1:** Xác định rõ instance mỗi lần sử dụng dịch vụ ADT. Trong trường hợp này, không có khái niệm "font hiện tại". Bạn truyền `fontId` cho mỗi routine thao tác lên font. Các hàm liên quan chỉ cần quản lý dữ liệu nội bộ, trong khi code phía client chỉ cần quan tâm tới `fontId`. Điều này yêu cầu bổ sung `fontId` làm tham số cho mỗi routine font.
- **Lựa chọn 2:** Cung cấp rõ ràng dữ liệu được sử dụng bởi các dịch vụ ADT. Ở cách này, bạn khai báo dữ liệu mà ADT sử dụng ngay trong mỗi routine gọi service. Nói cách khác, bạn tạo ra một kiểu dữ liệu Font, truyền nó cho mỗi routine của ADT. Bạn phải thiết kế các hàm ADT để chúng chỉ dùng dữ liệu Font được truyền vào mỗi lần gọi. Code phía client không cần `fontId` vì tự quản lý dữ liệu font. (Dù có thể truy xuất trực tiếp dữ liệu từ kiểu dữ liệu Font, bạn chỉ nên thao tác thông qua các routine của ADT – điều này gọi là giữ cho cấu trúc "đóng", tức đóng kín thông tin).
    - Ưu điểm: Các routine ADT không cần tra cứu dữ liệu font dựa trên `fontId`.
    - Nhược điểm: Dữ liệu font dễ bị lộ ra ngoài chương trình, làm tăng khả năng code phía client sử dụng các chi tiết hiện thực mà lẽ ra phải được giữ kín trong ADT.
- **Lựa chọn 3:** Dùng các instance ngầm định (cần hết sức thận trọng). Thiết kế một dịch vụ mới để thiết lập một instance font cụ thể là "font hiện tại", ví dụ: `SetCurrentFont(fontId)`. Sau đó, mọi dịch vụ khác sẽ thao tác với font hiện tại đó. Nếu dùng cách này, bạn không còn cần truyền `fontId` cho các routine khác. Với ứng dụng đơn giản, cách này giúp toàn bộ thao tác trở nên gọn gàng. Nhưng trong các ứng dụng phức tạp, việc phụ thuộc vào trạng thái instance toàn hệ thống đòi hỏi phải theo dõi instance font hiện tại trong toàn bộ code sử dụng các hàm Font. Sự phức tạp sẽ tăng nhanh và với các ứng dụng lớn, nên chọn giải pháp khác tốt hơn.

Bên trong ADT, bạn có rất nhiều lựa chọn để xử lý nhiều instance, nhưng xét ở bên ngoài, đó là các lựa chọn tổng quát khi bạn làm việc với ngôn ngữ không hướng đối tượng.

ADT và Class

ADT là nền tảng cho khái niệm class. Trong các ngôn ngữ hỗ trợ class, bạn có thể triển khai mỗi ADT thành một class riêng. Class thường được mở rộng với các khái niệm như inheritance (kế thừa) và polymorphism (đa hình). Bạn có thể xem class là ADT cộng thêm inheritance và polymorphism.

6.2 Giao diện class tốt (Good Class Interfaces)

Bước đầu tiên, và có lẽ quan trọng nhất trong việc tạo ra một class chất lượng cao, là thiết kế được giao diện tốt. Điều này bao gồm tạo ra một abstraction (sự trừu tượng) hợp lý cho giao diện cần biểu diễn và đảm bảo các chi tiết hiện thực được che giấu đằng sau abstraction đó.

Abstraction tốt

Như đã trình bày trong phần "Form Consistent Abstractions" (Mục 5.3), abstraction là khả năng nhìn nhận một thao tác phức tạp ở dạng đơn giản hóa. Giao diện class mang lại một abstraction cho phần hiện thực bị che giấu đằng sau giao diện đó. Giao diện class nên cung cấp tập hợp các routine (hàm, phương thức) liên kết nhất quán với nhau.

Ví dụ, bạn có thể có một class biểu diễn nhân viên (employee). Class này sẽ chứa dữ liệu mô tả tên, địa chỉ, số điện thoại nhân viên, v.v. Nó cung cấp các service để khởi tạo và thao tác với nhân viên. Ví dụ:

Ví dụ C++ về một giao diện class thể hiện abstraction tốt

```
class Employee {
public:
    // public constructors and destructors
    Employee();
    Employee(
```

```
        FullName name,
        String address,
        String workPhone,
        String homePhone,
        TaxId taxIdNumber,
        JobClassification jobClass
    );
    virtual ~Employee();

    // public routines
    FullName GetName() const;
    String GetAddress() const;
    String GetWorkPhone() const;
    String GetHomePhone() const;
    TaxId GetTaxIdNumber() const;
    JobClassification GetJobClassification() const;

private:

};
```

Bên trong, class này có thể chứa các routine và dữ liệu khác để hỗ trợ các service trên, nhưng người dùng class không cần biết về chúng. Abstraction trong giao diện class ở đây là tuyệt vời, bởi mỗi routine đều hướng tới một mục tiêu chung nhất quán.

Một class thể hiện abstraction kém

Một class biểu diễn abstraction kém là class chứa tập hợp lộn xộn các hàm với mục đích khác nhau, thiếu sự liên kết nhất quán. Ví dụ:

Ví dụ C++ về một giao diện class thể hiện abstraction kém

```
class Program {
public:
    // public routines
    void InitializeCommandStack();
    void PushCommand( Command command );
    Command PopCommand();
    void ShutdownCommandStack();
    void InitializeReportFormatting();
    void FormatReport( Report report );
    void PrintReport( Report report );
    void InitializeGlobalData();
    void ShutdownGlobalData();

private:

};
```

(Lưu ý: Các lỗi đánh máy trong văn bản gốc đã được biên dịch hợp lý để đảm bảo nội dung dễ hiểu.)

Khó nhận thấy bất kỳ mối liên hệ nào giữa ngăn xếp lệnh (command stack), các thủ tục báo cáo (report routines) hoặc dữ liệu toàn cục

Giao diện của lớp (class interface) không thể hiện một trừu tượng (abstraction) nhất quán, do đó lớp này có **mức độ liên kết (cohesion)** kém. Các thủ tục này nên được tổ chức lại thành những lớp hướng mục đích rõ ràng hơn, trong đó mỗi lớp cung cấp một trừu tượng tốt hơn thông qua giao diện của nó.

Nếu các thủ tục này là một phần của lớp **Program**, chúng có thể được chỉnh sửa để thể hiện một trừu tượng nhất quán, như sau:

Ví dụ C++ về giao diện lớp thể hiện trừu tượng tốt hơn

```
class Program {
public:

    // public routines
    void InitializeUserInterface();
    void ShutDownUserInterface();
    void InitializeReports();
    void ShutDownReports();

private:

};
```

Việc tái cấu trúc giao diện này giả định rằng một số thủ tục gốc đã được chuyển sang các lớp khác phù hợp hơn, và một số đã được chuyển thành các thủ tục private, sử dụng bởi `InitializeUserInterface()` và các thủ tục khác.

Đánh giá trừu tượng của lớp được dựa trên tập hợp các thủ tục public, tức là dựa vào **interface của lớp**. Các thủ tục bên trong lớp không nhất thiết thể hiện trừu tượng tốt riêng lẻ chỉ vì toàn bộ lớp thể hiện trừu tượng tốt; chúng cũng cần được thiết kế để thể hiện trừu tượng tốt. Để biết các nguyên tắc về điều này, tham khảo Mục 7.2, "Thiết kế ở cấp độ thủ tục (routine)".

Việc theo đuổi các giao diện hướng trừu tượng tốt dẫn đến một số nguyên tắc trong quá trình tạo giao diện lớp.

Thể hiện mức độ trừu tượng nhất quán trong giao diện lớp

Một cách hữu ích để hình dung về một lớp chính là cơ chế thực hiện các kiểu dữ liệu trừu tượng (Abstract Data Types - ADT) như mô tả trong Mục 6.1. **Mỗi lớp nên hiện thực duy nhất một ADT.** Nếu bạn nhận thấy một lớp hiện thực nhiều hơn một ADT, hoặc không thể xác định lớp đó đang hiện thực ADT nào, thì đã đến lúc bạn cần tái tổ chức lớp đó thành một hoặc nhiều ADT được định nghĩa rõ ràng.

Dưới đây là một ví dụ về lớp thể hiện giao diện với mức độ trừu tượng không đồng nhất:

Ví dụ C++ về giao diện lớp có mức độ trừu tượng lẫn lộn

```
class EmployeeCensus: public ListContainer {
public:
// public routines

void AddEmployee( Employee employee );
void RemoveEmployee( Employee employee );
// Các trình thủ tục trên thuộc mức trừu tượng "employee"

Employee NextItemInList();
Employee FirstItem();
Employee LastItem();
// Các trình thủ tục này thuộc mức trừu tượng "list"

private:

};
```

Lớp này đang thể hiện hai ADT: một **Employee** và một **ListContainer**. Dạng trừu tượng hỗn hợp này thường xuất hiện khi lập trình viên sử dụng các lớp container (container class) hoặc các lớp thư viện khác cho mục đích hiện thực và **không che giấu được chi tiết** về việc sử dụng lớp thư viện này.

Bạn nên tự hỏi: "Việc sử dụng lớp container có nên là một phần của trừu tượng hay không?" Thông thường, đây là chi tiết triển khai nên được *che giấu khỏi phần còn lại của chương trình*, chẳng hạn như cách làm dưới đây:

Ví dụ C++ về giao diện lớp với mức độ trừu tượng nhất quán

```
class EmployeeCensus {
public:

    // public routines
    void AddEmployee( Employee employee );
    void RemoveEmployee( Employee employee );
    Employee NextEmployee();
    Employee FirstEmployee();
    Employee LastEmployee();

private:
    ListContainer m_EmployeeList;
    // Việc dùng thư viện ListContainer nay đã được ẩn đi

};
```

Nhiều lập trình viên có thể cho rằng việc kế thừa từ **ListContainer** tiện lợi vì hỗ trợ đa hình (polymorphism), cho phép các hàm tìm kiếm hoặc sắp xếp hoạt động với đối tượng ListContainer. Tuy nhiên, lý do đó lại không đáp ứng tiêu chí chính của kế thừa: **"kế thừa chỉ nên áp dụng cho mối quan hệ 'là một' (is a)"**. Nếu kế thừa từ **ListContainer** thì có nghĩa *EmployeeCensus "là một" ListContainer*, điều này rõ ràng là không đúng. Nếu trừu tượng của đối tượng EmployeeCensus là có thể tìm kiếm hoặc sắp xếp, điều đó nên được thể hiện rõ ràng và nhất quán trong giao diện lớp, chứ không nên thông qua kế thừa.

Nếu ví giao diện public của lớp như hệ thống ngăn nước ở cửa tàu ngầm, thì những *thủ tục public không nhất quán* chính là các tấm chắn bị rò rỉ. Các tấm bị rò này có thể không để lọt nước nhanh như một cửa ngăn mở hẳn, nhưng nếu để lâu đủ lâu, chúng sẽ *làm chìm cả tàu*. Trong thực tế, điều này xảy ra khi bạn pha trộn các mức độ trừu tượng. Khi chương trình thay đổi, sự pha trộn phá vỡ cấu trúc chương trình và dần dần trở nên khó bảo trì.

Đảm bảo hiểu đúng trừu tượng mà lớp đang hiện thực

Nhiều lớp có thể đủ tương tự nhau để bạn phải chú ý xác định **trừu tượng đúng mà lớp nên thể hiện**. Từng có trường hợp, một phần mềm *cần cho phép chỉnh sửa thông tin dưới dạng bảng*. Ta định sử dụng một lưới đơn giản (grid control), nhưng các lưới có sẵn không cho phép tô màu ô nhập liệu, nên quyết định dùng một bảng tính (spreadsheet control) vì nó hỗ trợ tính năng này.

Bảng tính này phức tạp hơn rất nhiều so với lưới điều khiển, cung cấp khoảng 150 thủ tục so với 15 của lưới. Mục tiêu là chỉ sử dụng lưới, không phải bảng tính, nên một lập trình viên được giao nhiệm vụ viết **wrapper class** để che giấu việc sử dụng bảng tính. Tuy nhiên, người đó lại đưa ra lớp bọc lộ tất cả 150 thủ tục của bảng tính ra ngoài.

> Đây không phải điều nhóm muốn. Nhóm mong muốn một giao diện dạng lưới, che giấu hoàn toàn việc sử dụng bảng tính phức tạp bên dưới. Lớp bọc chỉ nên công khai 15 thủ tục tương ứng với lưới cộng thêm một thủ tục cho phép tô màu ô. Việc công khai 150 thủ tục chẳng những phá vỡ mục tiêu *encapsulation* mà còn khiến duy trì về sau trở nên khó khăn (nếu muốn thay đổi lớp hiện thực bên dưới). Lập trình viên đã không đạt được trừu tượng hóa và còn tự tạo thêm việc cho mình.

Tùy từng trường hợp cụ thể, trừu tượng đúng có thể là bảng tính hoặc lưới. *Khi phải chọn giữa hai trừu tượng tương đồng, hãy chắc chắn rằng bạn lựa chọn đúng trừu tượng nên thể hiện*.

Cung cấp dịch vụ theo cặp cùng với thao tác ngược lại

Hầu hết các thao tác đều có một thao tác tương ứng đối lập. Nếu bạn có thủ tục *bật* đèn, có lẽ cũng cần thủ tục *tắt* đèn. Thêm phần tử vào danh sách có lẽ cũng cần thủ tục *xóa* phần tử. Kích hoạt một mục thực đơn (menu item) thường đi kèm với thủ tục hủy kích hoạt. Khi thiết kế lớp, kiểm tra từng thủ tục public xem có cần một thao tác đối lập không. Không nên tạo thao tác đối lập một cách tùy tiện, nhưng cần cân nhắc kỹ lưỡng.

Di chuyển thông tin không liên quan sang lớp khác

Trong một số trường hợp, bạn sẽ nhận thấy một nửa số thủ tục của lớp chỉ làm việc với một nửa dữ liệu, và nửa còn lại dùng cho dữ liệu khác. Ở đây thực chất bạn đang có **hai lớp giả làm một**. Hãy tách chúng ra!

Tạo giao diện thiên về chương trình hơn là ý nghĩa ngữ cảnh khi có thể

Mỗi giao diện gồm hai phần: phần chương trình hóa (programmatic) và phần ngữ nghĩa (semantic). Phần programmatic bao gồm kiểu dữ liệu và các thuộc tính mà **trình biên dịch** có thể kiểm tra được. Phần semantic gồm các giả định về cách giao diện sẽ được sử dụng, mà trình biên dịch không thể kiểm tra được (ví dụ: "RoutineA phải được gọi trước RoutineB" hoặc "RoutineA sẽ gây lỗi nếu `dataMember1` chưa được khởi tạo nhưng đã truyền vào RoutineA").

Phần semantic nên được *tài liệu hóa bằng chú thích*, nhưng hãy làm cho giao diện **tối giản sự phụ thuộc vào tài liệu bằng cách chuyển càng nhiều sang kiểm soát của trình biên dịch càng tốt**. Bất kỳ khía cạnh nào không thể kiểm tra bằng trình biên dịch đều dễ bị sử dụng sai. Hãy tìm cách chuyển đối tượng semantic sang phần programmatic, ví dụ qua **Assert** hoặc các kỹ thuật tương tự.

Lưu ý sự thoái hóa trừu tượng giao diện khi bảo trì

Khi một lớp được sửa đổi và mở rộng, thường sẽ phát sinh các chức năng mới mà có vẻ như *không phù hợp hoàn toàn với giao diện lớp*, nhưng lại *quá khó* để hiện thực theo cách khác. Ví dụ, lớp Employee có thể dần phát triển thành:

Ví dụ về giao diện lớp Employee bị thoái hóa khi bảo trì

```
class Employee {
public:

    // public routines
    FullName GetName() const;
    Address GetAddress() const;
    PhoneNumber GetWorkPhone() const;

    bool IsJobClassificationValid( JobClassification jobClass );
    bool IsZipCodeValid( Address address );
    bool IsPhoneNumberValid( PhoneNumber phoneNumber );
    SqlQuery GetQueryToCreateNewEmployee() const;
    SqlQuery GetQueryToModifyEmployee() const;
    SqlQuery GetQueryToRetrieveEmployee() const;
```

```
private:

};
```

> Ban đầu, lớp này có giao diện rất rõ ràng và trong sáng, nhưng nay dần trở nên hỗn tạp với các hàm chỉ liên quan lỏng lẻo với nhau. Không có mối liên hệ logic giữa **nhân viên** với các hàm kiểm tra mã ZIP, số điện thoại, hay bậc nghề nghiệp. Các hàm truy xuất truy vấn SQL (expose SQL query) cũng thuộc mức trừu tượng thấp hơn Employee class, phá vỡ trừu tượng Employee.

Không thêm thành viên public không nhất quán với trừu tượng giao diện

Mỗi khi bổ sung thủ tục mới vào lớp, hãy tự hỏi: *"Thủ tục này có nhất quán với trừu tượng của giao diện hiện tại không?"* Nếu không, hãy tìm cách sửa đổi khác để **duy trì sự toàn vẹn của trừu tượng**.

Lớp với Độ Kết Dính Mạnh và Quan hệ với Trừu Tượng

Các lớp (class) có độ kết dính (cohesion) mạnh thường cung cấp các trừu tượng (abstraction) tốt, mặc dù mối quan hệ này không hoàn toàn tuyệt đối. Qua kinh nghiệm, tập trung vào trừu tượng mà interface của lớp trình bày sẽ mang lại nhiều hiểu biết sâu sắc về thiết kế lớp hơn là chỉ tập trung vào độ kết dính. Nếu bạn nhận thấy một lớp có độ kết dính yếu và không chắc phải khắc phục như thế nào, hãy tự hỏi liệu lớp đó có đang cung cấp một trừu tượng nhất quán hay không.

6.2 Interface Lớp Tốt

Tính Đóng Gói Tốt (Good Encapsulation)

> **Tham khảo chéo:** Như đã được thảo luận trong Mục 5.3, đóng gói (encapsulation) là một khái niệm mạnh mẽ hơn so với trừu tượng (abstraction).
> Trừu tượng giúp quản lý sự phức tạp bằng cách cung cấp các mô hình cho phép bạn bỏ qua các chi tiết triển khai. Đóng gói là cơ chế ngăn chặn bạn truy cập các chi tiết đó, dù bạn có muốn xem chúng đi nữa.

Hai khái niệm này có liên quan chặt chẽ: nếu không có đóng gói, trừu tượng sẽ bị phá vỡ. Trong thực tế, bạn hoặc có cả trừu tượng lẫn đóng gói, hoặc không có gì cả, không có trạng thái trung gian.

> "Yếu tố quan trọng nhất giúp phân biệt một module được thiết kế tốt với module kém chính là mức độ module đó che giấu dữ liệu nội bộ và các chi tiết triển khai khỏi các module khác."
> —Joshua Bloch

Giảm thiểu khả năng truy cập của lớp và thành viên

Giảm thiểu mức độ truy cập là một trong số các quy tắc nhằm thúc đẩy đóng gói. Nếu bạn phân vân liệu một routine nên là `public`, `private` hay `protected`, trường phái thiết kế cho rằng bạn nên ưu tiên cấp độ riêng tư nghiêm ngặt nhất có thể áp dụng (Meyers 1998, Bloch 2001). Tuy nhiên, hướng dẫn quan trọng hơn là:
"Điều gì bảo vệ tốt nhất cho tính toàn vẹn của trừu tượng interface?"
Nếu việc lộ routine là phù hợp với trừu tượng, có thể chấp nhận cho phép truy cập. Nếu không chắc, tốt hơn là ẩn đi nhiều còn hơn là ẩn ít.

Không công khai dữ liệu thành viên qua public

Công khai dữ liệu thành viên (member data) qua interface public vi phạm đóng gói và làm hạn chế sự kiểm soát của bạn đối với trừu tượng. Như Arthur Riel chỉ ra, một lớp Point (điểm) công khai các thành viên dữ liệu sẽ vi phạm đóng gói vì mã của client có thể tùy ý thay đổi dữ liệu Point mà Point không hề biết:

```
float x;
float y;
float z;
```

Tuy nhiên, một lớp Point cung cấp getter và setter lại duy trì đóng gói hoàn chỉnh:

```
float GetX();
float GetY();
float GetZ();
void SetX(float x);
void SetY(float y);
void SetZ(float z);
```

Khi đó, bạn không thể biết liệu lớp Point cài đặt x, y, z dưới dạng float, double, hay lưu trên thiết bị khác...

Tránh đưa chi tiết triển khai riêng tư vào interface của lớp

Với đóng gói thực sự, lập trình viên không thể thấy chi tiết triển khai; chúng phải được ẩn cả về nghĩa đen lẫn nghĩa bóng. Tuy nhiên, trong những ngôn ngữ phổ biến như C++, cấu trúc ngôn ngữ buộc lập trình viên phải công khai một số chi tiết trong phần khai báo class:

```
class Employee {
public:
    Employee(
        FullName name,
        String address,
        String workPhone,
        String homePhone,
        TaxId taxIdNumber,
        JobClassification jobClass
    );

    FullName GetName() const;
    String GetAddress() const;

private:
    // Các chi tiết triển khai bị lộ ra
    String m_Name;
    String m_Address;
    int m_jobClass;
};
```

Khai báo private trong file header dường như là lỗi nhẹ, nhưng lại khuyến khích lập trình viên khác "soi" chi tiết triển khai. Ví dụ trên, kiểu Address được đề nghị dùng trong interface, nhưng lại lộ ra là String trong triển khai thực tế.

Scott Meyers đã đưa ra một giải pháp phổ biến (Effective C++, Item 34, Meyers 1998): Tách interface khỏi implementation bằng cách sử dụng con trỏ tới implementation:

```
class Employee {
public:
    Employee();

    FullName GetName() const;
    String GetAddress() const;

private:
    EmployeeImplementation *m_implementation; // Các chi tiết triển khai được ẩn phía sau con trỏ này
};
```

Giờ đây, các chi tiết triển khai chỉ nằm trong class EmployeeImplementation vốn chỉ được thấy bởi lớp Employee, không phải bởi mã client sử dụng Employee.

Nếu dự án của bạn đã có rất nhiều code không áp dụng mô hình này, có thể sẽ không đáng để chuyển đổi toàn bộ. Nhưng khi đọc code mà lộ chi tiết triển khai, hãy cưỡng lại thói quen xem phần private để săn lùng thông tin triển khai.

Không Giả Định Cách Dùng của Người Dùng Lớp

Một class nên được thiết kế tuân theo hợp đồng (contract) nêu trong interface của nó. Không nên giả định về cách client sẽ sử dụng interface ấy (ngoài những gì đã được tài liệu hóa). Ví dụ, comment như sau là dấu hiệu class biết quá nhiều về client:

> -- Khởi tạo x, y, z thành 1.0 vì DerivedClass sẽ lỗi nếu chúng được khởi tạo là 0.0

Tránh Friend Class (Lớp bạn)

Trong một vài trường hợp đặc biệt, như mẫu thiết kế State pattern, friend class có thể sử dụng nghiêm túc để kiểm soát phức tạp (Gamma et al., 1995). Tuy nhiên, nhìn chung, friend class vi phạm đóng gói và khiến bạn phải suy nghĩ về nhiều đoạn code cùng lúc, gia tăng độ phức tạp.

Đừng Đưa Routine vào Interface Chỉ vì Nó Sử Dụng Toàn Routine Public

Việc một routine chỉ sử dụng các routine public không phải là lý do để công khai nó. Thay vào đó, hãy tự hỏi việc expose routine đó có phù hợp với trừu tượng mà interface trình bày hay không.

Ưu Tiên Tiện Lợi Khi Đọc Trên Tiện Lợi Khi Viết

Code sẽ được đọc nhiều hơn viết, kể cả trong giai đoạn phát triển ban đầu. Ưu tiên giải pháp mang lại sự thuận tiện khi đọc, dù có thể làm chậm khi viết, luôn là lựa chọn kinh tế hợp lý. Đôi khi, bạn sẽ muốn thêm routine vào interface để tiện cho client hiện thời, nhưng đây là bước mở màn cho những vấn đề về lâu dài – tốt nhất đừng đi bước đầu tiên đó.

> "Không thực sự là trừu tượng nếu bạn phải xem chi tiết triển khai mới hiểu được chuyện gì đang diễn ra."
> —P.J. Plauger

Đóng Gói Ngữ Nghĩa (Semantic Encapsulation)

Việc đảm bảo đóng gói về mặt ngữ nghĩa khó hơn nhiều so với đóng gói về mặt cú pháp. Cú pháp, bạn chỉ cần khai báo dữ liệu và routine nội bộ là private. Nhưng để thực sự đóng gói về ngữ nghĩa, bạn phải đảm bảo client không dựa vào chi tiết triển khai bên trong class. Một số ví dụ phá vỡ đóng gói ngữ nghĩa:

- Không gọi routine `InitializeOperations()` của Class A vì biết rằng routine `PerformFirstOperation()` sẽ tự động gọi nó.
- Không gọi routine `Connect()` tới database trước khi gọi `employee.Retrieve(database)` vì biết rằng routine này sẽ tự động tạo kết nối nếu chưa có.
- Không gọi routine `Terminate()` của Class A vì biết rằng routine `PerformFinalOperation()` đã gọi rồi.
- Tiếp tục dùng một con trỏ/reference tới `ObjectB` do `ObjectA` tạo ra, kể cả sau khi `ObjectA` ra khỏi phạm vi vì biết rằng `ObjectA` lưu trữ ObjectB ở static storage.
- Sử dụng hằng số `MAXIMUM_ELEMENTS` của Class B thay vì Class A vì biết hai giá trị này bằng nhau.

Vấn đề là, các ví dụ trên làm code client phụ thuộc vào chi tiết triển khai private thay vì interface public. Nếu bạn phải xem code bên trong để hiểu cách dùng class, bạn không chỉ lập trình dựa trên interface mà còn "lập trình xuyên qua interface vào chi tiết triển khai". Khi đóng gói bị phá vỡ, trừu tượng cũng sẽ sụp đổ.

Nếu bạn không thể sử dụng một class chỉ dựa vào tài liệu interface của nó, đáp án đúng không phải là đọc code nguồn để mò cách dùng. Đó có thể là sự chủ động, nhưng là lựa chọn sai lầm: thay vào đó hãy liên hệ tác giả của class và nói "Tôi không biết dùng class này thế nào". Tác giả class nên sửa tài liệu interface để bạn – và các lập trình viên khác trong tương lai – có thể dễ dàng hiểu cách sử dụng, thay vì trả lời bạn trực tiếp hoặc để ghi chú lưu lại trong trí nhớ bạn mà biến thành những phụ thuộc tinh vi vào chi tiết triển khai của class trong mã client.

Cẩn trọng với Coupling (Mức liên kết) quá chặt

Coupling (mức liên kết) đề cập đến mức độ gắn kết giữa hai *class* (lớp). Về tổng thể, liên kết càng lỏng thì càng tốt. Từ khái niệm này có thể rút ra một số nguyên tắc chung:

- **Giảm thiểu khả năng truy cập của các lớp và thành viên**
- **Tránh sử dụng friend class**, vì chúng tạo ra sự liên kết chặt chẽ
- **Khai báo dữ liệu trong lớp cơ sở là private thay vì protected** để các lớp dẫn xuất ít bị ràng buộc chặt chẽ với lớp cơ sở
- **Tránh để lộ dữ liệu thành viên trên giao diện public của lớp**
- **Thận trọng với các vi phạm ngữ nghĩa trong đóng gói (encapsulation)**
- **Tuân thủ "Law of Demeter" (Luật Demeter, sẽ thảo luận ở Mục 6.3 của chương này)**

Liên kết (coupling) gắn liền với khái niệm trừu tượng (abstraction) và đóng gói (encapsulation). Sự liên kết chặt thường xảy ra khi lớp trừu tượng không đủ chặt (abstraction bị rò rỉ) hoặc khi đóng gói bị phá vỡ. Nếu một lớp không cung cấp đầy đủ các dịch vụ cần thiết, các hàm khác có thể trực tiếp truy cập hoặc sửa đổi dữ liệu nội bộ của nó. Điều này biến lớp đó thành một "glass box" (hộp kính), thay vì "black box" (hộp đen), và gần như loại bỏ khả năng đóng gói của lớp.

6.3 Các vấn đề về Thiết kế và Triển khai

Việc định nghĩa các giao diện lớp tốt góp phần trọng yếu vào việc xây dựng chương trình có chất lượng cao. Thiết kế và triển khai nội tại của lớp cũng đóng vai trò quan trọng. Phần này sẽ thảo luận về các vấn đề liên quan đến *containment* (bao hàm), *inheritance* (kế thừa), hàm thành viên và dữ liệu, *class coupling* (mức liên kết giữa các lớp), *constructor* (hàm khởi tạo), và đối tượng giá trị so với đối tượng tham chiếu.

Containment – Quan hệ "has a" (Có)

Containment (bao hàm) là khái niệm đơn giản rằng một lớp chứa một phần tử dữ liệu nguyên thủy hoặc một đối tượng khác. Đã có nhiều tài liệu viết về kế thừa hơn về bao hàm, không phải vì kế thừa tốt hơn mà bởi vì kế thừa phức tạp hơn, dễ xảy ra lỗi hơn. Bao hàm chính là kỹ thuật làm việc hiệu quả trong lập trình hướng đối tượng.

Thực hiện "has a" thông qua containment
Một cách để hình dung containment là một quan hệ “has a”. Ví dụ: một nhân viên “has a” (có) tên, “has a” số điện thoại, “has a” mã số thuế, ... Bạn thường có thể đạt được điều này bằng cách khai báo các trường tên, số điện thoại, mã số thuế làm dữ liệu thành viên của lớp Employee.

Chỉ sử dụng kế thừa riêng tư để thực hiện "has a" khi thật sự cần thiết
Trong một số trường hợp, bạn có thể thấy rằng không thể đạt được bao hàm chỉ bằng cách tạo một đối tượng là thành viên của lớp khác. Khi đó, một số chuyên gia đề xuất sử dụng kế thừa riêng tư (private inheritance) từ lớp bị bao hàm (Meyers 1998, Sutter 2000). Lý do chính là để lớp chứa có thể truy cập các hàm thành viên hoặc dữ liệu thành viên được khai báo là protected trong lớp bị bao hàm. Tuy nhiên, trên thực tế, cách làm này thường dẫn đến mô hình liên kết quá chặt với lớp tổ tiên và vi phạm đóng gói. Điều này thường chỉ ra các sai sót trong thiết kế mà nên được giải quyết bằng các cách khác thay vì dùng kế thừa riêng tư.

Xem xét cẩn thận các lớp có hơn bảy trường dữ liệu thành viên
Con số "7±2" đã được xác định là số lượng đối tượng rời rạc mà một người có thể ghi nhớ khi thực hiện các nhiệm vụ khác (Miller 1956). Nếu một lớp chứa quá bảy trường dữ liệu thành viên, hãy cân nhắc việc tách lớp đó thành nhiều lớp nhỏ hơn (Riel 1996). Nếu các dữ liệu thành viên là các kiểu dữ liệu nguyên thủy như số nguyên hoặc chuỗi, bạn có thể nghiêng về phía cao của 7±2; nếu là các đối tượng phức tạp, thì nên nghiêng về phía thấp hơn.

Inheritance – Quan hệ “is a” (Là một)

Kế thừa là ý tưởng rằng một lớp là một chuyên biệt hóa của một lớp khác. Mục đích của kế thừa là tạo ra mã đơn giản hơn bằng cách định nghĩa một lớp cơ sở, xác định các thành phần chung của hai hay nhiều lớp dẫn xuất. Các thành phần chung có thể là giao diện hàm (routine interfaces), cài đặt, dữ liệu thành viên hoặc kiểu dữ liệu. Kế thừa giúp tránh phải lặp lại mã và dữ liệu ở nhiều nơi, bằng cách tập trung chúng trong một lớp cơ sở.

Khi quyết định sử dụng kế thừa, bạn phải đưa ra một số lựa chọn:

- Với mỗi hàm thành viên, liệu hàm đó có được nhìn thấy bởi các lớp dẫn xuất không? Liệu nó có cài đặt mặc định không? Liệu việc cài đặt mặc định đó có thể bị ghi đè (*override*) không?
- Đối với mỗi trường dữ liệu (bao gồm biến, hằng số định danh, kiểu enum, ...), liệu trường dữ liệu đó có được các lớp dẫn xuất nhìn thấy không?

Các mục sau sẽ giải thích chi tiết việc đưa ra các quyết định này.

Nguyên tắc quan trọng nhất: Thực hiện “is a” qua kế thừa công khai (public inheritance)

> Quy tắc quan trọng nhất trong lập trình hướng đối tượng với C++ là:
> **Kế thừa công khai (public inheritance) nghĩa là "is a".**
> Hãy ghi nhớ quy tắc này.
> — Scott Meyers

Khi một lập trình viên tạo một lớp mới bằng cách kế thừa từ lớp đã có, đồng nghĩa với việc khẳng định lớp mới “là một” phiên bản chuyên biệt hóa của lớp cũ. Lớp cơ sở đặt ra các kỳ vọng về cách hoạt động của lớp dẫn xuất, đồng thời thiết lập những ràng buộc mà lớp dẫn xuất phải tuân theo (Meyers 1998).

Nếu lớp dẫn xuất không hoàn toàn tuân thủ cách giao diện do lớp cơ sở quy định, thì kế thừa không phải là kỹ thuật phù hợp. Hãy cân nhắc sử dụng containment hoặc sửa đổi thiết kế ở tầng lớp cao hơn trong hệ thống kế thừa.

Thiết kế và tài liệu hóa cho kế thừa, hoặc cấm kế thừa

Kế thừa làm gia tăng độ phức tạp cho chương trình và vì thế, đây là một kỹ thuật dễ gây rủi ro. Như Java guru Joshua Bloch đã nói:

> "Thiết kế và tài liệu hóa cho việc kế thừa, hoặc cấm kế thừa."
> Nếu một lớp không hướng tới việc được kế thừa, hãy đặt các thành viên của nó là non-virtual trong C++, final trong Java, hoặc non-overridable trong Microsoft Visual Basic để đảm bảo không ai có thể kế thừa từ nó cả.

Tuân thủ Nguyên lý Thay thế của Liskov (Liskov Substitution Principle - LSP)

Trong một trong những bài báo nền tảng về lập trình hướng đối tượng, Barbara Liskov cho rằng không nên kế thừa từ lớp cơ sở trừ khi lớp dẫn xuất “là một” phiên bản chuyên biệt hóa thực sự của lớp cơ sở đó (Liskov 1988). Andy Hunt và Dave Thomas tổng kết nguyên lý LSP như sau:

> "Các subclass phải có khả năng được sử dụng thông qua giao diện của lớp cơ sở mà không cần người dùng phải biết sự khác biệt." (Hunt và Thomas 2000)

Nói cách khác, tất cả các hàm được định nghĩa trong lớp cơ sở phải mang ý nghĩa giống nhau khi chúng được sử dụng trong từng lớp dẫn xuất.

Ví dụ, nếu bạn có một lớp cơ sở *Account* và các lớp dẫn xuất *CheckingAccount*, *SavingsAccount*, và *AutoLoanAccount*, thì lập trình viên phải có khả năng gọi bất kỳ hàm nào được định nghĩa trong *Account* trên bất kỳ đối tượng nào của các lớp này mà không cần quan tâm đối tượng thuộc loại nào.

Nếu chương trình được viết tuân thủ LSP, kế thừa sẽ là một công cụ mạnh để giảm độ phức tạp vì lập trình viên có thể tập trung vào các thuộc tính tổng quát thay vì chi tiết triển khai. Nếu lập trình viên luôn phải quan tâm đến các khác biệt ngữ nghĩa khi sử dụng các lớp dẫn xuất, thì kế thừa lại làm tăng độ phức tạp.
Ví dụ:

> "Nếu tôi gọi hàm InterestRate() trên CheckingAccount hoặc SavingsAccount thì nó trả về lãi suất ngân hàng trả cho khách hàng, nhưng nếu gọi InterestRate() trên AutoLoanAccount, tôi phải đổi dấu vì nó trả về lãi suất khách hàng phải trả cho ngân hàng."
> Theo nguyên lý LSP, trong trường hợp này *AutoLoanAccount* không nên kế thừa từ *Account* vì ý nghĩa của hàm *InterestRate()* không còn đồng nhất với lớp cơ sở.

Chỉ kế thừa những gì cần kế thừa

Một lớp dẫn xuất có thể kế thừa giao diện hàm thành viên, hiện thực, hoặc cả hai. Bảng dưới đây liệt kê các biến thể của việc kế thừa hàm:

	Overridable	**Not Overridable**
Implementation Provided	Default Overridable	Non-Overridable
Implementation Not Provided	Abstract Overridable	Không sử dụng (không hợp lý để để trống mà lại không cho phép override)

Theo bảng trên, các hàm kế thừa có ba kiểu cơ bản:

- **Hàm trừu tượng có thể ghi đè (abstract overridable routine):** Lớp dẫn xuất kế thừa giao diện, không kế thừa hiện thực.
- **Hàm có thể ghi đè (overridable routine):** Lớp dẫn xuất kế thừa cả giao diện lẫn hiện thực mặc định, cho phép ghi đè lại hiện thực.
- **Hàm không cho phép ghi đè (non-overridable routine):** Lớp dẫn xuất kế thừa cả giao diện và hiện thực mặc định, nhưng không được phép ghi đè.

Khi quyết định tạo lớp mới bằng kế thừa, hãy cân nhắc kỹ thuật kế thừa phù hợp cho từng hàm thành viên. **Cảnh giác với việc kế thừa hiện thực chỉ vì cần giao diện, hoặc kế thừa giao diện chỉ vì cần hiện thực.** Nếu bạn chỉ muốn sử dụng cài đặt của một lớp chứ không muốn giao diện của nó, hãy dùng containment thay vì inheritance.

Không "override" hàm thành viên không cho phép ghi đè

Cả C++ lẫn Java đều cho phép lập trình viên tạo một hàm trong lớp dẫn xuất trùng tên với một hàm không cho phép ghi đè ở lớp cơ sở (chẳng hạn hàm private). Đối với người đọc mã ở lớp dẫn xuất, điều này rất dễ gây hiểu nhầm vì tưởng rằng đây là đa hình (polymorphic), nhưng thực ra chỉ là các hàm trùng tên. Nguyên tắc ở đây là: **Không sử dụng lại tên các hàm không cho phép ghi đè của lớp cơ sở trong các lớp dẫn xuất.**

Đưa các giao diện, dữ liệu và hành vi chung lên càng cao càng tốt trong cây kế thừa

Càng đưa các giao diện, dữ liệu và hành vi dùng chung lên cao trong cây kế thừa, các lớp dẫn xuất càng dễ sử dụng chúng. Tuy nhiên, không nên đưa lên quá cao khiến phá vỡ trừu tượng của lớp mức cao hơn. Hãy để trừu tượng làm kim chỉ nam: nếu việc đưa một hàm lên cao hơn làm phá vỡ trừu tượng của lớp đó, thì không nên thực hiện.

Cẩn trọng với các lớp chỉ có một thể hiện duy nhất

Việc chỉ có một thể hiện (instance) của một lớp có thể chỉ ra rằng thiết kế đang nhầm lẫn giữa đối tượng và lớp. Hãy cân nhắc xem có thể chỉ tạo ra một đối tượng thay vì tạo một lớp mới hay không.

Ghi chú: Các lỗi đánh máy nhỏ, lỗi xuống dòng trong nguyên bản đã được bỏ qua để giữ cho bản dịch mạch lạc và dễ đọc.

Cảnh Giác Với Lớp Cơ Sở Chỉ Có Một Lớp Kế Thừa

Khi gặp một lớp cơ sở (base class) chỉ có một lớp dẫn xuất (derived class) duy nhất, tôi thường nghi ngờ rằng lập trình viên đã "thiết kế trước"—cố gắng dự đoán các nhu cầu tương lai mà thường không thực sự hiểu rõ những nhu cầu đó. Cách tốt nhất để chuẩn bị cho công việc trong tương lai không phải là thiết kế thêm các lớp cơ sở mà "có thể sẽ cần về sau", mà là làm cho công việc hiện tại trở nên rõ ràng, trực quan và đơn giản nhất có thể. Điều này đồng nghĩa với việc không tạo ra bất kỳ cấu trúc kế thừa (inheritance structure) nào nhiều hơn mức thực sự cần thiết.

Cảnh Báo Khi Lớp Dẫn Xuất Ghi Đè Một Phương Thức Mà Không Làm Gì

Hãy nghi ngờ những lớp ghi đè (override) một phương thức nhưng lại không thực hiện bất kỳ xử lý nào trong phương thức dẫn xuất đó. Điều này thường chỉ ra một sai lầm trong thiết kế của lớp cơ sở. Ví dụ, giả sử bạn có một lớp Cat (Mèo) và một phương thức Scratch(), và sau này bạn phát hiện ra rằng một số con mèo bị cắt móng nên không thể cào. Bạn có thể bị cám dỗ tạo ra một lớp dẫn xuất từ Cat với tên ScratchlessCat và ghi đè phương thức Scratch() để không làm gì cả. Phương pháp này dẫn tới nhiều vấn đề:

- **Vi phạm trừu tượng hóa (abstraction) hoặc hợp đồng giao diện (interface contract)** được định nghĩa trong lớp Cat, bởi nó thay đổi ngữ nghĩa của giao diện.
- Phương pháp này sẽ nhanh chóng mất kiểm soát khi mở rộng sang các lớp dẫn xuất khác. Điều gì xảy ra nếu phát hiện ra một con mèo không có đuôi? Hoặc một con mèo không bắt chuột? Hoặc một con mèo không uống sữa? Cuối cùng bạn sẽ có những lớp dẫn xuất như ScratchlessTaillessMicelessMilklessCat.
- Theo thời gian, phương pháp này tạo ra mã nguồn khó bảo trì vì giao diện và hành vi của các lớp tổ tiên gần như không nói lên điều gì về hành vi của các lớp con.

  Nơi cần sửa là ở lớp Cat gốc, không phải ở lớp cơ sở hay các lớp con. Hãy tạo một lớp Claws (Móng) và đưa nó vào bên trong lớp Cat. Vấn đề cốt lõi là giả định rằng tất cả mèo đều có thể cào, vì vậy hãy sửa nó ngay từ nguồn thay vì chỉ "vá lỗi" ở đích đến.

Tránh Các Cây Kế Thừa Sâu

Lập trình hướng đối tượng cung cấp nhiều kỹ thuật để quản lý độ phức tạp. Tuy nhiên, bất kỳ công cụ mạnh mẽ nào cũng tiềm ẩn rủi ro và một số kỹ thuật hướng đối tượng có xu hướng làm tăng độ phức tạp thay vì giảm bớt.

Trong cuốn sách *Object-Oriented Design Heuristics* (1996), Arthur Riel đề xuất giới hạn cây kế thừa ở mức tối đa sáu cấp. Riel dựa trên "con số ma thuật 7±2", nhưng tôi cho rằng điều này khá lạc quan. Theo kinh nghiệm của tôi, hầu hết mọi người đều gặp khó khăn khi phải xử lý trên hai hoặc ba cấp kế thừa cùng lúc. "Con số ma thuật 7±2" có lẽ phù hợp hơn với tổng số lớp con trực tiếp của một lớp cơ sở, thay vì số cấp trong cây kế thừa.

Các nghiên cứu đã chỉ ra rằng những cây kế thừa sâu có liên quan đáng kể đến tỷ lệ lỗi tăng cao (Basili, Briand, and Melo 1996). Bất kỳ ai từng phải gỡ lỗi một hệ thống kế thừa phức tạp đều hiểu lý do. Cây kế thừa sâu làm tăng độ phức tạp, điều đối lập với mục tiêu của kế thừa. Luôn ghi nhớ mục tiêu kỹ thuật chính: sử dụng kế thừa để tránh lặp lại mã và giảm thiểu độ phức tạp.

Ưu Tiên Đa Hình Hơn Là Kiểm Tra Kiểu Đa Dạng

Các câu lệnh kiểm tra kiểu (type checking) lặp lại thường xuyên có thể gợi ý rằng việc sử dụng kế thừa sẽ là một lựa chọn thiết kế tốt hơn, mặc dù điều này không phải lúc nào cũng đúng.

Ví dụ kinh điển dưới đây minh họa cho một tình huống nên áp dụng tiếp cận hướng đối tượng hơn:

```
// Ví dụ C++ về câu lệnh switch nên được thay thế bằng đa hình
switch (shape_type) {
    case Shape_Circle:
        shape.DrawCircle();
        break;
    case Shape_Square:
        shape.DrawSquare();
        break;
}
```

Trong ví dụ này, các lời gọi `shape.DrawCircle()` và `shape.DrawSquare()` nên được thay thế bằng duy nhất một phương thức có tên `shape.Draw()`, mà có thể được gọi cho mọi hình dạng (shape), bất kể là hình tròn hay hình vuông.

Tuy nhiên, đôi khi câu lệnh switch thực sự phù hợp để phân biệt các đối tượng hoặc hành vi khác biệt hoàn toàn. Ví dụ bên dưới là một trường hợp sử dụng switch hợp lý:

```
// Ví dụ C++ về câu lệnh switch không nên thay bằng đa hình
switch (ui.Command()) {
    case Command_OpenFile:
        OpenFile();
        break;
    case Command_Print:
        Print();
        break;
    case Command_Save:
        Save();
        break;
    case Command_Exit:
        ShutDown();
        break;
}
```

Trong trường hợp này, có thể xây dựng một lớp cơ sở với các lớp dẫn xuất và phương thức đa hình DoCommand() cho mỗi lệnh (theo mẫu thiết kế Command pattern). Tuy nhiên, với trường hợp đơn giản như trên, ý nghĩa của DoCommand() sẽ trở nên mơ hồ, và sử dụng switch là giải pháp dễ hiểu hơn.

Hãy Đặt Mọi Dữ Liệu Là Private, Không Phải Protected

Như Joshua Bloch đã nói, "Kế thừa phá vỡ tính đóng gói (encapsulation)" (2001). Khi kế thừa từ một đối tượng, bạn nhận được quyền truy cập đặc biệt vào các phương thức và dữ liệu được bảo vệ (protected). Nếu lớp dẫn xuất thực sự cần truy cập các thuộc tính của lớp cơ sở, hãy cung cấp các hàm truy cập bảo vệ (protected accessor functions) thay vì để dữ liệu ở dạng protected.

Kế Thừa Đa Lần (Multiple Inheritance)

Nguy Cơ Của Kế Thừa Đa Lần

> "Nếu kế thừa là một cưa máy thì kế thừa đa lần là chiếc cưa máy thời kỳ 1950, không có bảo vệ lưỡi, không tự ngắt và động cơ khó dùng. Đôi khi công cụ này hữu dụng; nhưng hầu hết nên để nguyên trong gara cho an toàn."
> —Scott Meyers

Kế thừa là một công cụ mạnh mẽ. Nó như việc dùng cưa máy để đốn cây thay vì dùng cưa tay. Kế thừa đa lần mang đến một "hộp Pandora" về sự phức tạp không hề xuất hiện khi chỉ có kế thừa đơn.

Một số chuyên gia đề xuất sử dụng rộng rãi kế thừa đa lần (Meyer 1997), nhưng thực tế kinh nghiệm của tôi cho thấy, nó chủ yếu hữu dụng khi định nghĩa các "mixin"—các lớp đơn giản bổ sung một tập thuộc tính cho đối tượng. Mixin là cách "trộn" thuộc tính vào các lớp dẫn xuất; chúng có thể là các lớp như Displayable, Persistent, Serializable, hoặc Sortable. Các mixin hầu như luôn là abstract (trừu tượng) và không nên được khởi tạo độc lập.

Mixins cần kế thừa đa lần nhưng không dẫn đến vấn đề "diamond-inheritance" (kế thừa kim cương) kinh điển nếu các mixin thực sự độc lập với nhau. Chúng cũng giúp thiết kế dễ hiểu hơn bằng cách nhóm các thuộc tính lại với nhau ("chunking"). Lập trình viên sẽ dễ hiểu hơn một đối tượng sử dụng mixin Displayable và Persistent thay vì giải quyết từng phương thức riêng lẻ cho hai thuộc tính đó.

Java và Visual Basic công nhận giá trị của mixin bằng cách cho phép kế thừa đa giao diện (multiple inheritance of interfaces) nhưng chỉ cho kế thừa đơn với lớp cụ thể. C++ hỗ trợ kế thừa đa lần cho cả giao diện và cài đặt (implementation). Lập trình viên chỉ nên sử dụng kế thừa đa lần sau khi cân nhắc kỹ các lựa chọn thay thế và tác động của nó lên độ phức tạp cũng như khả năng hiểu của hệ thống.

Vì Sao Có Nhiều Quy Tắc Cho Kế Thừa?

Phần này đã đưa ra hàng loạt quy tắc để giúp bạn tránh rắc rối với kế thừa. Thông điệp chính là kế thừa thường đi ngược lại với mục tiêu kỹ thuật tối thượng của lập trình viên, đó là quản lý độ phức tạp.

> **ĐIỂM THEN CHỐT:** Để kiểm soát độ phức tạp, hãy duy trì một sự thận trọng lớn với việc lạm dụng kế thừa.

Tóm lược khi nào nên dùng kế thừa, khi nào nên dùng chứa (containment):

- Nếu nhiều lớp có chung dữ liệu nhưng không chung hành vi, hãy tạo một đối tượng chung để các lớp này chứa.

- Nếu nhiều lớp có chung hành vi nhưng không chung dữ liệu, hãy kế thừa từ một lớp cơ sở chung định nghĩa các phương thức chung.
- Nếu nhiều lớp có chung cả dữ liệu và hành vi, hãy kế thừa từ một lớp cơ sở chung định nghĩa cả dữ liệu lẫn phương thức.
- Hãy kế thừa khi muốn lớp cơ sở kiểm soát giao diện; hãy dùng chứa khi muốn kiểm soát giao diện của chính mình.

Hàm Thành Viên và Dữ Liệu Thành Viên

Một Số Nguyên Tắc Khi Hiện Thực Hàm và Dữ Liệu Thành Viên

- **Giữ số lượng phương thức trên mỗi lớp ở mức thấp nhất có thể:** Một nghiên cứu về các chương trình C++ cho thấy số lượng phương thức cao trên mỗi lớp liên quan đến tỷ lệ lỗi cao hơn (Basili, Briand và Melo 1996). Tuy nhiên, các yếu tố cạnh tranh khác cũng có ý nghĩa lớn như cây kế thừa sâu, số lượng lớn các phương thức được gọi trong một lớp và sự phụ thuộc mạnh giữa các lớp. Hãy cân nhắc đánh đổi giữa việc tối thiểu số lượng phương thức và các yếu tố khác.
- **Không cho phép các hàm thành viên và toán tử được tự động sinh ra mà bạn không mong muốn:** Đôi khi bạn muốn ngăn một số hàm—ví dụ không cho phép gán, hoặc không cho phép khởi tạo đối tượng. Dù trình biên dịch tự động sinh ra các toán tử, bạn vẫn có thể ngăn việc sử dụng bằng cách khai báo constructor, toán tử gán hoặc hàm tương ứng ở mức private, nhờ đó ngăn không cho code bên ngoài truy cập. (Việc làm constructor private là một kỹ thuật chuẩn để định nghĩa singleton class, sẽ đề cập ở phần sau của chương này.)
- **Giảm thiểu số lượng phương thức khác nhau được gọi từ trong một lớp:** Một nghiên cứu cho thấy số lượng lỗi trong một lớp có liên quan chặt chẽ với tổng số phương thức được gọi từ bên trong lớp (Basili, Briand, và Melo 1996).

Một số khái niệm liên quan đến “fan out”

Đọc thêm

- **Hạn chế tối đa việc gọi gián tiếp các routine (thủ tục) tới các class khác.** Các kết nối trực tiếp có thể gây rủi ro. Các kết nối gián tiếp—chẳng hạn như `account.ContactPerson().DaytimeContactInfo().PhoneNumber()`—thường thậm chí còn rủi ro hơn.

- Các nhà nghiên cứu đã đề xuất một quy tắc gọi là “Law of Demeter” (Lieberherr và Holland, 1989), bản chất quy định rằng: Object A có thể gọi bất kỳ routine nào của chính nó. Nếu Object A khởi tạo một Object B, nó có thể gọi bất kỳ routine nào của Object B. Tuy nhiên, nó nên tránh gọi các routine của các object được cung cấp bởi Object B.

- Trong ví dụ về **account** ở trên, `account.ContactPerson()` là chấp nhận được, nhưng `account.ContactPerson().DaytimeContactInfo()` thì không nên sử dụng.

  Đây là một giải thích đơn giản hóa. Để biết chi tiết, vui lòng xem các tài liệu tham khảo ở cuối chương này.

- **Nhìn chung, hãy tối thiểu hóa mức độ cộng tác giữa các class.** Cố gắng tối giản tất cả các yếu tố sau:
  - Số loại object được khởi tạo.
  - Số lần gọi routine trực tiếp khác nhau tới các object đã khởi tạo.
  - Số lần gọi routine tới các object trả về bởi các object khác đã khởi tạo.

6.3 Các vấn đề trong thiết kế và hiện thực hóa

Constructor (Hàm khởi tạo)

- Sau đây là một số hướng dẫn dành riêng cho constructor. Các nguyên tắc dành cho constructor khá giống nhau giữa các ngôn ngữ (C++, Java, Visual Basic, v.v.). Destructors (hàm hủy) thì khác biệt hơn, bạn nên tham khảo các tài liệu bổ sung ở cuối chương để biết thêm về destructor.

- **Khởi tạo tất cả dữ liệu thành viên** trong tất cả constructor, nếu có thể. Việc này là một biện pháp lập trình phòng thủ hiệu quả mà ít tốn kém.

Đọc thêm

- **Thực thi tính chất singleton (đơn thể) bằng constructor private:** Nếu bạn muốn định nghĩa một class chỉ cho phép khởi tạo một object duy nhất, bạn có thể thực thi điều này bằng cách ẩn (private) tất cả constructor của class và cung cấp một routine static `GetInstance()` để truy xuất instance đơn nhất đó.

Ví dụ dưới đây minh họa nguyên tắc trên trong Java:

```
public class MaxId {
    // constructors and destructors
    // Đây là constructor private
    private MaxId() {
    }

    // public routines
    // Routine public cung cấp truy cập đến instance đơn nhất
    public static MaxId GetInstance() {
        return m_instance;
    }

    // private members
    // Đây là instance đơn nhất
    private static final MaxId m_instance = new MaxId();
}
```

- Constructor private chỉ được gọi khi đối tượng static `m_instance` được khởi tạo. Theo cách này, nếu muốn tham chiếu đến singleton MaxId, bạn chỉ đơn giản sử dụng `MaxId.GetInstance()`.

Ưu tiên deep copy (sao chép sâu) thay vì shallow copy (sao chép nông), trừ khi đã được chứng minh ngược lại

- Một trong những quyết định quan trọng về object phức tạp là lựa chọn giữa deep copy và shallow copy. **Deep copy** là sao chép tất cả dữ liệu thành viên của object; còn **shallow copy** thường chỉ tham chiếu hoặc trỏ đến một bản copy tham chiếu duy nhất, dù ý nghĩa cụ thể của "deep" và "shallow" có thể khác biệt.

- **Động lực tạo shallow copy** thường là tăng hiệu năng. Tuy nhiên, việc tạo nhiều bản sao cho object lớn hiếm khi gây tác động hiệu năng có thể đo lường được. Số lượng nhỏ object có thể gây vấn đề về hiệu năng, nhưng các lập trình viên thường không đoán chính xác được phần code nào thật sự gây ra vấn đề.

  (Xem thêm Chương 25, "Chiến lược tối ưu hóa mã nguồn (Code-Tuning Strategies)")

- Việc thêm phức tạp vào code vì các lý do hiệu năng không rõ ràng là một đánh đổi không tốt. Đa số trường hợp, deep copy dễ viết và bảo trì hơn shallow copy. Shallow copy phải bổ sung code để đếm reference, đảm bảo việc sao chép, so sánh, xóa object an toàn, ... Những đoạn code này dễ phát sinh lỗi, do đó chỉ nên dùng shallow copy khi thực sự có lý do thuyết phục.

- Nếu bạn buộc phải sử dụng shallow copy, hãy tham khảo thảo luận chi tiết trong **More Effective C++, Item 29 (Scott Meyers, 1996)** về vấn đề này trong C++. Martin Fowler cũng mô tả các bước chuyển đổi giữa shallow copy và deep copy trong sách **Refactoring (1999)** (Fowler gọi chúng lần lượt là *reference objects* và *value objects*).

6.4 Lý do tạo class

Những lý do tạo class và routine có thể trùng lặp. Xem thêm phần 7.1.

Một số lý do hợp lý để tạo class:

1. **Mô hình hóa các object thực tế (real-world objects):**
   - Mặc dù không phải lý do duy nhất, nhưng đây vẫn là một lý do hợp lý. Hãy tạo class cho mỗi loại object thực tế mà chương trình cần mô hình hóa, đưa vào class các dữ liệu cần thiết, và xây dựng các routine mô tả hành vi của object đó. (Tham khảo thêm phần thảo luận về ADT (Abstract Data Type) ở Mục 6.1).

2. **Mô hình hóa object trừu tượng (abstract objects):**
   - Lý do khác để tạo class là mô tả object trừu tượng—tức là object không phải object thực thể cụ thể ngoài đời, mà là một sự trừu tượng cho các object cụ thể khác. Ví dụ kinh điển là object `Shape`. `Circle` và `Square` là các thực thể cụ thể, còn `Shape` là sự khái quát đại diện cho các loại hình khác nhau.
   - Trong thực tế, các abstraction này không có sẵn, do đó thiết kế cần phân tích và trừu tượng hóa các thực thể cụ thể. Đây chính là thách thức lớn trong thiết kế hướng đối tượng.

3. **Giảm độ phức tạp:**
   - **Lý do quan trọng nhất để tạo class là giảm độ phức tạp của chương trình.** Tạo class để ẩn mọi thông tin mà bạn không cần nghĩ tới sau khi đã thiết kế xong class đó. Bạn sẽ chỉ cần suy nghĩ về chi tiết khi viết class, còn sau đó chỉ việc sử dụng mà không cần biết đến cấu trúc bên trong.
   - Các lý do khác như tối thiểu hóa kích thước mã, tăng khả năng bảo trì, cải thiện tính đúng đắn đều hợp lý, nhưng nếu không có sức mạnh trừu tượng hóa của class, các chương trình phức tạp sẽ gần như không thể kiểm soát

được.

4. **Cô lập độ phức tạp:**
    - Mọi dạng phức tạp—thuật toán phức tạp, tập dữ liệu lớn, giao thức giao tiếp phức tạp...—đều dễ phát sinh lỗi. Nếu lỗi xảy ra, sẽ dễ tra cứu hơn khi chúng chỉ xuất hiện ở trong một class thay vì rải rác ở nhiều nơi trong code. Việc sửa đổi cũng sẽ chỉ tác động đến một class, các phần khác không bị ảnh hưởng.
5. **Ẩn chi tiết hiện thực (implementation details):**
    - Mong muốn ẩn chi tiết cài đặt là một lý do rất tốt để tạo class, bất kể chi tiết đó có phức tạp như truy cập database hay đơn giản như việc lưu trữ một trường dữ liệu ở dạng số hay chuỗi.
6. **Giới hạn tác động của thay đổi:**
    - Thiết kế class nên đảm bảo các khu vực dễ thay đổi (hardware dependencies, input/output, kiểu dữ liệu phức tạp, business rule,...) được cô lập. Khi cần thay đổi, chỉ cần điều chỉnh trong phạm vi một hoặc vài class nhất định.

   Tham khảo thêm phần “Hide Secrets (Information Hiding)” ở Mục 5.3 để biết các nguồn thay đổi phổ biến.
7. **Ẩn dữ liệu toàn cục (global data):**
    - Nếu thật sự cần dùng dữ liệu toàn cục, hãy ẩn chi tiết hiện thực của chúng sau một interface của class. Làm việc này qua các routine truy cập giúp bạn dễ thay đổi cấu trúc dữ liệu mà không ảnh hưởng đến toàn chương trình, đồng thời kiểm soát việc truy cập và nhận ra rằng dữ liệu này thực ra nên là object data (dữ liệu thuộc về object), không phải toàn cục.

   Xem thêm thảo luận về các vấn đề liên quan đến dữ liệu toàn cục tại Mục 13.3, "Global Data”.
8. **Đơn giản hóa truyền tham số (parameter passing):**
    - Nếu bạn phải truyền một tham số qua nhiều routine, điều này có thể chỉ ra rằng bạn nên gom các routine đó vào trong một class chung, với tham số này là object data. Việc này không phải là một mục tiêu tự thân, nhưng việc truyền nhiều tham số quanh chương trình thường gợi ý rằng cách tổ chức class hiện tại có thể chưa tối ưu.
9. **Điểm kiểm soát tập trung:**
    - Nên đảm bảo mỗi tác vụ được kiểm soát tại một điểm trung tâm duy nhất. Ví dụ như việc kiểm soát số lượng entry trong bảng, quản lý tài nguyên như file, kết nối database, máy in... đều nên do một class xử lý để dễ quản lý. Nếu sau này cần thay đổi backend (ví dụ chuyển từ database sang file phẳng hoặc dữ liệu trong bộ nhớ), chỉ cần sửa một class là đủ.

   Tham khảo chi tiết về thông tin ẩn giấu (information hiding) tại Mục 5.3 “Hide Secrets (Information Hiding)”.

Thúc đẩy khả năng tái sử dụng mã nguồn

Việc đặt mã nguồn vào các lớp (class) có cấu trúc tốt cho phép tái sử dụng mã trong các chương trình khác một cách dễ dàng hơn so với việc nhúng cùng đoạn mã đó vào một lớp lớn hơn. Ngay cả khi một đoạn mã chỉ được gọi từ một nơi trong chương trình và có thể hiểu được như một phần của một lớp lớn, vẫn hợp lý để đặt nó vào một lớp riêng nếu đoạn mã đó có khả năng được sử dụng ở một chương trình khác.

NASA từng tiến hành một nghiên cứu tại Phòng Thí nghiệm Kỹ thuật Phần mềm về mười dự án hướng tới việc tái sử dụng mã nguồn (McGarry, Waligora và McDermott 1989). Trong cả hai cách tiếp cận hướng đối tượng (object-oriented) và hướng hàm (functionally oriented), các dự án ban đầu không thể lấy nhiều mã nguồn từ các dự án trước, do cơ sở mã nguồn chưa đủ phong phú. Sau đó, các dự án sử dụng thiết kế hướng hàm có thể tận dụng khoảng 35% mã nguồn từ các dự án trước. Các dự án sử dụng phương pháp hướng đối tượng có thể tận dụng hơn 70% mã nguồn từ những dự án trước đó. Nếu có thể tránh việc viết lại đến 70% mã nguồn nhờ lập kế hoạch trước, hãy thực hiện điều đó!

> **Tham khảo chéo:** Đáng chú ý, cốt lõi trong phương pháp của NASA về việc tạo lớp có thể tái sử dụng không liên quan đến việc “thiết kế cho tái sử dụng” ngay từ đầu. NASA xác định các lớp có khả năng tái sử dụng vào thời điểm cuối dự án. Sau đó, họ tiến hành các bước cần thiết để làm cho lớp đó có thể tái sử dụng như một dự án đặc biệt, được thực hiện vào cuối dự án chính hoặc ngay khi bắt đầu một dự án mới. Cách làm này giúp ngăn chặn “vàng mã” (gold-plating)—việc tạo ra những chức năng không cần thiết, chỉ làm phức tạp thêm hệ thống.

Lập kế hoạch cho một “họ” chương trình (family of programs)

Nếu bạn dự đoán một chương trình sẽ được chỉnh sửa hoặc mở rộng, nên cô lập các phần có khả năng thay đổi bằng cách đặt chúng vào các lớp riêng. Như vậy bạn có thể dễ dàng chỉnh sửa hoặc thay thế hoàn toàn các lớp này mà không ảnh hưởng đến phần còn lại của chương trình. Việc suy nghĩ không chỉ về một chương trình riêng lẻ mà còn về toàn bộ "họ" các chương trình có thể xuất hiện là một phương pháp hữu ích để dự đoán các loại thay đổi lớn (Parnas 1976).

Một vài năm trước, tôi quản lý một nhóm phát triển loạt chương trình phục vụ khách hàng bán bảo hiểm. Mỗi chương trình cần điều chỉnh cho phù hợp với biểu phí, định dạng báo cáo của từng khách hàng. Tuy nhiên, nhiều phần trong các chương trình là tương tự: các lớp nhập thông tin khách hàng tiềm năng, lưu dữ liệu khách vào cơ sở dữ liệu, tra cứu biểu phí, tính tổng chi phí bảo hiểm cho nhóm,... Nhóm phát triển đã phân tách chương trình để mọi thành phần thay đổi tùy theo khách hàng đều nằm trong một lớp riêng. Việc lập trình ban đầu có thể mất khoảng ba tháng, nhưng khi có khách hàng mới, chúng tôi chỉ cần viết một số lớp mới phù hợp và tích hợp vào mã hiện tại. Chỉ sau vài ngày làm việc, phần mềm đã được "cá nhân hóa" cho khách hàng!

Đóng gói các thao tác liên quan

Trong trường hợp bạn không thể ẩn thông tin, chia sẻ dữ liệu hoặc lập kế hoạch mở rộng, bạn vẫn có thể đóng gói một tập hợp các thao tác lại thành các nhóm hợp lý, ví dụ: các hàm lượng giác, hàm thống kê, các hàm xử lý chuỗi, thao tác với bit, thao tác đồ họa,... Lớp (class) là một cách để kết hợp các thao tác liên quan. Bạn cũng có thể sử dụng package (gói), namespace (không gian tên), hoặc file header, tùy thuộc vào ngôn ngữ lập trình sử dụng.

Thực hiện refactoring cụ thể

Nhiều kỹ thuật refactoring (tái cấu trúc mã) được nêu trong Chương 24 "Refactoring" dẫn đến việc tạo ra các lớp mới—bao gồm việc chia một lớp thành hai lớp, ẩn delegate, loại bỏ lớp trung gian, hoặc giới thiệu extension class (lớp mở rộng). Những lớp mới này có thể được tạo ra nhằm đáp ứng các mục tiêu đã đề cập ở phần trước.

Những loại lớp cần tránh

Mặc dù nói chung lớp là hữu ích, bạn vẫn có thể gặp một số vấn đề sau đây cần lưu ý:

Tránh tạo "god class" (lớp thượng đế)

Hạn chế tạo các lớp toàn năng/tất biết (god class) có khả năng kiểm soát tất cả và biết về mọi thứ. Nếu một lớp chủ yếu lấy dữ liệu từ các lớp khác thông qua các hàm `Get()` và `Set()` (tức là can thiệp quá mức vào hoạt động của lớp khác), hãy cân nhắc xem liệu chức năng đó có nên được tổ chức lại vào các lớp bị lấy dữ liệu không thay vì dồn hết lên lớp god class (Riel 1996).

> **Tham khảo chéo:** Loại lớp này thường được gọi là cấu trúc (structure). Xem thêm về cấu trúc ở Mục 13.1 "Structures".

Loại bỏ các lớp không liên quan

Nếu một lớp chỉ chứa dữ liệu mà không có hành vi (behavior), hãy tự hỏi liệu nó có thực sự là một lớp hay không, và cân nhắc hạ cấp nó về thuộc tính (attribute) của một hoặc vài lớp khác.

Tránh đặt tên lớp bằng động từ

Một lớp chỉ có hành vi mà không có dữ liệu thường không phải là lớp đích thực. Hãy cân nhắc chuyển một lớp như `DatabaseInitialization()` hay `StringBuilder()` thành một routine (thủ tục/hàm) thuộc một lớp khác.

Tóm tắt các lý do nên tạo lớp

Dưới đây là danh sách các lý do chính đáng để tạo một lớp:

- **Mô hình hóa các đối tượng thực tế (real-world objects)**
- **Mô hình hóa các đối tượng trừu tượng (abstract objects)**
- **Giảm độ phức tạp**
- **Cô lập yếu tố phức tạp**
- **Ẩn chi tiết triển khai**
- **Hạn chế ảnh hưởng của thay đổi**
- **Ẩn dữ liệu toàn cục (global data)**
- **Đơn giản hóa việc truyền tham số**
- **Tạo các điểm kiểm soát trung tâm**

- **Thúc đẩy khả năng tái sử dụng mã nguồn**
- **Lập kế hoạch cho một "họ" chương trình**
- **Đóng gói các thao tác liên quan**
- **Thực hiện tái cấu trúc cụ thể**

6.5 Các vấn đề cụ thể theo ngôn ngữ lập trình

Cách tiếp cận với class (lớp) ở các ngôn ngữ lập trình khác nhau có nhiều điểm thú vị. Hãy xem xét cách ghi đè (override) một routine thành viên để đạt đa hình (polymorphism) ở lớp dẫn xuất:

- Trong **Java**, mọi routine đều có thể được ghi đè theo mặc định; routine cần khai báo `final` để ngăn không cho lớp con ghi đè.
- Trong **C++**, routine không thể bị ghi đè theo mặc định; routine cần khai báo `virtual` ở lớp cơ sở để có thể bị ghi đè.
- Trong **Visual Basic**, routine phải khai báo `Overridable` ở lớp cơ sở và lớp dẫn xuất phải dùng từ khóa `Overrides`.

Một số điểm về class có thể khác biệt lớn tùy ngôn ngữ:

- **Hành vi của constructor và destructor trong cây kế thừa**
- **Hành vi của constructor và destructor khi gặp xử lý ngoại lệ (exception-handling)**
- **Tầm quan trọng của constructor mặc định (không có đối số)**
- **Thời điểm gọi destructor hoặc finalizer**
- **Sự hợp lý khi ghi đè các toán tử dựng sẵn của ngôn ngữ, bao gồm phép gán và so sánh bằng**
- **Cách quản lý bộ nhớ khi một object được tạo, phá hủy, hay đi ra khỏi phạm vi**

Các thảo luận chi tiết về những vấn đề này nằm ngoài phạm vi cuốn sách này, nhưng mục "Additional Resources" sẽ giới thiệu các tài liệu chuyên sâu cho từng ngôn ngữ.

6.6 Bên ngoài lớp: Package (gói)

> **Tham khảo chéo:** Xem thêm về sự khác biệt giữa class và package tại "Levels of Design" trong Mục 5.2.

Hiện nay, class là công cụ tốt nhất giúp lập trình viên đạt được tính mô-đun. Tuy nhiên, mô-đun hóa (modularity) là chủ đề lớn, vượt ra ngoài phạm vi của class. Trong nhiều thập kỷ qua, phát triển phần mềm đã tiến bộ chủ yếu nhờ mở rộng kích thước các khối xây dựng mã nguồn mà chúng ta có thể thao tác: ban đầu là câu lệnh (statement), tiếp đến là subroutine (thủ tục/hàm con), và gần đây là class.

Có thể thấy, chúng ta sẽ hỗ trợ tốt hơn các mục tiêu như trừu tượng hóa (abstraction) và đóng gói (encapsulation) nếu có công cụ tốt để nhóm các object (đối tượng) lại với nhau. Ada đã hỗ trợ khái niệm package hơn một thập kỷ trước, Java hiện cũng vậy. Nếu bạn làm việc với ngôn ngữ không hỗ trợ package trực tiếp, bạn vẫn có thể tạo phiên bản "gần đúng" của package bằng cách áp dụng các quy chuẩn lập trình sau:

- **Quy ước đặt tên** để phân biệt class nào là public (công khai) và class nào chỉ sử dụng nội bộ trong package.
- **Quy ước đặt tên, tổ chức mã nguồn (cấu trúc dự án)** hoặc kết hợp cả hai để xác định mỗi class thuộc về package nào.
- **Các quy tắc rằng package nào được phép sử dụng package nào khác**, bao gồm việc sử dụng theo hình thức kế thừa, bao chứa (containment), hoặc cả hai.

Đây là những ví dụ điển hình về phân biệt lập trình trong một ngôn ngữ (programming in a language) và lập trình vào một ngôn ngữ (programming into a language). Để tìm hiểu thêm về sự khác biệt này, hãy xem Mục 34.4 "Program into Your Language, Not in It".

CHECKLIST: Chất lượng Class

> **Tham khảo chéo:** Đây là danh sách kiểm tra về chất lượng class. Để biết các bước xây dựng class, hãy xem checklist "The Pseudocode Programming Process" trong Chương 9, trang 233.

Abstract Data Types

- ❑ Bạn đã xem xét các class trong chương trình như các kiểu dữ liệu trừu tượng (abstract data types) và đánh giá giao diện của chúng từ góc độ đó chưa?

Abstraction

- ❑ Class có mục đích trung tâm rõ ràng không?

Tài liệu Tham khảo về Lập trình Hướng đối tượng và Thiết kế Chương trình

Tài liệu về Lập trình Hướng đối tượng

Meyer, Bertrand. *Object-Oriented Software Construction*, Ấn bản thứ 2. New York, NY: Prentice Hall PTR, 1997.

- Cuốn sách này cung cấp một phân tích sâu sắc về các kiểu dữ liệu trừu tượng (abstract data types) và giải thích vai trò nền tảng của chúng đối với các lớp (classes).
- Các chương 14–16 bàn luận chi tiết về tính kế thừa (inheritance).
- Meyer trình bày luận điểm ủng hộ kế thừa đa năng (multiple inheritance) tại chương 15.

Riel, Arthur J. *Object-Oriented Design Heuristics*, Reading, MA: Addison-Wesley, 1996.

- Cuốn sách này chứa nhiều gợi ý nhằm cải thiện thiết kế chương trình, chủ yếu ở cấp độ lớp.
- Tác giả từng tránh đọc cuốn sách này nhiều năm vì hình thức dày cộp của nó — thật hài hước khi bản thân đều thuộc nhóm tác giả "ngồi trong nhà kính". Tuy nhiên, phần nội dung thực tế chỉ khoảng 200 trang.
- Văn phong của Riel dễ tiếp cận, cuốn hút. Nội dung tập trung, thực tiễn.

Tài liệu về Ngôn ngữ Lập trình

C++

Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*, Ấn bản thứ 2, Reading, MA: Addison-Wesley, 1998.
Meyers, Scott. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Reading, MA: Addison-Wesley, 1996.

- Cả hai cuốn sách của Meyers đều là tài liệu tham khảo kinh điển (canonical reference) dành cho lập trình viên C++.
- Các cuốn sách này vừa mang tính giải trí, vừa giúp người đọc cảm nhận được những sắc thái tinh tế của ngôn ngữ C++ dưới góc nhìn của một "luật gia ngôn ngữ".

Java

Bloch, Joshua. *Effective Java Programming Language Guide*, Boston, MA: Addison-Wesley, 2001.

- Sách của Bloch cung cấp nhiều lời khuyên hữu ích dành riêng cho Java, đồng thời giới thiệu các thực hành tốt trong lập trình hướng đối tượng nói chung.

Visual Basic

Những cuốn sách sau là tài liệu tham khảo tốt về các lớp trong Visual Basic:

- **Foxall, James. *Practical Standards for Microsoft Visual Basic .NET*, Redmond, WA: Microsoft Press, 2003.**
- **Cornell, Gary và Jonathan Morrison. *Programming VB .NET: A Guide for Experienced Programmers*, Berkeley, CA: Apress, 2002.**
- **Barwell, Fred, và cộng sự. *Professional VB .NET*, Ấn bản thứ 2, Wrox, 2002.**

Chương 6: Xây dựng Lớp (Working Classes)

Các Ý Chính

- **Giao diện lớp (class interface) cần đảm bảo tính trừu tượng (abstraction) nhất quán.** Nhiều vấn đề phát sinh khi nguyên tắc này bị phá vỡ.
- **Giao diện lớp nên che giấu một đối tượng nào đó:** có thể là giao diện hệ thống, quyết định thiết kế, hoặc chi tiết hiện thực.
- **Bao chứa (containment) thường được ưu tiên hơn so với kế thừa (inheritance) trừ phi bạn đang mô hình hóa mối quan hệ "là một" ("is a" relationship).**
- **Kế thừa là công cụ hữu ích, nhưng nó làm tăng độ phức tạp, đi ngược lại Nguyên tắc Kỹ thuật Chính của phần mềm: quản lý sự phức tạp.**

- **Lớp là công cụ chính để quản lý sự phức tạp.** Dành đủ sự chú ý cần thiết cho việc thiết kế lớp để đạt được mục tiêu này.

Chương 7: Các Thủ tục/Phương thức Chất lượng Cao (High-Quality Routines)

Mục lục

Chủ đề liên quan

- Các bước xây dựng routine: Mục 9.3
- Làm việc với class: Chương 6
- Kỹ thuật thiết kế tổng quát: Chương 5
- Kiến trúc phần mềm: Mục 3.5

Tóm tắt nội dung

Chương 6 đã mô tả chi tiết về cách tạo ra các lớp. Chương này sẽ tập trung đặc biệt vào các routine (thủ tục/phương thức), những đặc điểm tạo nên sự khác biệt giữa một routine tốt và routine kém.

Nếu bạn muốn tìm hiểu các vấn đề ảnh hưởng đến thiết kế routine trước khi đi sâu vào chi tiết, hãy đọc Chương 5 “Thiết kế trong quá trình xây dựng” trước, sau đó quay lại chương này. Một số đặc tính quan trọng của routine chất lượng cao cũng được bàn tới ở Chương 8, “Lập trình phòng thủ” (Defensive Programming). Nếu bạn quan tâm tới các bước thực hiện tạo routine và lớp, Chương 9 “Quy trình Lập trình bằng Pseudocode” (Pseudocode Programming Process) có thể là điểm xuất phát phù hợp hơn.

Trước khi đi sâu vào chi tiết về routine chất lượng cao, sẽ hữu ích khi xác định hai thuật ngữ cơ bản: “routine” là gì? Routine là một phương thức (method) hoặc thủ tục (procedure) độc lập, được gọi tới nhằm thực hiện một mục đích duy nhất. Ví dụ: hàm (function) trong C++, phương thức (method) trong Java, hàm hoặc thủ tục phụ (sub procedure) trong Microsoft Visual Basic. Đôi khi, macro trong C và C++ cũng có thể được xem là routine. Nhiều kỹ thuật nhằm tạo routine chất lượng cao có thể áp dụng cho những biến thể này.

Một routine chất lượng cao là gì? Đây là một câu hỏi khó. Có lẽ câu trả lời đơn giản nhất là nêu ra điều routine chất lượng cao không phải. Dưới đây là ví dụ về một routine kém chất lượng:

```
void HandleStuff( CORP_DATA & inputRec, int crntQtr, EMP_DATA empRec,
double & estimRevenue, double ytdRevenue, int screenX, int screenY,
COLOR_TYPE & newColor, COLOR_TYPE & prevColor, StatusType & status,
int expenseType )
{
    int i;
    for ( i = 0; i < 100; i++ ) {
        inputRec revenue[i] = 0;
        inputRec expense[i] = corpExpense[ crntQtr ][ i ];
    }
    UpdateCorpDatabase( empRec );
    estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;
    newColor = prevColor;
    status = SUCCESS;
    if ( expenseType == 1 ) {
        for ( i = 0; i < 12; i++ )
            profit[i] = revenue[i] - expense_type1[i];
    }
    else if ( expenseType == 2 ) {
        profit[i] = revenue[i] - expense_type2[i];
    }
    else if ( expenseType == 3 )
        profit[i] = revenue[i] - expense_type3[i];
}
```

(*Đã sửa một số lỗi về dấu chấm và gạch dưới cho dễ hiểu; bản gốc đánh máy sai và thiếu dấu.*)

Vấn đề của routine này là gì?

Bạn nên nhận ra ít nhất 10 vấn đề khác nhau trong routine trên. Sau khi tự liệt kê, hãy so sánh với danh sách sau:

- **Tên routine kém**: `HandleStuff()` không cung cấp thông tin về chức năng routine.
- **Routine không có tài liệu chú thích** (vấn đề này lớn hơn phạm vi routine đơn lẻ, sẽ được bàn ở Chương 32 "Mã tự chú thích").
- **Bố cục mã kém**: Cách trình bày trên trang không cho thấy tổ chức logic. Các chiến lược bố cục (layout strategy) được sử dụng tuỳ tiện, với các phong cách khác nhau ở những đoạn code khác nhau — thử so sánh nơi `expenseType == 2` và `expenseType == 3`. (Vấn đề layout sẽ bàn ở Chương 31 "Bố cục và Phong cách".)
- **Biến đầu vào của routine bị thay đổi**: `inputRec` là biến đầu vào nhưng lại bị sửa đổi. Nếu là đầu vào, giá trị của nó không nên bị thay đổi (trong C++ nên sử dụng `const`). Nếu biến này dự kiến bị sửa đổi, tên gọi không nên là `inputRec`.
- **Routine đọc và ghi biến toàn cục** — đọc từ `corpExpense` và ghi vào `profit`. Thay vào đó, routine nên giao tiếp trực tiếp với các routine khác, hạn chế sử dụng biến toàn cục.
- **Routine không có mục tiêu đơn nhất**: Routine khởi tạo biến, ghi dữ liệu vào cơ sở dữ liệu, thực hiện tính toán — nhưng các hoạt động này không liên kết với nhau về mặt logic. Mong muốn routine chỉ thực hiện một mục tiêu rõ ràng duy nhất.
- **Routine không tự bảo vệ trước dữ liệu không hợp lệ**: Nếu `crntQtr == 0`, phép chia `ytdRevenue * 4.0 / (double) crntQtr` sẽ gây lỗi chia cho 0.
- **Routine sử dụng các số "ma thuật" (magic numbers)**: 100, 4.0, 12, 2, 3. (Xem thêm ở Mục 12.1 "Các con số nói chung".)
- **Một số tham số không được sử dụng**: `screenX` và `screenY` không xuất hiện trong nội dung routine.
- **Một tham số được truyền sai dạng**: `prevColor` truyền theo dạng tham chiếu (`&`) nhưng không hề được gán giá trị mới trong routine.
- **Routine có quá nhiều tham số**: Giới hạn tối đa để dễ hiểu là khoảng 7 tham số, nhưng routine này có tới 11 tham số. Thứ tự trình bày tham số lộn xộn, thiếu rõ ràng, làm người đọc khó quan sát hoặc đếm nổi số lượng.
- **Thứ tự tham số sắp xếp kém và không có chú thích** (vấn đề về sắp xếp tham số sẽ bàn trong chương này, còn chú thích ở Chương 32).

  Ngoài máy tính, routine là phát minh vĩ đại nhất của khoa học máy tính. Routine giúp chương trình dễ đọc, dễ hiểu hơn bất kỳ đặc tính nào khác của ngôn ngữ lập trình, và quả là "tội ác" nếu lạm dụng routine bằng các đoạn mã kiểu như ví dụ vừa rồi.

  Routine cũng là kỹ thuật tuyệt vời nhất từng phát minh ra nhằm tiết kiệm không gian và cải thiện hiệu năng. Hãy tưởng tượng mã nguồn của bạn sẽ phình ra cỡ nào nếu phải lặp lại toàn bộ đoạn mã cho mỗi lần gọi thay vì chỉ nhảy vào routine.

  Bạn nói: "Được thôi, tôi đã biết routine rất tuyệt, và lúc nào cũng sử dụng chúng. Chủ đề này quá cơ bản, thế anh còn muốn tôi làm gì nữa?"
  Tôi muốn bạn hiểu rằng, có rất nhiều lý do hợp lý để xây dựng một routine và có những cách đúng, sai trong việc này. Khi còn là sinh viên ngành khoa học máy tính, tôi cứ tưởng lý do duy nhất để tạo routine là tránh lặp mã. Cuốn giáo trình nhập môn chỉ nói rằng routine tốt vì nó giúp phát triển, gỡ lỗi, tài liệu hóa, bảo trì chương trình dễ hơn, thế thôi. Ngoài chi tiết cú pháp về tham số và biến cục bộ, sách giáo trình đã không giải thích đầy đủ về lý thuyết & thực tiễn về routine.

Các mục tiếp theo sẽ cung cấp giải thích hoàn chỉnh hơn.

7.1 Những Lý do Hợp lý để Tạo Một Routine

Dưới đây là các lý do hợp lý để tạo routine. Các lý do này phần nào chồng lấn nhau và không phải là một bộ tập độc lập:

- **Giảm độ phức tạp**: Lý do quan trọng nhất để tạo một routine là giảm độ phức tạp của chương trình. Hãy tạo routine để che giấu thông tin (information hiding), để bạn không phải bận tâm đến các chi tiết bên trong. Tất nhiên, khi viết routine bạn phải quan tâm tới các chi tiết đó, nhưng **sau khi routine hoàn thành, bạn có thể quên chúng đi mà chỉ cần sử dụng routine mà không cần kiến thức về cách hoạt động bên trong của nó**.

Dấu hiệu nên tách một routine thành routine riêng biệt

Một dấu hiệu cho thấy cần tách một routine riêng biệt là **vòng lặp lồng nhau ở mức sâu** hoặc có điều kiện lồng ghép phức tạp. Hãy giảm độ phức tạp của routine bao quanh bằng cách tách phần mã lồng trong đó ra thành một routine riêng.

Tạo một lớp trừu tượng trung gian, dễ hiểu

Đưa một đoạn mã vào một routine được đặt tên phù hợp là một trong những phương pháp tốt nhất để diễn giải mục đích của nó. Thay vì đọc một chuỗi các câu lệnh như sau:

```
if ( node <> NULL ) then
   while ( node next <> NULL ) do
      node = node next
      leafName = node name
   end while
else
   leafName = ""
end if
```

bạn chỉ cần đọc một câu lệnh như sau:

```
leafName = GetLeafName( node )
```

Routine mới rất ngắn, hầu hết thông tin giải thích đã thể hiện qua tên routine. Tên này nâng mức trừu tượng cao hơn so với tám dòng mã ban đầu giúp mã nguồn dễ đọc và dễ hiểu hơn, đồng thời giảm độ phức tạp bên trong routine chứa đoạn mã này.

Tránh lặp mã (duplicate code)

Lý do phổ biến nhất để tạo routine là **tránh trùng lặp mã**. Việc có mã giống nhau ở hai routine cho thấy đang gặp vấn đề về phân rã chương trình. Hãy tách phần mã trùng lặp ra khỏi cả hai routine, viết thành một phiên bản tổng quát trong một lớp cơ sở (base class) và đặt hai routine đặc thù vào các lớp dẫn xuất (subclass). Ngoài ra, bạn cũng có thể đưa phần mã chung vào routine mới cho cả hai routine gọi đến.

Khi chỉ có một nơi chứa mã, bạn:

- Tiết kiệm không gian (không còn code lặp lại).
- Dễ dàng chỉnh sửa (chỉ cần sửa tại một chỗ).
- Đảm bảo tính nhất quán, hạn chế lỗi phát sinh do chỉnh sửa không đồng nhất ở nhiều nơi.

Hỗ trợ kế thừa (support subclassing)

Việc ghi đè (override) một routine ngắn, được phân tách tốt sẽ dễ dàng và ít sinh lỗi hơn routine dài, cấu trúc kém. Giữ routine có thể ghi đè đơn giản sẽ giảm nguy cơ lỗi trong các lớp dẫn xuất.

Ẩn trình tự xử lý (hide sequences)

Nên ẩn đi thứ tự xử lý các sự kiện. Ví dụ, nếu chương trình lấy dữ liệu từ người dùng rồi mới lấy dữ liệu phụ từ file, các routine tương ứng không nên phụ thuộc vào việc routine kia đã được gọi trước hay chưa. Nếu có hai dòng code đọc phần tử đầu stack và giảm biến `stackTop`, hãy đóng gói hai dòng này vào một routine `PopStack()` để ẩn giả định về thứ tự thao tác. Việc này tốt hơn là đưa giả định đó rải rác khắp chương trình.

Ẩn thao tác với con trỏ (pointer operations)

Thao tác với con trỏ thường khó đọc và dễ sai. Việc tách các thao tác này vào routine giúp bạn tập trung vào mục đích thay vì chi tiết kỹ thuật thao tác con trỏ. Nếu chỉ thao tác ở một nơi, khả năng đảm bảo đúng cao hơn. Khi muốn thay đổi kiểu dữ liệu (data type) tốt hơn con trỏ, bạn chỉ cần cập nhật một nơi mà không ảnh hưởng tới nơi khác.

Cải thiện tính di động (portability)

Sử dụng routine để cô lập các chức năng không di động, như đặc tả ngôn ngữ không chuẩn, phụ thuộc vào phần cứng hoặc hệ điều hành... Việc này giúp dễ dàng xác định và xử lý những phần cần điều chỉnh khi chuyển chương trình sang môi trường khác.

Đơn giản hóa điều kiện logic phức tạp (complicated boolean tests)

Thường không cần hiểu tường tận các phép thử boolean để nắm được luồng chương trình. Việc đóng gói kiểm tra logic phức tạp thành một hàm giúp code:

1. Dễ đọc hơn (vì tách chi tiết khỏi dòng chính).
2. Tên hàm mô tả rõ ý nghĩa. Việc dùng tên hàm nêu bật vai trò của điều kiện, khuyến khích bạn đầu tư làm cho chi tiết phép thử dễ hiểu hơn. Nhờ đó, cả dòng lệnh chính và điều kiện đều trở nên sáng rõ. Đơn giản hóa kiểm tra logic là một ví dụ của việc giảm phức tạp đã bàn luận ở trên.

Nâng cao hiệu năng (performance)

Có thể tối ưu mã chỉ ở một nơi thay vì nhiều chỗ. Khi tất cả mã chỉ tập trung tại một routine, việc profile và tìm điểm gây kém hiệu suất dễ hơn. Tối ưu hóa routine này sẽ mang lại lợi ích cho tất cả nơi sử dụng, dù trực tiếp hay gián tiếp. Qua đó giúp bạn dễ dàng chuyển sang thuật toán hoặc ngôn ngữ hiệu quả hơn nếu cần.

> **Lưu ý:** Với nhiều lý do chính đáng như trên để tạo routine, không cần thiết phải bắt buộc mọi routine đều phải ngắn. Một số công việc có thể thực hiện tốt hơn khi được gói trong một routine lớn. (Độ dài tối ưu cho routine được thảo luận ở Mục 7.4, "How Long Can a Routine Be?").

Các thao tác tưởng như quá đơn giản để đóng gói thành routine

Một trong những trở ngại lớn khiến nhiều người ngại tạo routine hiệu quả là do không muốn đóng gói những đoạn mã rất ngắn. Xây dựng một routine cho 2-3 dòng mã có thể bị cho là quá mức cần thiết, nhưng thực tế cho thấy các routine nhỏ đem lại nhiều lợi ích.

Điểm mấu chốt

Routine nhỏ giúp **nâng cao khả năng đọc** mã nguồn. Ví dụ, từng xuất hiện dòng mã sau ở khoảng một tá (12) vị trí trong chương trình:

```
points = deviceUnits * ( POINTS_PER_INCH / DeviceUnitsPerInch() )
```

Dù không quá phức tạp, đọc kỹ thì ai cũng hiểu là chuyển đổi đơn vị đo về đơn vị point. Tuy vậy, nếu chuyển dòng này thành một routine tên rõ ràng như sau:

```
Function DeviceUnitsToPoints ( deviceUnits Integer ): Integer
   DeviceUnitsToPoints = deviceUnits *
      ( POINTS_PER_INCH / DeviceUnitsPerInch() )
End Function
```

Thì mọi vị trí dùng sẽ được viết như sau:

```
points = DeviceUnitsToPoints( deviceUnits )
```

Dòng này rõ ràng và gần như tự giải thích ý nghĩa.

Một lý do khác để đưa thao tác nhỏ vào function là: **các thao tác nhỏ thường phát triển thành tác vụ lớn hơn**. Ví dụ, ban đầu không phát hiện ra trường hợp hàm `DeviceUnitsPerInch()` trả về 0. Phải xử lý chia cho 0 bằng cách thêm điều kiện kiểm tra:

```
Function DeviceUnitsToPoints( deviceUnits: Integer ) Integer;
   if ( DeviceUnitsPerInch() <> 0 )
      DeviceUnitsToPoints = deviceUnits *
         ( POINTS_PER_INCH / DeviceUnitsPerInch() )
   else
      DeviceUnitsToPoints = 0
   end if
End Function
```

Nếu đoạn mã này vẫn xuất hiện ở 12 nơi, bạn sẽ phải thêm 36 dòng kiểm tra mới. Nếu dùng routine, chỉ cần viết thêm 3 dòng.

Tóm tắt lý do nên tạo routine

Danh sách dưới đây tóm tắt các lý do xác đáng để tạo routine:

- Giảm độ phức tạp (reduce complexity)
- Tạo trừu tượng trung gian, dễ hiểu (introduce an intermediate, understandable abstraction)
- Tránh lặp mã (avoid duplicate code)
- Hỗ trợ kế thừa (support subclassing)
- Ẩn trình tự xử lý (hide sequences)
- Ẩn thao tác với con trỏ (hide pointer operations)
- Cải thiện tính di động (improve portability)
- Đơn giản hóa điều kiện logic phức tạp (simplify complicated boolean tests)
- Nâng cao hiệu năng (improve performance)

Ngoài ra, nhiều lý do để tạo class (lớp) cũng đồng thời là lý do để tạo routine:

- Cô lập độ phức tạp (isolate complexity)
- Ẩn chi tiết hiện thực (hide implementation details)
- Hạn chế ảnh hưởng của thay đổi (limit effects of changes)
- Ẩn dữ liệu toàn cục (hide global data)
- Tạo điểm điều khiển tập trung (make central points of control)
- Tạo mã có thể tái sử dụng (facilitate reusable code)
- Thực hiện tái cấu trúc cụ thể (accomplish a specific refactoring)

7.2 Thiết kế ở cấp độ routine

Khái niệm **cohesion** (độ liên kết) được giới thiệu trong bài báo của Wayne Stevens, Glenford Myers và Larry Constantine (1974). Các khái niệm hiện đại hơn như **abstraction** (trừu tượng) và **encapsulation** (đóng gói) đem lại nhiều góc nhìn sâu sắc hơn ở cấp độ lớp, nhưng **cohesion** vẫn được dùng phổ biến như một nguyên tắc thiết kế ở cấp độ routine.

> **Tham khảo:** Về sự liên kết ở cấp độ tổng quát, xem mục "Aim for Strong Cohesion" trong Section 5.3.

Với routine, cohesion đề cập đến mức độ các thao tác trong routine có liên quan chặt chẽ với nhau. Một số lập trình viên gọi là "strength": các thao tác trong routine liên quan với nhau chặt chẽ đến mức nào? Ví dụ, routine như `Cosine()` có cohesion hoàn hảo vì dành trọn cho một tác vụ duy nhất. Ngược lại, routine như `CosineAndTan()` có cohesion thấp hơn vì cố làm nhiều việc. Mục tiêu là mỗi routine chỉ làm một việc tốt nhất và không làm thêm việc khác.

Lợi ích thực tiễn

Cohesion cao giúp tăng độ tin cậy của chương trình. Một nghiên cứu trên 450 routine cho thấy **50% routine có cohesion cao không có lỗi**, trong khi chỉ có **18% routine cohesion thấp là không lỗi** (Card, Church, Agresti 1986). Nghiên cứu khác cũng với 450 routine (một sự trùng hợp ngẫu nhiên) nhận thấy routine có tỷ lệ coupling-to-cohesion (ghép nối so với liên kết) cao nhất chứa **nhiều lỗi gấp 7 lần** so với group thấp nhất, và **chi phí sửa lỗi cao gấp 20 lần** (Selby và Basili 1991).

Các thảo luận về cohesion thường chia thành nhiều cấp độ khác nhau. Tuy nhiên, điều quan trọng là bạn hiểu khái niệm và sử dụng làm tiêu chí giúp routine của bạn đạt mức cohesion cao nhất có thể.

Ví dụ về các routine có tính liên kết cao

Các ví dụ về routine (chương trình con) có tính liên kết (cohesion) cao bao gồm:

- `sin()`
- `GetCustomerName()`
- `EraseFile()`
- `CalculateLoanPayment()`
- `AgeFromBirthdate()`

Tất nhiên, việc đánh giá tính liên kết của các routine này dựa trên giả định rằng chúng thực hiện đúng chức năng như tên gọi —nếu chúng làm thêm những việc khác, thì mức độ liên kết sẽ giảm đi và tên cũng không còn phù hợp.

Một số loại liên kết khác thường được xem là chưa lý tưởng

Sequential cohesion (liên kết tuần tự)

Liên kết tuần tự xuất hiện khi một routine chứa các thao tác cần được thực hiện theo một thứ tự nhất định, chia sẻ dữ liệu qua từng bước, nhưng các thao tác đó khi kết hợp lại cũng chưa tạo thành một chức năng hoàn chỉnh.

Ví dụ về liên kết tuần tự: một routine nhận ngày sinh, tính tuổi của nhân viên và thời gian còn lại đến khi nghỉ hưu. Nếu routine này tính tuổi trước, sau đó sử dụng kết quả vừa thu được để tính thời gian đến nghỉ hưu, thì nó có liên kết tuần tự. Nếu routine tách biệt việc tính tuổi và tính thời gian về hưu (chỉ cùng sử dụng dữ liệu ngày sinh), thì chỉ có liên kết giao tiếp (communicational cohesion).

Làm thế nào để nâng cao tính liên kết chức năng (functional cohesion)? Hãy tạo các routine riêng biệt để:

- Tính tuổi nhân viên từ ngày sinh
- Tính thời gian đến khi nghỉ hưu từ ngày sinh
  Routine tính thời gian đến nghỉ hưu có thể gọi routine tính tuổi. Cả hai routine sẽ có liên kết chức năng. Các routine khác có thể gọi một trong hai hoặc cả hai routine này.

Communicational cohesion (liên kết giao tiếp)

Liên kết giao tiếp xảy ra khi các thao tác trong một routine sử dụng cùng một bộ dữ liệu và không có mối liên hệ nào khác. Ví dụ, một routine in báo cáo tổng hợp rồi khởi tạo lại dữ liệu tổng hợp được truyền vào; hai thao tác này chỉ liên quan qua cùng một bộ dữ liệu.

Để tăng mức độ liên kết tốt hơn, dữ liệu tổng hợp nên được khởi tạo lại gần nơi nó được tạo ra, không nên đặt trong routine in báo cáo. Nên tách các thao tác này thành các routine riêng biệt:

- Routine đầu tiên in báo cáo
- Routine thứ hai khởi tạo lại dữ liệu, được gọi gần nơi tạo/chỉnh sửa dữ liệu
  Cả hai routine sẽ được gọi từ routine cấp cao vốn đang gọi routine có liên kết giao tiếp này.

Temporal cohesion (liên kết theo thời điểm)

Liên kết theo thời điểm xuất hiện khi các thao tác được kết hợp vào một routine vì chúng được thực thi cùng thời điểm. Ví dụ điển hình:

- `Startup()`
- `CompleteNewEmployee()`
- `Shutdown()`

Một số lập trình viên coi liên kết này là không thể chấp nhận, bởi nó thường đi kèm thói quen lập trình xấu như nhồi nhét đủ loại mã lệnh vào routine `Startup()`. Để tránh vấn đề này, hãy xem các routine mang tính thời điểm là bộ tổ chức cho các sự kiện khác.
Ví dụ, routine `Startup()` có thể:

- Đọc file cấu hình
- Khởi tạo file tạm
- Thiết lập bộ quản lý bộ nhớ (memory manager)
- Hiển thị màn hình giao diện ban đầu

Để routine này hiệu quả, hãy để nó gọi các routine con thực hiện từng hoạt động cụ thể thay vì thực hiện trực tiếp trong đó. Cách này làm rõ mục đích của routine là điều phối các hoạt động, không phải trực tiếp thực hiện từng bước.

Vấn đề đặt ra ở đây là lựa chọn tên cho routine ở mức độ trừu tượng phù hợp. Nếu đặt tên routine là `ReadConfigFileInitScratchFileEtc()`, điều này gợi ý routine đó có liên kết ngẫu nhiên (coincidental cohesion). Nếu đặt tên là `Startup()`, ý nghĩa sẽ rõ ràng hơn: routine có một mục tiêu duy nhất và mang tính liên kết chức năng.

Những loại liên kết không thể chấp nhận

Những loại liên kết sau thường không thể chấp nhận được, bởi chúng tạo ra code tổ chức kém, khó kiểm tra lỗi và khó sửa đổi. Nếu routine có liên kết kém, tốt nhất nên đầu tư viết lại để có liên kết tốt hơn thay vì chỉ cố xác định chính xác vấn đề.

Tuy nhiên, biết đâu là những dạng liên kết nên tránh cũng rất hữu ích:

Procedural cohesion (liên kết theo thủ tục)

Liên kết theo thủ tục xảy ra khi các thao tác trong cùng một routine được thực hiện theo thứ tự xác định. Ví dụ, một routine lần lượt nhận tên nhân viên, địa chỉ, rồi số điện thoại. Thứ tự này chỉ quan trọng vì trùng với thứ tự hiển thị trên màn hình nhập liệu. Một routine khác nhận những thông tin còn lại của nhân viên. Routine này có liên kết theo thủ tục bởi các thao tác phải đúng thứ tự mà không có lý do khác để kết hợp chúng lại.

Để nâng cao liên kết, hãy tách các thao tác này thành các routine riêng biệt. Đảm bảo routine gọi các routine này có một nhiệm vụ hoàn chỉnh:

- Sử dụng `GetEmployee()` thay vì `GetFirstPartOfEmployeeData()`.
  Có thể cần thay đổi cả những routine lấy những thông tin còn lại. Thông thường, phải chỉnh sửa hai hay nhiều routine gốc trước khi đạt được liên kết chức năng đối với bất kỳ routine nào.

Logical cohesion (liên kết theo logic)

Liên kết theo logic xuất hiện khi nhiều thao tác bị nhồi vào cùng một routine, và thao tác nào sẽ được chọn tùy thuộc vào một cờ điều khiển (control flag) truyền vào. Chỉ có luồng điều khiển (logic) mới liên kết các thao tác lại—chúng đều nằm trong một khối lệnh `if` hoặc `case`. Không phải vì các thao tác có liên hệ logic thực sự nào. Với thuộc tính này, thực ra nên gọi là "liên kết phi logic" (illogical cohesion).

Ví dụ, routine `InputAll()` nhập tên khách hàng, thông tin chấm công của nhân viên, hoặc dữ liệu hàng tồn kho tuỳ vào cờ truyền vào routine. Các ví dụ khác: `ComputeAll()`, `EditAll()`, `PrintAll()`, `SaveAll()`.
Vấn đề chính của kiểu routine này là: không nên cần truyền cờ để kiểm soát xử lý của routine khác. Thay vì để một routine thực hiện ba thao tác khác nhau tuỳ theo một cờ truyền vào, tốt hơn nên có ba routine—mỗi routine thực hiện một thao tác khác nhau. Nếu các thao tác này chia sẻ một số đoạn mã hoặc dữ liệu, hãy chuyển chúng xuống routine cấp thấp hơn, rồi đóng gói (package) các routine liên quan vào một class (lớp).

> **Tham khảo thêm:**
>
> Trong một số trường hợp, việc tạo routine có liên kết logic là có thể chấp nhận được nếu routine chủ yếu chỉ chứa các lệnh `if` hoặc `case` để gọi các routine khác. Nếu mục đích duy nhất là phân phối lệnh (dispatch), routine này thường là thiết kế tốt. Thuật ngữ kỹ thuật cho routine dạng này là "event handler" (trình xử lý sự kiện), thường dùng trong môi trường tương tác như Apple Macintosh, Microsoft Windows hoặc các môi trường GUI (Graphical User Interface) khác.
>
> Để biết thêm về việc thay thế các câu lệnh điều kiện bằng tính đa hình (polymorphism), xem phần "Replace conditionals with polymorphism (especially repeated case statements)" trong Mục 24.3.

Coincidental cohesion (liên kết ngẫu nhiên)

Liên kết ngẫu nhiên xảy ra khi các thao tác trong routine không có bất kỳ mối quan hệ nhận diện được nào với nhau. Một số tên gọi phù hợp hơn: "không liên kết", "liên kết hỗn loạn".
Đoạn mã C++ chất lượng thấp được giới thiệu đầu chương là ví dụ của dạng liên kết này.
Thường rất khó chuyển routine có liên kết ngẫu nhiên sang bất kỳ loại liên kết tốt hơn—thường cần thiết kế và hiện thực lại sâu hơn.

Không có thuật ngữ nào là thần thánh

Những thuật ngữ này không phải phép màu hay bất biến, hãy học lấy ý nghĩa thực tế thay vì học thuộc lòng thuật ngữ.
Gần như luôn có thể viết routine với liên kết chức năng (functional cohesion); do đó, hãy chú trọng vào loại này để đạt hiệu quả tối đa.

> **Ý chính:**
> Tập trung vào liên kết chức năng để tối ưu lợi ích khi phát triển routine.

7.3 Đặt tên routine tốt

> **Tham khảo thêm:**
> Để biết chi tiết về đặt tên biến, xem Chương 11: "Sức mạnh của đặt tên biến".

Định hướng đặt tên routine

Một tên routine tốt cần mô tả rõ ràng mọi chức năng mà routine thực hiện. Sau đây là một số hướng dẫn để đặt tên routine hiệu quả:

Mô tả mọi thứ routine thực hiện

Trong tên routine, hãy mô tả đầy đủ các đầu ra và hiệu ứng phụ (side effect). Nếu một routine tính tổng kết báo cáo và đồng thời mở file xuất kết quả, tên `ComputeReportTotals()` là chưa đủ.
Tên `ComputeReportTotalsAndOpenOutputFile()` đủ chi tiết nhưng lại quá dài và khó chịu. Nếu bạn có nhiều routine có hiệu ứng phụ, bạn sẽ có rất nhiều tên dài thừa thãi.
Cách giải quyết không phải là đặt tên ngắn lại, mà là lập trình sao cho các hành động xảy ra trực tiếp, không tạo hiệu ứng phụ gián tiếp.

Tránh động từ mơ hồ, vô nghĩa

Một số động từ quá rộng, gần như mang nghĩa bất kỳ, ví dụ: `HandleCalculation()`, `PerformServices()`, `OutputUser()`, `ProcessInput()`, `DealWithOutput()`. Những tên này không cho biết routine thực sự làm gì; nếu có thì cũng chỉ nói routine có liên quan đến tính toán, dịch vụ, người dùng, nhập, xuất.
Ngoại lệ duy nhất là khi "handle" được dùng trong ý nghĩa kỹ thuật rõ ràng như "handle một event".

Đôi khi vấn đề chỉ là tên routine chưa rõ ràng, bản thân routine thiết kế vẫn tốt. Ví dụ, thay `HandleOutput()` bằng `FormatAndPrintOutput()`, bạn sẽ biết rõ routine thực hiện gì.

> **Ý chính:**
> Nếu tên routine chỉ mơ hồ do hoạt động của routine thiếu rõ ràng thì giải pháp nên là thiết kế lại routine và cách đặt tên liên quan, sao cho mục đích mạnh mẽ, rõ ràng hơn.

Không đặt tên routine chỉ khác nhau bằng số thứ tự

Có lập trình viên viết toàn bộ mã trong một function lớn, sau đó cứ mỗi 15 dòng lại tạo function mới đặt tên là `Part1`, `Part2` v.v. Rồi tạo function cấp cao hơn gọi lần lượt các phần đó. Cách tạo và đặt tên routine này đặc biệt tệ (và hy vọng là hiếm gặp). Tuy nhiên, đôi khi người lập trình cũng dùng số để phân biệt các routine gần giống nhau như `OutputUser`, `OutputUser1`, `OutputUser2`.

(Phần tiếp theo sẽ hướng dẫn chi tiết đổi tên, đặt tên meaningful hơn—nếu cần bạn hỏi tiếp để nhận phần sau.)

Đặt tên cho routine càng dài càng tốt khi cần thiết

Nghiên cứu cho thấy độ dài tối ưu trung bình của một tên biến là từ 9 đến 15 ký tự. Routine (thủ tục/hàm) thường phức tạp hơn biến, và do đó tên gọi phù hợp cho chúng có xu hướng dài hơn. Tuy nhiên, tên routine thường được gắn kèm với tên đối tượng, điều này thực chất đã cung cấp một phần của tên miễn phí. Nhìn chung, khi tạo tên cho routine, điều quan trọng là làm cho tên trở nên rõ ràng nhất có thể, tức là nên đặt tên dài hoặc ngắn tùy ý miễn sao đảm bảo dễ hiểu.

> **Tham khảo thêm:** Để đặt tên cho function, hãy sử dụng mô tả của giá trị trả về. Một function trả về một giá trị, và function nên được đặt tên theo giá trị mà nó trả về. Ví dụ, `cos()`, `customerId.Next()`, `printer.IsReady()`, và `pen.CurrentColor()` đều là những tên function hợp lý, chỉ rõ ràng giá trị mà function trả về.

Để đặt tên cho procedure (thủ tục), hãy bắt đầu bằng một động từ mạnh, sau đó là đối tượng. Một procedure với functional cohesion (liên kết chức năng) thường thực hiện một thao tác trên một đối tượng nào đó. Tên gọi của procedure nên phản ánh hành động mà thủ tục thực hiện, nghĩa là tên nên có dạng động từ+kết hợp với đối tượng. Ví dụ: `PrintDocument()`, `CalcMonthlyRevenues()`, `CheckOrderInfo()`, và `RepaginateDocument()` đều là những tên procedure tốt.

Trong các ngôn ngữ lập trình hướng đối tượng, bạn không cần phải bao gồm tên đối tượng trong tên procedure vì bản thân đối tượng đã được đề cập trong lệnh gọi. Bạn sẽ gọi các routine như: `document.Print()`, `orderInfo.Check()`, và `monthlyRevenues.Calc()`.

Nếu đặt tên ví dụ như `document.PrintDocument()` thì điều này trở nên thừa và có thể gây sai lệch khi áp dụng cho các lớp dẫn xuất. Ví dụ, nếu `Check` là một lớp dẫn xuất từ `Document`, câu lệnh `check.Print()` sẽ rõ ràng là in một séc, trong khi `check.PrintDocument()` nghe như đang in sổ séc hoặc báo cáo tháng, nhưng lại không rõ là in séc.

> **Tham khảo thêm:** Về các cặp thuật ngữ đối lập thường gặp khi đặt tên biến, xem mục “Common Opposite-pairs in Variable Names” tại Phần 11.1.

Sử dụng các cặp từ đối lập một cách chính xác

Việc áp dụng quy ước đặt tên cho các cặp thuật ngữ đối lập giúp tăng tính nhất quán và nâng cao khả năng đọc mã nguồn. Các cặp đối lập như `first/last` là những cặp phổ biến và dễ hiểu. Tuy nhiên, các cặp đối lập như `FileOpen()` và `_lclose()` lại không đối xứng và dễ gây nhầm lẫn. Dưới đây là một số cặp đối lập thông dụng:

- add/remove
- increment/decrement
- open/close
- begin/end
- insert/delete
- show/hide
- create/destroy
- lock/unlock
- source/target
- first/last
- min/max
- start/stop
- get/put
- next/previous
- up/down
- get/set
- old/new

Thiết lập quy ước đặt tên cho các phép toán phổ biến

Trong một số hệ thống, việc phân biệt các loại phép toán là rất quan trọng. Áp dụng quy ước đặt tên là phương pháp dễ dàng và tin cậy nhất để chỉ ra sự khác biệt này.

Ví dụ, trong một dự án, mỗi đối tượng được gán một định danh duy nhất (unique identifier). Tuy nhiên, nhóm phát triển đã không thống nhất quy ước đặt tên routine để trả về identifier của đối tượng, dẫn đến nhiều tên routine như sau:

```
employee id Get()
dependent GetId()
supervisor()
candidate id()
```

Lớp `Employee` truy cập đối tượng `id`, sau đó gọi routine `Get()`. Lớp `Dependent` cung cấp routine `GetId()`. Lớp `Supervisor` trả về giá trị `id` mặc định. Lớp `Candidate` sử dụng mặc định giá trị trả về của đối tượng `id` và truy xuất `id`. Đến giữa dự án, không ai còn nhớ routine nào dùng cho đối tượng nào, nhưng lúc đó quá nhiều code đã được viết rồi nên không thể sửa đồng bộ được nữa. Hệ quả là mỗi thành viên phải dành không ít thời gian để nhớ cú pháp lấy `id` tương ứng với từng class. Nếu có quy ước đặt tên cho routine truy xuất id ngay từ đầu, sự bất tiện này đã có thể tránh được.

7.4 Routine có thể dài đến mức nào?

Khi đến Mỹ, nhóm Pilgrims tranh luận về độ dài tối đa nên có của một routine. Sau cả chuyến đi dài, họ vẫn chưa đi đến thống nhất, và vì thế vấn đề về độ dài tối đa của routine vẫn còn là chủ đề bàn luận bất tận cho đến ngày nay.

Độ dài tối đa lý tưởng của routine thường được mô tả là vừa một màn hình hoặc một đến hai trang tài liệu chương trình, tức khoảng 50 đến 150 dòng. Ví dụ, IBM từng giới hạn routine ở mức 50 dòng, còn TRW giới hạn ở mức hai trang (McCabe 1976). Trong các chương trình hiện đại, thường có rất nhiều routine ngắn đan xen với một số routine dài. Tuy nhiên, routine dài vẫn chưa hề biến mất. Gần đây, tôi từng làm việc tại hai khách hàng, một nơi các lập trình viên đang phải vật lộn với một routine khoảng 4.000 dòng, còn nơi kia là routine hơn 12.000 dòng!

Rất nhiều nghiên cứu đã được thực hiện về độ dài routine, một số kết quả có giá trị đối với các chương trình hiện đại, số khác thì không:

- **Nghiên cứu của Basili và Perricone** cho thấy độ lớn của routine tỷ lệ nghịch với số lỗi: khi độ dài routine tăng lên (đến 200 dòng code), số lỗi trên mỗi dòng code lại giảm (Basili và Perricone 1984).
- Một nghiên cứu khác nhận thấy độ lớn routine không liên quan đến số lỗi, mặc dù độ phức tạp về cấu trúc và lượng dữ liệu lại có liên hệ với số lỗi (Shen et al. 1985).
- Nghiên cứu năm 1986 phát hiện rằng routine nhỏ (32 dòng code trở xuống) không có mối tương quan với chi phí phát triển hoặc tỷ lệ lỗi thấp hơn (Card, Church, và Agresti 1986; Card và Glass 1990). Bằng chứng cho thấy routine lớn hơn (65 dòng trở lên) lại có chi phí phát triển trên mỗi dòng code rẻ hơn.
- Nghiên cứu thực nghiệm trên 450 routine nhận thấy các routine nhỏ (dưới 143 câu lệnh nguồn, kể cả chú thích) có số lỗi trên mỗi dòng code cao hơn 23% so với routine lớn, nhưng chi phí sửa lỗi lại thấp hơn 2,4 lần (Selby và Basili 1991).
- Một nghiên cứu khác cho thấy code cần thay đổi ít nhất khi routine có độ dài trung bình từ 100 đến 150 dòng code (Lind và Vairavan 1989).
- Nghiên cứu tại IBM cho thấy các routine dễ mắc lỗi nhất là routine dài hơn 500 dòng. Vượt quá 500 dòng, tỷ lệ lỗi tỷ lệ thuận với độ dài routine (Jones 1986a).

Routine trong chương trình hướng đối tượng nên dài bao nhiêu?

Đa số routine trong các chương trình hướng đối tượng sẽ là accessor routine, vốn rất ngắn. Tuy nhiên, đôi khi các thuật toán phức tạp sẽ dẫn đến routine dài hơn, và trong những trường hợp này, routine nên phát triển tự nhiên lên đến 100–200 dòng (đếm các dòng mã nguồn không bao gồm comment hoặc dòng trống).

Nhiều thập kỷ nghiên cứu cho thấy routine dài như vậy không dễ mắc lỗi hơn routine ngắn. Các yếu tố như cohesion (tính liên kết), độ sâu lồng nhau, số lượng biến, số điểm quyết định, số lượng comment cần thiết để giải thích routine, và các yếu tố liên quan đến độ phức tạp, cần được dùng để xác định độ dài routine, chứ không nên quy định cứng số dòng.

Tuy nhiên, nếu bạn muốn viết routine dài hơn 200 dòng, cần thận trọng. Không có nghiên cứu nào cho thấy việc tăng kích thước routine lớn hơn 200 dòng lại giúp giảm chi phí hay giảm lỗi. Khi vượt ngưỡng 200 dòng, khả năng hiểu code sẽ bị giới hạn.

7.5 Sử dụng tham số (parameter) trong routine như thế nào?

Các interface (giao diện) giữa các routine là một trong những khu vực dễ phát sinh lỗi nhất trong chương trình. Một nghiên cứu của Basili và Perricone (1984) chỉ ra rằng 39% tổng số lỗi là lỗi giao tiếp nội bộ—lỗi trong việc truyền dữ liệu giữa các routine. Sau đây là một số hướng dẫn nhằm giảm thiểu lỗi khi làm việc với interface:

> **Tham khảo thêm:** Về chú thích tham số của routine, xem “Commenting Routines” tại Mục 32.5. Về định dạng tham số, xem Mục 31.7, “Laying Out Routines”.

Sắp xếp tham số theo thứ tự input–modify–output

Thay vì sắp xếp tham số một cách ngẫu nhiên hoặc theo thứ tự chữ cái, hãy liệt kê các tham số chỉ-dùng-làm-input trước, tiếp theo là các tham số vừa-input-vừa-output, và cuối cùng là chỉ-output. Quy ước này phản ánh trình tự các thao tác trong routine —nhập liệu, thay đổi, và trả về kết quả. Dưới đây là các ví dụ về danh sách tham số trong ngôn ngữ Ada:

```
procedure InvertMatrix(
   originalMatrix: in Matrix;
   resultMatrix: out Matrix
);

procedure ChangeSentenceCase(
   desiredCase: in StringCase;
   sentence: in out Sentence
);

procedure PrintPageNumber(
   pageNumber: in Integer;
   status: out StatusType
);
```

Ada sử dụng các từ khóa `in` và `out` để làm rõ vai trò input/output của tham số.

Quy ước này mâu thuẫn với quy ước trong thư viện C, nơi tham số sẽ bị thay đổi thường được đặt lên đầu. Tuy nhiên, theo tôi, thứ tự input–modify–output có ý nghĩa hơn, và nếu bạn kiên định với một quy ước nào đó, điều này vẫn sẽ giúp người đọc hiểu dễ dàng hơn.

Cân nhắc tạo từ khóa in và out cho riêng bạn

Các ngôn ngữ hiện đại khác như C++ không hỗ trợ từ khóa `in` và `out` như Ada, nhưng vẫn có thể sử dụng preprocessor để tạo các từ khóa này cho mục đích chú thích.

```
#define IN
#define OUT

void InvertMatrix(
   IN Matrix originalMatrix,
   OUT Matrix *resultMatrix
);

void ChangeSentenceCase(
   IN StringCase desiredCase,
   IN OUT Sentence *sentenceToEdit
);

void PrintPageNumber(
   IN int pageNumber,
   OUT StatusType &status
);
```

Ở đây, các macro `IN` và `OUT` chỉ mang ý nghĩa tài liệu/chú thích code.

Lưu ý: Bản dịch đã được hiệu chỉnh về ngôn từ và trình bày để tăng tính dễ hiểu, đồng thời sửa một số lỗi định dạng gốc nhằm bảo đảm văn bản nhất quán và mạch lạc.

Chương 7: Các Thủ Tục (Routine) Chất Lượng Cao

Xem xét các rủi ro khi mở rộng cú pháp ngôn ngữ

Trước khi áp dụng kỹ thuật này, bạn cần cân nhắc hai hạn chế quan trọng. Việc tự định nghĩa các từ khóa IN và OUT sẽ mở rộng ngôn ngữ C++ theo cách không quen thuộc với hầu hết những người đọc mã nguồn của bạn. Nếu bạn quyết định mở rộng ngôn ngữ như vậy, hãy đảm bảo thực hiện một cách nhất quán, tốt nhất là trên toàn dự án. Hạn chế thứ hai là các từ khóa IN và OUT này sẽ không được trình biên dịch kiểm soát, nghĩa là về mặt kỹ thuật, bạn có thể đánh dấu một tham số là IN nhưng vẫn sửa đổi giá trị của nó trong thủ tục. Điều này có thể khiến người đọc mã nguồn nghĩ rằng mã là đúng, dù thực tế không phải vậy. Thông thường, việc sử dụng từ khóa `const` của C++ sẽ là phương pháp tốt hơn để xác định tham số chỉ dùng để nhập (input-only).

Đặt thứ tự tham số một cách nhất quán

Nếu nhiều thủ tục sử dụng các tham số tương tự, hãy đặt các tham số đó theo cùng một thứ tự. Thứ tự các tham số trong thủ tục có thể đóng vai trò như một phương tiện ghi nhớ (mnemonic), và sự thiếu nhất quán trong thứ tự này có thể khiến việc ghi nhớ trở nên khó khăn. Ví dụ, trong ngôn ngữ C, thủ tục `fprintf()` giống hệt với `printf()`, chỉ khác là nó thêm đối tượng file vào vị trí tham số đầu tiên. Một thủ tục tương tự, `fputs()`, lại tương tự với `puts()` nhưng thêm file vào vị trí tham số cuối cùng. Sự khác biệt nhỏ nhưng không cần thiết này khiến cho các tham số của các thủ tục này trở nên khó nhớ hơn mức cần thiết.

Ngược lại, thủ tục `strncpy()` trong C nhận các tham số lần lượt là chuỗi đích, chuỗi nguồn và số byte tối đa; thủ tục `memcpy()` cũng nhận các tham số với thứ tự tương tự. Sự nhất quán này giúp việc ghi nhớ tham số trở nên dễ dàng hơn.

Sử dụng hiệu quả tất cả các tham số

Nếu bạn truyền một tham số vào thủ tục, hãy sử dụng nó. Nếu không dùng đến, hãy loại bỏ tham số đó khỏi khai báo thủ tục. Sự tồn tại của các tham số không được sử dụng thường đi đôi với tỷ lệ lỗi tăng cao. Trong một nghiên cứu, 46% các thủ tục không có biến không sử dụng thì cũng hoàn toàn không có lỗi, trong khi chỉ 17-29% các thủ tục có nhiều hơn một biến không tham chiếu là không có lỗi ([Card, Church, và Agresti, 1986]).

> **Dữ liệu thực tế:** Thủ tục có tham số không sử dụng dễ xuất hiện lỗi chương trình hơn.

Quy tắc loại bỏ tham số không sử dụng có một ngoại lệ: Nếu bạn biên dịch điều kiện một phần chương trình, có thể bạn sẽ loại bỏ các phần của thủ tục sử dụng tham số đó. Hãy thận trọng với thực hành này, nhưng nếu bạn chắc chắn nó hoạt động thì vẫn có thể chấp nhận được. Nói chung, nếu bạn có lý do hợp lý để giữ tham số, hãy để nguyên; còn nếu không, hãy nỗ lực dọn sạch mã nguồn.

Đặt biến trạng thái hoặc lỗi ở cuối danh sách tham số

Theo thông lệ, các biến trạng thái (status) hoặc biến báo lỗi nên đứng cuối trong danh sách tham số. Đây là những tham số chỉ dùng để xuất ra kết quả (output-only) và không phải là mục đích chính của thủ tục, vì vậy việc đặt ở cuối là hợp lý.

Không dùng tham số như biến làm việc

Việc sử dụng tham số truyền vào như biến trung gian cho các phép toán là một thực hành nguy hiểm. Hãy sử dụng biến cục bộ thay thế. Ví dụ dưới đây minh họa sai lầm khi sử dụng tham số làm biến trung gian trong Java:

```
int Sample( int inputVal ) {
    inputVal = inputVal * CurrentMultiplier( inputVal );
    inputVal = inputVal + CurrentAdder( inputVal );
    return inputVal;
}
```

> Tại thời điểm trả về, biến `inputVal` không còn chứa giá trị nhập vào ban đầu nữa.

Trong đoạn mã trên, tên biến `inputVal` trở nên gây hiểu nhầm bởi vì sau các thao tác, nó không còn mang giá trị nhập vào gốc nữa mà mang một giá trị đã được tính toán. Nếu sau này bạn cần sử dụng giá trị nhập gốc ở các vị trí khác, rất có thể bạn sẽ giả định nhầm về nội dung của biến này.

Cách giải quyết

Đổi tên biến `inputVal`? Có thể không giải quyết triệt để! Đặt tên là `workingVal` chỉ giải quyết một phần vì nó không nói rõ nguồn gốc bên ngoài của giá trị. Cách đặt tên như `inputValThatBecomesWorkingVal`, `x`, hay `val` cũng không giúp ích nhiều.

Giải pháp tốt hơn là nên tạo biến trung gian làm việc rõ ràng:

```
int Sample( int inputVal ) {
    int workingVal = inputVal;
    workingVal = workingVal * CurrentMultiplier( workingVal );
    workingVal = workingVal + CurrentAdder( workingVal );
    return workingVal;
}
```

> Nếu cần dùng giá trị gốc `inputVal`, vẫn có thể truy cập được bất cứ lúc nào.

Việc thêm biến `workingVal` làm rõ vai trò của `inputVal` và loại bỏ nguy cơ vô tình dùng nhầm giá trị. (*Lưu ý*: Không nên coi đây là lý do để đặt tên biến thực tế là `inputVal` hay `workingVal`. Trong ví dụ chỉ dùng để minh họa vai trò các biến.)

Việc gán giá trị đầu vào cho biến làm việc giúp nhấn mạnh nguồn gốc dữ liệu, đồng thời loại bỏ khả năng sửa đổi một tham số trong danh sách một cách vô ý. Trong C++, bạn còn có thể yêu cầu trình biên dịch kiểm soát việc này bằng cách sử dụng từ

khóa `const` cho tham số.

Tài liệu hóa giả định giao diện với tham số

Nếu bạn giả định dữ liệu truyền vào thủ tục có đặc điểm nhất định, hãy ghi chú lại ngay khi bạn xác định giả định đó, cả bên trong thủ tục lẫn tại nơi thủ tục được gọi. Đừng đợi đến khi hoàn thành thủ tục mới quay lại bổ sung chú thích bởi bạn sẽ không nhớ hết giả định. Tốt hơn nữa, hãy sử dụng *assertion* (biểu thức khẳng định) để biến giả định thành mã cho chính xác.

> Tham khảo thêm ở Chương 8 (*Defensive Programming*) và Chương 32 (*Self-Documenting Code*).

Một số giả định giao diện cần tài liệu hóa bao gồm:

- Tham số chỉ dùng để nhập (input-only), tham số bị thay đổi (modified), hoặc tham số chỉ dùng để xuất (output-only).
- Đơn vị đo lường của tham số số học (inch, feet, meter, v.v.).
- Ý nghĩa của mã trạng thái (status codes), giá trị lỗi nếu không dùng kiểu liệt kê (enumerated types).
- Dải giá trị mong đợi.
- Giá trị cụ thể không được phép xuất hiện.

Giới hạn số lượng tham số của thủ tục

Bảy là số lượng "thần kỳ" mà con người có thể hiểu cùng lúc. Nghiên cứu tâm lý cho thấy hầu hết mọi người không thể theo dõi hơn bảy "khối" thông tin cùng lúc ([Miller, 1956]). Do đó, số lượng tham số hợp lý của một thủ tục nên nhỏ hơn hoặc bằng bảy.

Trên thực tế, khả năng giới hạn số lượng tham số còn phụ thuộc vào cách ngôn ngữ xử lý kiểu dữ liệu phức tạp. Nếu sử dụng ngôn ngữ hiện đại hỗ trợ dữ liệu có cấu trúc, bạn có thể truyền một kiểu dữ liệu tổng hợp chứa 13 trường mà vẫn chỉ coi là một "khối" thông tin. Nếu dùng ngôn ngữ nguyên thủy hơn, có thể phải truyền từng trường riêng lẻ.

> Tham khảo thêm về thiết kế giao diện ở mục "Good Abstraction" trong Mục 6.2.

Nếu bạn thấy mình thường xuyên truyền nhiều tham số, nghĩa là coupling (liên kết) giữa các thủ tục quá chặt chẽ. Hãy thiết kế lại để giảm coupling; nếu truyền cùng một dữ liệu cho nhiều thủ tục, hãy nhóm các thủ tục đó vào cùng một lớp và để dữ liệu dùng chung làm dữ liệu lớp (class data).

Xem xét quy ước đặt tên tham số theo vai trò

Nếu bạn cần phân biệt tham số đầu vào, thay đổi hoặc đầu ra, hãy áp dụng quy ước tên, ví dụ: đặt tiền tố *i_*, *m_*, *o_*, hoặc verbose hơn là *Input_*, *Modify_*, *Output_*.

Truyền các biến cần thiết để duy trì tính trừu tượng của giao diện

Có hai quan điểm về cách truyền các thành viên (member) của một object (đối tượng) vào thủ tục. Giả sử bạn có một object cung cấp dữ liệu thông qua 10 hàm truy cập (access routine), và thủ tục được gọi cần dùng ba trường dữ liệu này.

1. **Trường phái thứ nhất**: Chỉ nên truyền ba thành phần cụ thể cần thiết. Lý do là để giữ kết nối (connection) giữa các thủ tục ở mức tối thiểu, giảm coupling và làm cho việc hiểu cũng như tái sử dụng dễ dàng hơn. Họ cho rằng truyền cả object sẽ làm lộ ra toàn bộ 10 hàm truy cập cho thủ tục nhận vào, gây vi phạm nguyên tắc đóng gói (encapsulation).

2. **Trường phái thứ hai**: Nên truyền toàn bộ object để giao diện ổn định hơn, cho phép thủ tục được gọi có quyền truy cập bất kỳ thành viên nào nếu cần về sau mà không phải thay đổi giao diện. Họ lập luận rằng truyền ba thành phần cụ thể mới thực sự vi phạm đóng gói, vì làm lộ ra thành phần cụ thể thủ tục đang sử dụng.

Quan điểm trung dung là: yếu tố quyết định là *tính trừu tượng* mà giao diện thủ tục thể hiện. Nếu ý nghĩa trừu tượng là thủ tục chỉ cần ba thành phần cụ thể (dù tình cờ cả ba cùng nằm trong một object), hãy truyền từng thành phần riêng rẽ. Nếu trừu tượng là luôn có một object nhất định dưới tay (và thủ tục sẽ làm việc với toàn thể object đó), thì truyền cả object để không phá vỡ tính trừu tượng đó.

Các Lưu Ý Trong Thiết Kế Và Sử Dụng Routine

> **Ghi chú:** Một số lỗi đánh máy, xuống dòng bất thường trong bản gốc đã được chỉnh sửa lại cho rõ nghĩa.

Thiết Kế Routine Và Việc Sử Dụng Tham Số

Nói chung, nếu phần code dùng để "thiết lập" trước khi gọi một routine, hoặc "thu dọn" sau khi gọi routine xuất hiện thường xuyên, đó là dấu hiệu cho thấy routine đó không được thiết kế tốt.

Nếu bạn thường xuyên phải thay đổi danh sách tham số (parameter list) của routine, và các tham số đó đều lấy từ cùng một đối tượng, đây là dấu hiệu cho thấy bạn nên truyền cả đối tượng thay vì truyền từng phần tử cụ thể.

Sử Dụng Tham Số Đặt Tên (Named Parameter)

Ở một số ngôn ngữ lập trình, bạn có thể liên kết rõ ràng các tham số hình thức với tham số thực tế, điều này giúp code tự giải thích hơn cũng như tránh lỗi do nhầm lẫn thứ tự tham số.

Ví dụ trong Visual Basic:

```
Private Function Distance3d( _
    ByVal xDistance As Coordinate, _
    ByVal yDistance As Coordinate, _
    ByVal zDistance As Coordinate _
)
End Function

Private Function Velocity( _
    ByVal latitude As Coordinate, _
    ByVal longitude As Coordinate, _
    ByVal elevation As Coordinate _
)
    Distance = Distance3d(xDistance := latitude, yDistance := longitude, zDistance := elevation)
End Function
```

Kỹ thuật này đặc biệt hữu ích khi bạn có danh sách tham số dài với cùng kiểu dữ liệu, điều này dễ dẫn đến nhầm lẫn khi chèn nhầm tham số mà trình biên dịch không phát hiện được. Việc liên kết rõ ràng tham số có thể là dư thừa trong nhiều môi trường, nhưng đối với phần mềm an toàn hay đòi hỏi tính tin cậy cao, sự đảm bảo này rất đáng giá.

Đảm Bảo Tham Số Thực Khớp Với Tham Số Hình Thức

Tham số hình thức (formal parameter), còn gọi là “tham số giả”, là các biến được khai báo trong định nghĩa routine. Tham số thực (actual parameter) là biến, hằng số, hoặc biểu thức được sử dụng khi gọi routine.

Một lỗi phổ biến là truyền nhầm kiểu biến vào routine — ví dụ, dùng kiểu integer khi cần floating point. (Vấn đề này chủ yếu xuất hiện ở các ngôn ngữ kiểu yếu như C, đặc biệt khi không sử dụng đầy đủ cảnh báo biên dịch. Ở các ngôn ngữ kiểu mạnh như C++ hay Java, vấn đề này ít gặp hơn.) Đối với tham số chỉ dùng để truyền vào (input only), thông thường trình biên dịch sẽ tự động chuyển đổi kiểu, nhưng nếu là tham số dùng cho cả đầu vào và đầu ra thì lỗi truyền nhầm kiểu có thể gây ra hậu quả nghiêm trọng.

Vì vậy, hãy rèn luyện thói quen kiểm tra kỹ kiểu của các đối số trong danh sách tham số, đồng thời luôn chú ý các cảnh báo của trình biên dịch liên quan đến kiểu tham số không khớp.

7.6 Những Lưu Ý Đặc Biệt Khi Sử Dụng Function

Ngôn ngữ hiện đại như C++, Java, và Visual Basic hỗ trợ cả function và procedure. **Function** là routine trả về giá trị; **procedure** là routine không trả về giá trị. Trong C++, tất cả các routine thường được gọi là "function"; nhưng function có kiểu trả về void về mặt ngữ nghĩa lại là procedure. Việc phân biệt giữa function và procedure không chỉ ở mặt cú pháp mà còn ở mặt ngữ nghĩa, và bạn nên đặt trọng tâm vào ý nghĩa (semantic) khi lựa chọn.

Khi Nào Dùng Function, Khi Nào Dùng Procedure

Theo quan điểm "thuần túy", function chỉ nên trả về một giá trị duy nhất và chỉ nhận tham số đầu vào, đặt tên theo giá trị trả về (như `sin()`, `CustomerID()`, `ScreenHeight()`). Procedure, ngược lại, có thể nhận/hiệu chỉnh/thay đổi đầu vào, đầu ra, hoặc cả hai.

Một thực tiễn phổ biến là function vừa thực hiện hành động như một procedure, vừa trả về giá trị trạng thái. Ví dụ:

```
if (report.FormatOutput(formattedReport) = Success) then
```

Trong ví dụ này, `report.FormatOutput()` vừa có tham số đầu ra `formattedReport`, vừa trả về kết quả kiểm tra trạng thái. Về nguyên tắc, đây là function nhưng hoạt động giống procedure. Việc sử dụng giá trị trả về để xác định thành công/thất bại là hợp lý nếu bạn giữ sự nhất quán.

Một cách khác là tạo procedure với biến trạng thái như tham số xuất, ví dụ:

```
report.FormatOutput(formattedReport, outputStatus)
if (outputStatus = Success) then
```

Hoặc:

```
outputStatus = report.FormatOutput(formattedReport)
if (outputStatus = Success) then
```

Tóm lại, hãy sử dụng function nếu mục đích chính là trả về giá trị như tên function; còn lại nên dùng procedure.

> **Ý chính:** Nếu routine chủ yếu trả về giá trị được nêu rõ ở tên function, hãy dùng function; nếu không, hãy dùng procedure.

Đặt Giá Trị Trả Về Cho Function

Sử dụng function tiềm ẩn nguy cơ function trả về giá trị sai. Thông thường, điều này xảy ra khi trong function có nhiều đường dẫn (path) thực thi mà một trong số đó không gán giá trị trả về.

- **Kiểm tra mọi đường dẫn trả về:** Khi viết function, hãy hình dung và kiểm thử mọi đường dẫn để đảm bảo luôn gán đủ giá trị trả về.
- **Khởi tạo giá trị mặc định cho biến trả về:** Nên khởi tạo biến trả về ngay từ đầu để tránh trường hợp thiếu gán giá trị.
- **Không trả về reference hoặc pointer tới dữ liệu cục bộ:** Khi routine kết thúc, dữ liệu cục bộ sẽ mất hiệu lực, đồng nghĩa mọi reference hoặc pointer tới chúng đều invalid. Nếu cần trả thông tin về dữ liệu nội bộ, hãy lưu trong thành viên của class (class member), sau đó cung cấp accessor function để truy xuất giá trị đó.

7.7 Macro Routine và Inline Routine

> **Tham khảo thêm:** Ngay cả khi ngôn ngữ lập trình của bạn không hỗ trợ, bạn có thể tự xây dựng các macro routine (routine được tạo bằng preprocessor macro). Tham khảo chi tiết ở mục 30.5 “Building Your Own Programming Tools”.

Đặt Dấu Ngoặc Đầy Đủ Cho Macro

Bởi vì macro và đối số của nó sẽ được mở rộng thành code, bạn phải cẩn thận để đảm bảo chúng được mở rộng đúng như mong đợi. Một lỗi phổ biến là viết macro như sau:

```
#define Cube(a) a*a*a
```

Nếu truyền giá trị không nguyên tử cho `a`, phép nhân sẽ không ra kết quả đúng. Ví dụ, `Cube(x+1)` sẽ mở rộng thành `x+1 * x + 1 * x + 1`, không đúng ý định do thứ tự ưu tiên toán tử.

Phiên bản cải tiến như sau (nhưng vẫn chưa hoàn hảo):

```
#define Cube(a) (a)*(a)*(a)
```

Cách tốt nhất là đặt toàn bộ biểu thức trong dấu ngoặc:

```
#define Cube(a) ((a)*(a)*(a))
```

Đặt Dấu Ngoặc Nhọn Cho Macro Nhiều Lệnh

Macro có thể chứa nhiều lệnh, nếu coi như một lệnh duy nhất sẽ gây lỗi, ví dụ:

```
#define LookupEntry(key, index) \
    index = (key - 10) / 5; \
    index = min(index, MAX_INDEX); \
    index = max(index, MIN_INDEX);

for (entryCount = 0; entryCount < numEntries; entryCount++)
    LookupEntry(entryCount, tableIndex[entryCount]);
```

Chỉ có dòng đầu tiên của macro được thực thi trong vòng lặp. Để tránh lỗi, hãy đặt dấu ngoặc nhọn:

```
#define LookupEntry(key, index) { \
    index = (key - 10) / 5; \
    index = min(index, MAX_INDEX); \
    index = max(index, MIN_INDEX); \
}
```

Việc sử dụng macro thay thế cho function call nhìn chung được đánh giá là nguy hiểm, khó hiểu, và là thực hành lập trình không tốt. Chỉ nên dùng kỹ thuật này khi hoàn cảnh thực sự đòi hỏi.

Quy Tắc Đặt Tên Macro

Đối với macro mở rộng thành code (thay thế function), nên đặt tên giống tên function để có thể thay thế lẫn nhau mà không phải sửa đổi code nơi sử dụng. Ở C++, quy ước là đặt tên macro bằng chữ in hoa. Tuy nhiên, nếu macro có thể được thay thế bằng routine, hãy sử dụng quy ước đặt tên cho routine để thuận tiện chuyển đổi về sau.

> **Lưu ý:** Áp dụng các khuyến nghị trên kèm với cân nhắc rủi ro đặc thù của dự án và ngôn ngữ lập trình bạn đang sử dụng.

Hạn chế trong việc sử dụng Macro Routine

Các vấn đề liên quan đến tác dụng phụ

> Cân nhắc đến các vấn đề khác do tác dụng phụ (side effects), đây là một lý do nữa để tránh sử dụng tác dụng phụ.

Những hạn chế khi sử dụng Macro Routine

Các ngôn ngữ hiện đại như C++ cung cấp nhiều lựa chọn thay thế cho việc sử dụng macro, bao gồm:

- `const` để khai báo giá trị hằng số.
- `inline` để định nghĩa hàm sẽ được biên dịch thành mã nội tuyến.
- `template` để định nghĩa các phép toán tiêu chuẩn như `min`, `max`,... theo cách đảm bảo an toàn kiểu dữ liệu (type-safe).
- `enum` để khai báo các kiểu liệt kê (enumerated types).
- `typedef` để định nghĩa thay thế kiểu dữ liệu đơn giản.

Như Bjarne Stroustrup, nhà thiết kế của C++ đã chỉ ra:

> "Hầu hết mọi macro đều thể hiện một khiếm khuyết của ngôn ngữ lập trình, của chương trình, hoặc của lập trình viên. Khi bạn sử dụng macro, bạn nên mong đợi chất lượng phục vụ kém hơn từ các công cụ chẳng hạn như debugger (trình gỡ lỗi), cross-reference tools (công cụ tham chiếu chéo), và profiler (công cụ phân tích hiệu năng)" (Stroustrup 1997).

Macro hữu ích trong việc hỗ trợ biên dịch có điều kiện (conditional compilation)—xem thêm ở Mục 8.6, "Debugging Aids"—nhưng các lập trình viên cần trọng chỉ sử dụng macro thay cho routine (thủ tục/hàm) như là giải pháp cuối cùng.

Inline Routine

Ngôn ngữ C++ hỗ trợ từ khóa `inline`. Một inline routine cho phép lập trình viên xử lý mã như một routine khi viết mã, tuy nhiên trình biên dịch thường sẽ chuyển hóa mỗi lần gọi routine thành mã nội tuyến khi biên dịch. Theo lý thuyết, `inline` có thể giúp tạo ra code hiệu quả bằng cách loại bỏ chi phí gọi routine.

Cẩn trọng khi sử dụng inline routine

- Sử dụng inline routine một cách tiết kiệm.
- Inline routine vi phạm nguyên tắc đóng gói (encapsulation), bởi C++ yêu cầu đặt phần mã hiện thực của inline routine vào tập tin header, lộ ra cho mọi lập trình viên sử dụng tập tin đó.
- Inline routine đòi hỏi toàn bộ mã routine được sinh ra mỗi lần routine được gọi, với inline routine có kích cỡ lớn sẽ làm tăng kích thước mã nguồn, kéo theo các vấn đề riêng.

  Kết luận về việc tối ưu hiệu năng nhờ inlining giống như mọi kỹ thuật tối ưu khác: cần thực hiện profile và đo lường sự cải thiện. Nếu lợi ích hiệu năng dự kiến không đủ lớn để biện minh cho việc profile kiểm chứng, nó cũng không xứng với việc đánh đổi chất lượng mã.

DANH SÁCH KIỂM TRA: Routine Chất lượng Cao

Các vấn đề tổng thể

- ❑ Lý do tạo routine có đủ thỏa đáng?
- ❑ Tất cả thành phần của routine nào nên tách thành routine riêng đã được xử lý thích hợp chưa?

- ❑ Tên routine có phải dạng động từ-kèm-tân ngữ rõ ràng cho procedure (thủ tục) hoặc mô tả giá trị trả về cho function (hàm) không?
- ❑ Tên routine có mô tả đầy đủ những gì routine làm không?
- ❑ Đã thống nhất quy ước đặt tên cho các thao tác phổ biến chưa?
- ❑ Routine đảm bảo tính cohesion chức năng (functional cohesion) mạnh, chỉ làm một việc tốt nhất chưa?
- ❑ Các routine có loose coupling (liên kết lỏng lẻo)—liên kết giữa các routine nhỏ gọn, trực quan, và linh hoạt không?
- ❑ Độ dài routine được xác định tự nhiên theo chức năng, không bị gò ép bởi tiêu chuẩn mã hóa cứng nhắc?

Các vấn đề về việc truyền tham số

- ❑ Danh sách tham số mang đến abstraction interface (giao diện trừu tượng) nhất quán chưa?
- ❑ Tham số có thứ tự hợp lý, tương tự đối với những routine cùng loại?
- ❑ Đã ghi chú các giả định về giao diện chưa?
- ❑ Routine có tối đa bảy tham số trở xuống không?
- ❑ Mỗi tham số đầu vào đều được sử dụng?
- ❑ Mỗi tham số đầu ra đều được sử dụng?
- ❑ Routine tránh dùng tham số đầu vào làm biến làm việc chưa?
- ❑ Nếu là function, hàm có trả về giá trị hợp lệ trong mọi trường hợp không?

Các điểm chính

- **Lý do quan trọng nhất để tạo ra routine là tăng khả năng kiểm soát trí tuệ (intellectual manageability) của chương trình**; có nhiều lý do tốt khác, trong đó tiết kiệm bộ nhớ chỉ là phụ, còn tăng khả năng đọc, tin cậy và khả năng chỉnh sửa là các lý do quan trọng hơn.
- **Đôi khi thao tác đơn giản nhất lại cần tách ra thành routine riêng.**
- **Routine có thể được phân loại theo nhiều mức độ cohesion (độ gắn kết), nhưng hầu hết nên đạt cohesion chức năng (functional cohesion), đó là tốt nhất.**
- **Tên routine phản ánh chất lượng của nó; tên tệ nhưng đúng thì routine có thể thiết kế kém, tên tệ mà sai nghĩa thì nó không cho biết routine làm gì—dù thế nào, tên routine tệ nghĩa là phải sửa mã.**
- **Chỉ nên dùng function khi mục tiêu chính là trả về giá trị cụ thể theo tên hàm.**
- **Lập trình viên cẩn trọng chỉ dùng macro routine khi cần thiết và cuối cùng.**

Chương 8: Lập Trình Phòng Thủ (Defensive Programming)

Nội dung

Chủ đề liên quan

- **Information hiding (che giấu thông tin)**: Xem "Hide Secrets (Information Hiding)" tại Mục 5.3
- **Thiết kế cho sự thay đổi**: "Identify Areas Likely to Change" tại Mục 5.3
- **Kiến trúc phần mềm**: Mục 3.5
- **Thiết kế trong xây dựng**: Chương 5
- **Gỡ lỗi**: Chương 23

Giới thiệu về Lập trình Phòng thủ

Lập trình phòng thủ không có nghĩa là phòng thủ chống lại chỉ trích về mã—kiểu “Tôi làm đúng rồi mà!”. Ý tưởng này dựa trên nguyên tắc của defensive driving (lái xe phòng thủ). Trong lái xe phòng thủ, bạn luôn chuẩn bị tinh thần rằng không thể đoán trước người khác sẽ làm gì. Nhờ vậy, nếu người khác làm sai, bạn vẫn hạn chế được rủi ro. Bạn chịu trách nhiệm bảo vệ chính mình, dù lỗi thuộc về người khác.

> **Điểm then chốt:** Trong lập trình phòng thủ, nguyên tắc cốt lõi là nếu routine nhận dữ liệu sai, nó sẽ không gặp sự cố, dù lỗi do routine khác gây ra. Nói rộng hơn, đây là sự thừa nhận rằng chương trình luôn có vấn đề và cần sửa đổi, vì vậy lập trình viên thông minh sẽ mã hóa chương trình tương ứng.

Chương này sẽ hướng dẫn cách bảo vệ bạn trước dữ liệu không hợp lệ, những sự kiện "không bao giờ xảy ra", và lỗi của những lập trình viên khác. Nếu bạn đã có kinh nghiệm, có thể bỏ qua phần tiếp theo về xử lý dữ liệu vào và bắt đầu từ mục 8.2 về assertion.

8.1 Bảo vệ Chương trình khỏi Dữ liệu Vào Không hợp lệ

Khi học ở trường, bạn có thể đã nghe câu "Garbage in, garbage out" (Đầu vào rác, đầu ra rác). Đây cũng chính là dạng caveat emptor (cảnh báo người dùng) trong phát triển phần mềm. Tuy nhiên, đối với phần mềm sản xuất, "garbage in, garbage out" không còn phù hợp. Một chương trình tốt không bao giờ xuất ra dữ liệu rác, bất kể đầu vào nhận được. Một chương trình tốt sẽ dùng các chiến lược như "garbage in, nothing out", "garbage in, error message out", hoặc "no garbage allowed in" (không cho phép đầu vào rác).

> **Điểm then chốt:** Với tiêu chuẩn ngày nay, "garbage in, garbage out" cho thấy một chương trình lỏng lẻo, thiếu an toàn.

Có ba cách tổng quát để xử lý dữ liệu rác:

1. **Kiểm tra giá trị toàn bộ dữ liệu từ nguồn bên ngoài**
   Khi nhận dữ liệu từ file, người dùng, mạng, hoặc nguồn ngoại lai khác, cần kiểm tra đảm bảo dữ liệu nằm trong phạm vi chấp nhận được:
   - Đảm bảo giá trị kiểu số nằm trong phạm vi cho phép.
   - Chuỗi không quá dài.
   - Nếu chuỗi đại diện cho giá trị hạn chế (ví dụ: mã giao dịch tài chính…), phải đảm bảo chuỗi hợp lệ; nếu không, loại bỏ giá trị đó.
   - Nếu làm việc với ứng dụng an toàn, cần đặc biệt đề phòng các kiểu tấn công như:
     - Buffer overflow (tràn bộ đệm)
     - Injected SQL commands (lệnh SQL chèn độc hại)
     - Injected HTML/XML code (mã HTML hoặc XML chèn độc hại)
     - Integer overflow (tràn số nguyên)
     - Dữ liệu truyền tới system calls (lệnh hệ thống)
2. **Kiểm tra giá trị toàn bộ tham số đầu vào của routine** Việc kiểm tra này tương tự kiểm tra dữ liệu từ nguồn ngoài, chỉ khác là dữ liệu đến từ routine khác trong hệ thống chứ không phải giao diện bên ngoài. Mục 8.5, "Barricade Your Program to Contain the Damage Caused by Errors," bàn chi tiết cách xác định những routine nào cần kiểm tra đầu vào.
3. **Quyết định cách xử lý đầu vào không hợp lệ** Sau khi phát hiện tham số không hợp lệ, bạn sẽ xử lý ra sao? Tùy trường hợp có thể lựa chọn một trong nhiều biện pháp, các phương pháp này được trình bày chi tiết tại Mục 8.3, "Error-Handling Techniques."

Lập trình phòng thủ là phương án hỗ trợ thêm cho các kỹ thuật nâng cao chất lượng được đề cập trong sách. Hình thức tốt nhất của coding phòng thủ là **không làm phát sinh lỗi ngay từ đầu**: thiết kế lặp, viết giả mã trước khi viết mã, viết test case trước khi viết code, kiểm tra thiết kế ở mức thấp… đều là những hoạt động giúp phòng tránh lỗi; chúng cần được ưu tiên cao hơn so với lập trình phòng thủ. Tuy vậy, bạn hoàn toàn có thể phối hợp các kỹ thuật này với phòng thủ để đạt kết quả tốt nhất.

Như Hình 8-1 minh họa, việc bảo vệ mình trước các vấn đề tưởng chừng nhỏ nhặt có thể tạo ra khác biệt lớn hơn bạn nghĩ. Phần còn lại của chương sẽ trình bày cụ thể các tùy chọn kiểm tra dữ liệu từ nguồn ngoài, kiểm tra tham số đầu vào và xử lý dữ liệu không hợp lệ.

> **Hình 8-1:** Một phần của cầu nổi Interstate-90 tại Seattle đã bị chìm trong bão do các bể nổi không được che chắn, dẫn đến việc nước tràn vào và khiến cầu chìm. Trong quá trình xây dựng, việc bảo vệ mình trước những chi tiết nhỏ lại là điều quan trọng hơn bạn nghĩ.

Khi một assertion (khẳng định) là đúng

Khi một assertion là đúng, điều này có nghĩa mọi thành phần của hệ thống đang hoạt động như mong đợi. Khi assertion sai, điều đó cho thấy đã phát hiện một lỗi bất ngờ trong mã. Ví dụ, nếu hệ thống giả định rằng tệp thông tin khách hàng sẽ không bao giờ có nhiều hơn 50.000 bản ghi, chương trình có thể chứa một assertion kiểm tra rằng số lượng bản ghi phải nhỏ hơn hoặc bằng 50.000. Khi số bản ghi nhỏ hơn hoặc bằng 50.000, assertion sẽ không đưa ra cảnh báo. Tuy nhiên, nếu gặp nhiều hơn 50.000 bản ghi, assertion sẽ thông báo mạnh mẽ rằng có lỗi trong chương trình.

Assertions đặc biệt hữu ích trong các chương trình lớn, phức tạp và các chương trình yêu cầu độ tin cậy cao. Chúng giúp lập trình viên nhanh chóng phát hiện các giả định không nhất quán giữa các interface (giao diện), những lỗi phát sinh khi mã bị

thay đổi, v.v.

Thông thường, một assertion nhận hai đối số: một biểu thức boolean (đúng/sai) mô tả giả định cần phải đúng, và một thông báo sẽ được hiển thị nếu giả định đó không đúng. Ví dụ về assertion trong Java, khi giả sử biến `denominator` (mẫu số) phải khác không:

```
assert denominator != 0 : "denominator is unexpectedly equal to 0 ";
```

Assertion trên xác nhận rằng `denominator` không bằng 0. Đối số thứ nhất `denominator != 0` là biểu thức boolean. Đối số thứ hai là thông báo sẽ in ra nếu assertion sai, tức là nếu điều kiện kiểm tra không đúng.

> **Lưu ý quan trọng:** Hãy sử dụng assertion để ghi nhận các giả định trong mã và phát hiện các tình huống bất ngờ.

Các ví dụ về giả định có thể kiểm tra bằng assertion

- Giá trị của một input parameter (tham số đầu vào) nằm trong phạm vi dự kiến (hoặc của output parameter).
- Một file hoặc stream mở (hoặc đóng) khi một routine (thủ tục) bắt đầu (hoặc kết thúc) thực thi.
- File hoặc stream ở đầu (hoặc cuối) khi routine bắt đầu (hoặc kết thúc).
- File hoặc stream mở theo chế độ chỉ đọc, chỉ ghi, hoặc vừa đọc vừa ghi.
- Giá trị của một biến chỉ đọc không bị thay đổi trong routine.
- Một pointer (con trỏ) khác null.
- Mảng hoặc container truyền vào một routine có thể chứa tối thiểu X phần tử dữ liệu.
- Bảng đã được khởi tạo với các giá trị thực tế.
- Container rỗng (hoặc đầy) khi routine bắt đầu (hoặc kết thúc).
- Kết quả từ một routine phức tạp, tối ưu hóa mạnh khớp với kết quả từ một routine chậm hơn nhưng được viết rõ ràng.

Đây chỉ là các ví dụ cơ bản; các routine của bạn sẽ còn có nhiều giả định cụ thể khác có thể ghi nhận bằng assertion.

Thời điểm sử dụng assertion

Thông thường, bạn không muốn người dùng nhìn thấy các thông báo assertion trong mã vận hành thực tế (production code); assertion chủ yếu phục vụ trong quá trình phát triển và bảo trì phần mềm. Các assertion thường được biên dịch vào mã khi phát triển và loại bỏ khỏi mã khi đóng gói cho sản xuất, nhằm tránh ảnh hưởng đến hiệu năng hệ thống.

Xây dựng cơ chế assertion riêng

Nhiều ngôn ngữ lập trình như C++, Java, Microsoft Visual Basic đều hỗ trợ assertion. Nếu ngôn ngữ bạn sử dụng không hỗ trợ trực tiếp, việc xây dựng routine assertion là khá đơn giản. Ví dụ, macro `assert` chuẩn trong C++ không hỗ trợ thông báo chi tiết. Sau đây là phiên bản mở rộng của macro ASSERT bằng C++:

```
#define ASSERT( condition, message ) { \
    if ( !(condition) ) { \
        LogError( "Assertion failed: ", \
        #condition, message ); \
        exit( EXIT_FAILURE ); \
    } \
}
```

Hướng dẫn sử dụng assertion

1. Phân biệt giữa assertion và xử lý lỗi

- **Sử dụng mã xử lý lỗi** cho các tình huống bạn dự đoán sẽ xảy ra.
- **Sử dụng assertion** cho các tình huống không bao giờ được xảy ra.

Mã xử lý lỗi kiểm tra các trường hợp dữ liệu không hợp lệ mà lập trình viên đã lường trước và cần được xử lý trong sản phẩm. Assertion thường dùng để kiểm tra các lỗi logic trong mã.

Nếu sử dụng mã xử lý lỗi cho điều kiện bất thường, chương trình có thể phục hồi gracefully (mượt mà). Nếu assertion được kích hoạt, hành động khắc phục không chỉ là xử lý lỗi mà là sửa mã nguồn, biên dịch lại và phát hành phần mềm mới.

Assertion là một dạng tài liệu khả thực (executable documentation) giúp ghi nhận các giả định, chủ động hơn so với lời chú thích (comment) trong code.

2. Không đặt mã khả thực vào assertion

Nếu bạn đặt lệnh thực thi vào assertion, khi loại bỏ assertion lúc biên dịch cho môi trường sản xuất, bạn cũng loại bỏ cả lệnh thực thi đó. Thay vào đó, hãy viết mã khả thực trên dòng riêng, gán kết quả cho biến trạng thái và kiểm tra biến đó bằng assertion, ví dụ:

```
actionPerformed = PerformAction()
Debug Assert( actionPerformed )  ' Không thể thực hiện hành động
```

3. Sử dụng assertion để ghi nhận và kiểm tra precondition và postcondition

Precondition (tiền điều kiện) và postcondition (hậu điều kiện) là thành phần quan trọng trong thiết kế phần mềm, còn gọi là "design by contract" (thiết kế dựa trên hợp đồng).

- **Precondition** là các điều kiện mà phía client (mã gọi procedure hoặc class) cam kết đúng trước khi gọi.
- **Postcondition** là các điều kiện mà routine hoặc class đảm bảo sẽ đúng khi kết thúc thực thi.

Assertion là công cụ hữu ích để kiểm tra động các precondition và postcondition, thay vì chỉ ghi chú bằng comment.

Ví dụ sử dụng assertion để kiểm tra precondition và postcondition trong Visual Basic:

```
Private Function Velocity ( _
    ByVal latitude As Single, _
    ByVal longitude As Single, _
    ByVal elevation As Single _
) As Single
    ' Preconditions
    Debug Assert ( -90 <= latitude And latitude <= 90 )
    Debug Assert ( 0 <= longitude And longitude < 360 )
    Debug Assert ( -500 <= elevation And elevation <= 75000 )

    ' Postcondition
    Debug Assert ( 0 <= returnVelocity And returnVelocity <= 600 )
    ' return value
    Velocity = returnVelocity
End Function
```

Nếu các biến `latitude`, `longitude`, `elevation` đến từ nguồn bên ngoài, hãy kiểm tra giá trị bất hợp lệ bằng mã xử lý lỗi (error-handling code) thay vì assertion. Nếu các giá trị này đến từ nguồn tin cậy nội bộ và routine giả định chúng luôn hợp lệ, assertion là phù hợp.

4. Kết hợp cả assertion và mã xử lý lỗi khi cần thiết

Trong các hệ thống lớn, phức tạp do nhiều nhóm thiết kế, phát triển trong thời gian dài và môi trường khác nhau, đôi khi cần sử dụng cả assertion lẫn xử lý lỗi cho cùng một loại lỗi để đảm bảo độ chắc chắn và an toàn cho phần mềm.

Tự kiểm tra:
Bản dịch đảm bảo giữ nguyên các đoạn code, sử dụng đúng các thuật ngữ chuyên ngành, phân đoạn rõ ràng và ngắn gọn, logic liền mạch với văn phong học thuật, trang trọng. Các chú thích bổ sung đã được lồng ghép tại vị trí thích hợp để làm rõ nội dung gốc.

Ứng dụng lớn, phức tạp và lâu đời: Vai trò của assertion (khẳng định)

Đối với các ứng dụng cực kỳ lớn, phức tạp và có tuổi thọ kéo dài như Word, assertion (khẳng định) rất có giá trị vì chúng giúp loại bỏ càng nhiều lỗi xuất hiện trong quá trình phát triển càng tốt. Tuy nhiên, với độ phức tạp lớn (hàng triệu dòng mã lệnh) và đã qua nhiều thế hệ sửa đổi, sẽ không thực tế nếu giả định rằng mọi lỗi có thể đều được phát hiện và khắc phục trước khi phần mềm được phát hành. Vì thế, các lỗi phát sinh cũng cần được xử lý trong phiên bản sản xuất (production) của hệ thống.

Ví dụ sử dụng assertion trong Velocity

> Dưới đây là ví dụ minh họa cho cách sử dụng assertion và xử lý lỗi đầu vào trong ngôn ngữ Visual Basic, áp dụng cho hàm Velocity:

```
' Visual Basic Example of Using Assertions to Document Preconditions and Postconditions
Private Function Velocity ( _
    ByRef latitude As Single, _
    ByRef longitude As Single, _
    ByRef elevation As Single _
) As Single
    ' Preconditions
    Debug Assert ( -90 <= latitude And latitude <= 90 )
    Debug Assert ( 0 <= longitude And longitude < 360 )
    Debug Assert ( -500 <= elevation And elevation <= 75000 )

    ' Sanitize input data. Values should be within the ranges asserted above,
    ' but if a value is not within its valid range, it will be changed to the
    ' closest legal value.
```

```
    If ( latitude < -90 ) Then
        latitude = -90
    ElseIf ( latitude > 90 ) Then
        latitude = 90
    End If
    If ( longitude < 0 ) Then
        longitude = 0
    ElseIf ( longitude > 360 ) Then
    ' ...
End Function
```

Trong ví dụ này, assertion được sử dụng để ghi chú các điều kiện tiên quyết (precondition) và hậu điều kiện (postcondition). Ngoài ra, mã nguồn cũng có đoạn xử lý "sanitize input data" nhằm hiệu chỉnh giá trị đầu vào về giá trị hợp lệ gần nhất nếu phát hiện dữ liệu sai phạm.

8.3 Các kỹ thuật xử lý lỗi

Assertion được dùng để xử lý những lỗi lẽ ra không nên xảy ra trong mã nguồn. Vậy với các lỗi dự kiến có thể xảy ra, chúng ta nên xử lý thế nào? Tùy vào từng tình huống, bạn có thể áp dụng các biện pháp sau:

- Trả về giá trị trung tính (neutral value)
- Thay thế bằng phần dữ liệu hợp lệ tiếp theo
- Trả về kết quả giống như lần trước
- Thay thế bằng giá trị hợp lệ gần nhất
- Ghi lại thông báo cảnh báo vào file log
- Trả về mã lỗi (error code)
- Gọi một routine/object xử lý lỗi
- Hiển thị thông báo lỗi cho người dùng
- Tắt chương trình
- Hoặc kết hợp các biện pháp trên

Các phương án xử lý lỗi chi tiết

1. Trả về giá trị trung tính (Return a neutral value)

Đôi khi cách ứng phó tốt nhất với dữ liệu sai là tiếp tục hoạt động và trả về giá trị được biết là vô hại. Ví dụ:

- Tính toán số học trả về 0
- Hàm thao tác chuỗi trả về chuỗi rỗng
- Hàm thao tác con trỏ trả về con trỏ rỗng

Một routine vẽ trong trò chơi video nhận màu sai có thể sử dụng màu nền hoặc màu mặc định. Tuy nhiên, đối với routine hiển thị dữ liệu chụp X-quang cho bệnh nhân ung thư, việc trả về giá trị "trung tính" là không phù hợp; trường hợp này nên kết thúc chương trình thay vì hiển thị dữ liệu bệnh nhân sai.

---

2. Thay thế bằng phần dữ liệu hợp lệ tiếp theo (Substitute the next piece of valid data)

Khi xử lý luồng dữ liệu, đôi khi chỉ cần trả về phần dữ liệu hợp lệ tiếp theo. Ví dụ:

- Khi đọc bản ghi từ cơ sở dữ liệu và gặp bản ghi bị lỗi, chỉ cần tiếp tục cho đến khi gặp bản ghi hợp lệ.
- Lấy dữ liệu từ nhiệt kế 100 lần mỗi giây, nếu có lần không lấy được số đo hợp lệ thì chỉ cần chờ 1/100 giây và lấy số liệu tiếp theo.

---

3. Trả về kết quả giống lần trước (Return the same answer as the previous time)

Nếu phần mềm đo nhiệt độ không lấy được số liệu, có thể trả về giá trị lần đo trước, vì xác suất nhiệt độ thay đổi nhiều trong 1/100 giây là thấp. Trong trò chơi, khi nhận yêu cầu hiển thị màu không hợp lệ, cũng có thể dùng lại màu lần trước. Tuy nhiên, với các ứng dụng như máy rút tiền (ATM), việc dùng lại kết quả trước đó (chẳng hạn trả về số tài khoản của khách lần trước) là không thể chấp nhận.

---

4. Thay thế bằng giá trị hợp lệ gần nhất (Substitute the closest legal value)

Một phương án hợp lý là trả về giá trị hợp lệ gần nhất, như ví dụ về Velocity ở trên. Khi cảm ứng nhiệt độ chỉ đo được từ 0 đến 100 độ C, nếu số đo dưới 0 sẽ trả về 0; nếu lớn hơn 100 sẽ trả về 100. Đối với thao tác chuỗi, nếu độ dài chuỗi báo là nhỏ

hơn 0, có thể dùng 0 thay thế. Một số thiết bị (ví dụ đồng hồ tốc độ xe hơi) cũng áp dụng cách này, ví dụ khi xe lùi, đồng hồ tốc độ hiển thị 0 thay vì giá trị âm.

---

5. Ghi lại thông báo cảnh báo vào file log (Log a warning message to a file)

Khi phát hiện dữ liệu sai, có thể ghi lại một thông báo cảnh báo vào file log và tiếp tục chạy chương trình. Biện pháp này có thể kết hợp với kỹ thuật thay thế giá trị hợp lệ gần nhất hoặc thay thế dữ liệu tiếp theo. Khi sử dụng log, cần cân nhắc tính bảo mật, xác định xem log có thể công khai hay phải mã hóa, bảo vệ.

---

6. Trả về mã lỗi (Return an error code)

Bạn có thể thiết kế sao cho chỉ một số bộ phận hệ thống chịu trách nhiệm xử lý lỗi. Phần còn lại chỉ phát hiện lỗi và báo cáo lên các routine cấp cao hơn trong call hierarchy để xử lý. Cơ chế báo lỗi có thể là:

- Thiết lập giá trị biến trạng thái (status variable)
- Trả mã trạng thái như giá trị trả về của hàm
- Ném ra exception (ngoại lệ) nhờ cơ chế exception tích hợp của ngôn ngữ lập trình

Điều quan trọng là xác định rõ những phần nào trực tiếp xử lý lỗi, phần nào chỉ báo lỗi. Nếu liên quan đến bảo mật, các routine gọi luôn phải kiểm tra mã trả về.

---

7. Gọi routine/object xử lý lỗi (Call an error-processing routine/object)

Một cách khác là tập trung xử lý lỗi vào một routine hoặc object xử lý lỗi toàn cục (global error-handling routine/object). Ưu điểm là giúp tập trung trách nhiệm, dễ gỡ lỗi. Tuy nhiên, toàn bộ chương trình sẽ phải biết đến routine này và bị ràng buộc với nó. Nếu muốn tái sử dụng một số đoạn mã ở hệ thống khác, sẽ phải mang theo module xử lý lỗi này.

Lưu ý quan trọng về bảo mật: Nếu chương trình bị tràn bộ đệm (buffer overrun), kẻ tấn công có thể kiểm soát địa chỉ routine/object xử lý lỗi. Do đó, sau khi xảy ra buffer overrun, không còn an toàn khi tiếp tục sử dụng cơ chế này.

---

8. Hiển thị thông báo lỗi tại vị trí gặp lỗi (Display an error message wherever the error is encountered)

Biện pháp này giúp giảm chi phí xử lý lỗi. Tuy nhiên, nhược điểm là có thể khiến các thông báo liên quan đến giao diện người dùng (UI - User Interface) rải rác khắp ứng dụng. Điều này gây khó khăn trong việc tạo ra UI nhất quán, tách biệt rõ ràng UI với hệ thống xử lý nghiệp vụ, hoặc khi muốn bản địa hóa phần mềm sang ngôn ngữ khác. Ngoài ra, cần tránh tiết lộ quá nhiều thông tin cho kẻ tấn công qua thông báo lỗi, vì chúng có thể tận dụng để tìm lỗ hổng.

---

9. Xử lý lỗi phù hợp tại từng vị trí (Handle the error in whatever way works best locally)

Một số thiết kế cho phép từng đoạn mã tự quyết định cách xử lý từng lỗi cụ thể. Phương pháp này cung cấp sự linh hoạt cho từng lập trình viên, nhưng cũng tiềm ẩn rủi ro lớn là hiệu năng toàn hệ thống khó đáp ứng các tiêu chí về tính đúng đắn (correctness) hoặc độ bền vững (robustness). Các vấn đề liên quan đến UI cũng có thể phát sinh nếu hiện diện khắp hệ thống.

---

10. Tắt chương trình (Shut down)

Một số hệ thống sẽ dừng hoạt động ngay khi phát hiện lỗi, đặc biệt trong các ứng dụng an toàn, trọng yếu (safety-critical applications). Ví dụ, nếu phần mềm điều khiển thiết bị xạ trị phát hiện dữ liệu liều xạ sai, hành động đúng là phải dừng hoạt động. Dùng lại giá trị cũ, giá trị gần đúng, hay giá trị trung tính đều không phù hợp, vì rủi ro ảnh hưởng nghiêm trọng. Tốt nhất nên khởi động lại thiết bị.

Tương tự, Windows có thể được cấu hình để dừng máy chủ nếu security log (nhật ký bảo mật) bị đầy, điều này phù hợp với môi trường yêu cầu bảo mật cao.

Độ bền vững (Robustness) so với Tính đúng đắn (Correctness)

Như hai ví dụ về trò chơi và máy x-quang đã chỉ ra, phong cách xử lý lỗi phù hợp phụ thuộc vào loại phần mềm. Xử lý lỗi thường thiên về ưu tiên tính đúng đắn hoặc độ bền vững:

- **Tính đúng đắn (correctness):** Không bao giờ trả về kết quả sai. Không trả về gì còn tốt hơn trả về kết quả không chính xác.

- **Độ bền vững (robustness):** Luôn cố gắng duy trì hoạt động của phần mềm, ngay cả khi có thể dẫn đến kết quả đôi lúc không chính xác.

Ứng dụng an toàn, trọng yếu (safety-critical) thường ưu tiên tính đúng đắn. Ngược lại, các phần mềm tiêu dùng (consumer applications) lại ưu tiên độ bền vững — bất kỳ kết quả nào cũng tốt hơn là phần mềm bị dừng đột ngột. Ví dụ, trình xử lý văn bản đôi khi hiển thị một phần dòng văn bản ở cuối màn hình, nhưng vẫn tiếp tục hoạt động, đảm bảo người dùng không bị gián đoạn.

Tác động của Xử lý Lỗi tới Thiết kế ở Cấp Độ Cao

Tôi biết rằng lần tới khi tôi nhấn Page Up hoặc Page Down, màn hình sẽ làm mới và hiển thị sẽ trở lại trạng thái bình thường.

Tác động của Thiết kế Cấp Độ Cao đối với Xử lý Lỗi

Với rất nhiều lựa chọn về cách xử lý, bạn cần thận trọng trong việc xử lý các tham số không hợp lệ một cách nhất quán trong toàn bộ chương trình. Phương thức xử lý lỗi sẽ ảnh hưởng đến khả năng phần mềm đáp ứng các yêu cầu liên quan đến tính đúng đắn, độ vững chắc (robustness) và các thuộc tính phi chức năng (nonfunctional attributes) khác.

> **Lưu ý quan trọng:** Việc quyết định phương pháp tổng thể để xử lý tham số xấu là một quyết định kiến trúc hoặc thiết kế cấp cao, cần được giải quyết ở các cấp độ đó.

Khi đã quyết định phương pháp xử lý, bạn phải đảm bảo thực hiện nhất quán. Nếu bạn quyết định để mã ở cấp cao hơn xử lý lỗi, và mã ở cấp thấp hơn chỉ báo cáo lỗi, hãy chắc chắn rằng code cấp cao thực sự có xử lý các lỗi đó! Một số ngôn ngữ cho phép bạn bỏ qua việc một hàm trả về mã lỗi—trong C++ chẳng hạn, bạn không bắt buộc phải xử lý giá trị trả về của một hàm—tuy nhiên, đừng bỏ qua thông tin lỗi! Hãy kiểm tra giá trị trả về của hàm. Dù bạn không mong đợi hàm đó bao giờ trả về lỗi, hãy kiểm tra nó. Toàn bộ mục tiêu của lập trình phòng thủ (defensive programming) là bảo vệ khỏi những lỗi bạn không lường trước.

Hướng dẫn này áp dụng cả với hàm hệ thống cũng như hàm bạn tự viết. Trừ khi bạn đã đặt ra định hướng kiến trúc không kiểm tra lỗi từ lệnh gọi hệ thống, hãy kiểm tra mã lỗi sau mỗi lần gọi. Nếu phát hiện lỗi, hãy đưa vào số hiệu lỗi và mô tả lỗi tương ứng.

8.4 Ngoại lệ (Exceptions)

Ngoại lệ (exception) là một phương tiện cụ thể để mã có thể truyền đạt lỗi hoặc sự kiện ngoại lệ đến mã gọi nó. Nếu mã trong một routine gặp một điều kiện bất ngờ mà không biết cách xử lý, nó sẽ ném (throw) một ngoại lệ, về bản chất là "giơ tay đầu hàng" và nói: "Tôi không biết xử lý thế nào với điều này—hy vọng ai đó ở tầng trên sẽ biết cách!" Khi mã không thể nhận diện ngữ cảnh lỗi, nó có thể trả quyền kiểm soát về các phần khác của hệ thống nơi có khả năng xử lý hoặc phản ứng phù hợp hơn.

Ngoại lệ cũng có thể dùng để làm mạch lạc các logic rối rắm bên trong một đoạn mã đơn lẻ, như ví dụ "Viết lại với try-finally" ở Mục 17.3. Cấu trúc cơ bản của ngoại lệ là một routine sử dụng `throw` để ném một đối tượng ngoại lệ, và một routine khác trong chuỗi gọi sẽ bắt (catch) nó trong khối try-catch.

Các ngôn ngữ phổ biến khác nhau về mặt triển khai ngoại lệ. Bảng sau tóm tắt các khác biệt chủ yếu giữa ba ngôn ngữ:

Thuộc tính ngoại lệ (Exception Attribute)	C++	Java	Visual Basic
Hỗ trợ try-catch	Có	Có	Có
Hỗ trợ try-catch-finally	Không	Có	Có
Có thể ném gì (what can be thrown)	Đối tượng ngoại lệ (exception object) hoặc đối tượng dẫn xuất từ lớp Exception; con trỏ đối tượng; tham chiếu đối tượng; kiểu dữ liệu như string hoặc int	Đối tượng ngoại lệ hoặc đối tượng dẫn xuất từ lớp Exception	Đối tượng ngoại lệ hoặc đối tượng dẫn xuất từ lớp Exception
Tác động khi ngoại lệ không được bắt	Gọi `std::unexpected()`, mặc định gọi `std::terminate()`, tiếp tục gọi `abort()`	Dừng luồng thực thi nếu ngoại lệ là "checked exception"; không ảnh hưởng nếu là "runtime exception"	Dừng chương trình

Thuộc tính ngoại lệ (Exception Attribute)	C++	Java	Visual Basic
Ngoại lệ ném ra phải được định nghĩa trong interface	Không	Có	Không
Ngoại lệ bắt phải được định nghĩa trong interface	Không	Có	Không

> Các chương trình sử dụng ngoại lệ như một phần của xử lý bình thường sẽ gặp phải tất cả các vấn đề về khả năng đọc và bảo trì tương tự như code spaghetti truyền thống.
>
> — *Andy Hunt and Dave Thomas*

Những lưu ý khi sử dụng ngoại lệ

- **Dùng ngoại lệ để thông báo cho phần khác của chương trình về lỗi không nên bị bỏ qua:** Lợi ích lớn nhất của ngoại lệ là khả năng thông báo điều kiện lỗi mà không thể bị làm ngơ (Meyers 1996). Các phương pháp xử lý lỗi khác tạo ra khả năng một lỗi bị truyền đi mà không ai phát hiện. Ngoại lệ loại trừ nguy cơ này.
- **Chỉ ném ngoại lệ cho các điều kiện thực sự ngoại lệ:** Ngoại lệ nên dùng cho các tình huống thực sự hiếm hoi—những điều kiện không thể xử lý bằng các phương pháp lập trình thường thấy khác. Mục đích của ngoại lệ tương tự như assertion (khẳng định) — cho những sự kiện không chỉ là hy hữu mà còn "không bao giờ nên xảy ra".
- **Ngoại lệ là sự đánh đổi giữa khả năng xử lý điều kiện bất ngờ và sự phức tạp tăng lên:** Khi dùng ngoại lệ, tính đóng gói (encapsulation) bị suy yếu vì mã gọi phải biết có thể có ngoại lệ nào được ném ra bên trong. Điều này tăng độ phức tạp của mã, mâu thuẫn với Mệnh lệnh Kỹ thuật Chính của Phần mềm: Quản lý Độ phức tạp.

Đừng dùng ngoại lệ chỉ để "đẩy trách nhiệm"

Nếu một điều kiện lỗi có thể được xử lý tại chỗ, hãy xử lý nó ở đó. Đừng ném một ngoại lệ chưa được xử lý lên trên nếu bạn có thể tự xử lý tại chỗ.

Tránh ném ngoại lệ trong constructor và destructor trừ khi có bắt tại chỗ

Quy tắc xử lý ngoại lệ trở nên phức tạp rất nhanh nếu ném ngoại lệ trong constructor hoặc destructor. Ví dụ, trong C++, destructor sẽ không được gọi nếu một đối tượng chưa được xây dựng đầy đủ, nghĩa là nếu code trong constructor ném ngoại lệ, destructor sẽ không được gọi, dễ dẫn đến rò rỉ tài nguyên. Quy tắc tương tự áp dụng cho exceptions trong destructor.

Các chuyên gia ngôn ngữ có thể nói nhớ các quy tắc này là việc "tầm thường", nhưng lập trình viên thường sẽ khó nhớ hết. Hãy tránh viết dạng code này để giảm bớt sự phức tạp không cần thiết.

> **Tham khảo chéo:** Để biết thêm về duy trì tính nhất quán của tầng trừu tượng giao diện, xem "Good Abstraction" ở mục 6.2.

Ném ngoại lệ ở mức trừu tượng phù hợp

Hàm nên thể hiện một tầng trừu tượng đồng nhất trong giao diện, và lớp cũng vậy. Các ngoại lệ ném ra là một phần của giao diện routine, cũng như các kiểu dữ liệu cụ thể.

Khi chọn truyền một ngoại lệ cho caller, cần đảm bảo mức trừu tượng của ngoại lệ phù hợp với giao diện routine. Dưới đây là ví dụ về việc làm sai:

Ví dụ Java xấu: Ném ngoại lệ ở mức trừu tượng không nhất quán

```
class Employee {
    public TaxId GetTaxId() throws EOFException {
    }
}
```

Ở đây, `GetTaxId()` truyền một ngoại lệ cấp thấp `EOFException` lên caller. Nó không tự chịu trách nhiệm với exception này mà còn phơi bày chi tiết cách hiện thực bên trong Employee, khiến code phía client phải phụ thuộc cả vào code lớp dưới của Employee (nơi phát sinh EOFException). Điều này phá vỡ đóng gói và làm quản lý trở nên khó khăn.

Thay vào đó, nên ném ngoại lệ có cùng tầng trừu tượng với giao diện lớp:

Ví dụ Java đúng: Ném ngoại lệ ở mức trừu tượng phù hợp

```
class Employee {
    public TaxId GetTaxId() throws EmployeeDataNotAvailable {
    }
}
```

Code xử lý ngoại lệ trong `GetTaxId()` có thể ánh xạ lỗi cấp thấp (ví dụ: `io_disk_not_ready`) thành ngoại lệ `EmployeeDataNotAvailable`, đảm bảo duy trì tầng trừu tượng của interface.

Bao gồm tất cả thông tin liên quan trong thông điệp ngoại lệ

Mỗi ngoại lệ xảy ra trong hoàn cảnh cụ thể, và thông tin này rất quan trọng cho người đọc thông điệp ngoại lệ. Hãy đảm bảo thông điệp có chứa các thông tin để hiểu rõ lý do ngoại lệ phát sinh. Nếu ngoại lệ do lỗi chỉ số mảng, thông điệp phải ghi rõ giới hạn trên, dưới và giá trị chỉ số không hợp lệ.

Tránh khối catch rỗng (empty catch blocks)

Đôi khi bạn có thể cảm thấy nên "lơ đi" một ngoại lệ như sau:

Ví dụ Java xấu: Bỏ qua ngoại lệ

```
try {
    // lots of code
} catch (AnException exception) {
}
```

Cách này chứng tỏ hoặc code trong try là sai (vì sinh ra exception mà không có lý do chính đáng), hoặc code trong catch là sai (vì không xử lý ngoại lệ hợp lệ). Xác định nguyên nhân thực sự, rồi sửa code ở khối try hoặc catch cho hợp lý!

Nếu thực sự có những trường hợp hiếm mà ngoại lệ ở tầng dưới không còn là ngoại lệ ở tầng trừu tượng caller, ít nhất nên ghi chú rõ lý do khối catch rỗng. Có thể dùng chú thích hoặc ghi log vào file, ví dụ:

Ví dụ Java đúng: Ghi nhận ngoại lệ được bỏ qua

```
try {
    // lots of code
} catch (AnException exception) {
    LogError("Unexpected exception");
}
```

Nắm rõ các ngoại lệ mà thư viện trả về

Nếu bạn làm việc với một ngôn ngữ không yêu cầu routine hoặc lớp phải định nghĩa rõ ngoại lệ nó có thể ném, hãy chắc chắn rằng bạn biết các ngoại lệ mà mọi mã thư viện bạn dùng có thể sinh ra.

Nếu mã thư viện không ghi chú các ngoại lệ mà nó phát sinh

Nếu mã thư viện không ghi chú rõ các ngoại lệ (exception) mà nó phát sinh, hãy xây dựng một đoạn mã thử nghiệm (prototyping code) để kiểm tra các thư viện này và phát hiện các ngoại lệ tiềm ẩn.

Xây dựng bộ báo cáo ngoại lệ tập trung (Centralized Exception Reporter)

Một cách tiếp cận nhằm đảm bảo tính nhất quán trong xử lý ngoại lệ là sử dụng bộ báo cáo ngoại lệ tập trung (centralized exception reporter). Bộ báo cáo này cung cấp một kho lưu trữ tập trung về các loại ngoại lệ, cách thức xử lý từng ngoại lệ, định dạng thông báo ngoại lệ và các thông tin liên quan.

Dưới đây là ví dụ về một **trình xử lý ngoại lệ đơn giản** chỉ in ra thông báo chẩn đoán:

Ví dụ Visual Basic về bộ báo cáo ngoại lệ tập trung - Phần 1

```
Sub ReportException( _
   ByVal className, _
   ByVal thisException As Exception _
)
   Dim message As String
   Dim caption As String
   message = "Exception: " & thisException.Message & " " & ControlChars.CrLf & _
             "Class: " & className & ControlChars.CrLf & _
             "Routine: " & thisException.TargetSite.Name & ControlChars.CrLf
   caption = "Exception"
```

```
   MessageBox.Show(message, caption, MessageBoxButtons.OK, _
                   MessageBoxIcon.Exclamation)
End Sub
```

Bạn sẽ sử dụng **trình xử lý ngoại lệ tổng quát này** như sau:

Ví dụ Visual Basic về bộ báo cáo ngoại lệ tập trung - Phần 2

```
Try

Catch exceptionObject As Exception
   ReportException(CLASS_NAME, exceptionObject)
End Try
```

Đoạn mã trong phiên bản này của `ReportException()` rất đơn giản. Trong một ứng dụng thực tế, bạn có thể làm đoạn mã này đơn giản hoặc phức tạp tùy thuộc nhu cầu xử lý ngoại lệ của mình.

Khi quyết định xây dựng một bộ báo cáo ngoại lệ tập trung, cần cân nhắc các vấn đề tổng quan liên quan đến xử lý lỗi tập trung, như đã được đề cập trong mục **"Gọi một routine/đối tượng xử lý lỗi" ở Mục 8.3**.

Chuẩn hóa việc sử dụng ngoại lệ trong dự án

Để việc xử lý ngoại lệ trở nên dễ quản lý hơn về mặt tư duy, bạn có thể chuẩn hóa quy ước sử dụng ngoại lệ theo các cách sau:

- Nếu bạn đang làm việc với ngôn ngữ như C++ cho phép phát sinh nhiều loại đối tượng, dữ liệu và con trỏ khác nhau, hãy chuẩn hóa loại đối tượng cụ thể mà bạn sẽ phát sinh (throw). Để đảm bảo khả năng tương thích với các ngôn ngữ khác, nên cân nhắc chỉ phát sinh các đối tượng kế thừa từ lớp cơ sở `Exception`.
- Xem xét tạo một lớp ngoại lệ riêng cho từng dự án, lớp này có thể làm lớp cơ sở cho tất cả các ngoại lệ phát sinh trong dự án. Việc này giúp tập trung và chuẩn hóa việc ghi nhật ký, báo cáo lỗi, v.v.
- Xác định rõ các trường hợp mã nguồn được phép sử dụng cú pháp throw-catch để xử lý lỗi cục bộ.
- Xác định rõ các trường hợp mã nguồn được phép phát sinh ngoại lệ mà không xử lý cục bộ.
- Quyết định xem có sử dụng bộ báo cáo ngoại lệ tập trung hay không.
- Định nghĩa rõ việc ngoại lệ có được phép phát sinh trong constructor và destructor hay không.

Tham khảo thêm

> Với nhiều phương pháp xử lý lỗi thay thế, xem mục "Error-Handling Techniques" (Kỹ thuật xử lý lỗi) trước đó trong chương này.

Cân nhắc các phương án thay thế ngoại lệ

Nhiều ngôn ngữ lập trình đã hỗ trợ ngoại lệ từ 5-10 năm hoặc hơn, tuy nhiên vẫn chưa hình thành nhiều quan điểm chung về cách sử dụng chúng một cách an toàn.

Một số lập trình viên sử dụng ngoại lệ chỉ vì ngôn ngữ họ dùng hỗ trợ cơ chế xử lý lỗi này. Tuy nhiên, bạn nên cân nhắc đầy đủ các phương án xử lý lỗi, bao gồm:

- Xử lý lỗi ngay tại nơi phát sinh (local handling)
- Truyền mã lỗi (error code)
- Ghi nhật ký thông tin gỡ lỗi vào tập tin
- Tắt hệ thống
- Hoặc sử dụng các cách tiếp cận khác

Việc sử dụng ngoại lệ chỉ vì ngôn ngữ hỗ trợ cơ chế này là một ví dụ điển hình của **lập trình theo ngôn ngữ (programming in a language)** thay vì **lập trình khai thác ngôn ngữ (programming into a language)** (xem thêm chi tiết ở mục “Your Location on the Technology Wave” và mục "Program into Your Language, Not in It").

Cuối cùng, cần cân nhắc liệu chương trình của bạn **thực sự cần xử lý ngoại lệ hay không**. Như Bjarne Stroustrup đã chỉ ra, đôi khi phản ứng tối ưu với một lỗi thời gian chạy nghiêm trọng là giải phóng toàn bộ tài nguyên đã cấp phát rồi kết thúc chương trình. Hãy để người dùng chạy lại chương trình với đầu vào hợp lệ (Stroustrup 1997).

8.5 Dựng rào chắn trong chương trình để hạn chế thiệt hại do lỗi gây ra

Barricade là chiến lược ngăn chặn thiệt hại (damage-containment strategy). Điều này tương tự lý do các khoang tàu thường được cách biệt; nếu xảy ra sự cố thủng vỏ tàu, khoang bị ảnh hưởng sẽ được cách ly để phần còn lại không bị tác động. Tương tự, tường lửa trong tòa nhà được sử dụng để ngăn chặn cháy lan.

> (Trước đây, "firewall" được dùng đồng nghĩa với "barricade" nhưng hiện tại chủ yếu chỉ dùng để chỉ các biện pháp chặn lưu lượng mạng độc hại.)

Một cách để tạo **rào chắn** trong lập trình phòng thủ là xác định một số giao diện (interface) như ranh giới giữa "vùng an toàn". Hãy kiểm tra tính hợp lệ của dữ liệu vượt qua ranh giới này và xử lý hợp lý nếu dữ liệu không hợp lệ.

Hình minh họa bên dưới cho thấy:

> Một số phần của phần mềm sẽ làm việc với dữ liệu "bẩn" (dirty data) và không tin cậy, chúng chịu trách nhiệm làm sạch dữ liệu và tạo nên rào chắn. Các phần bên trong rào chắn có thể giả định dữ liệu đã sạch (clean) và đáng tin cậy (trusted).

Phương pháp này có thể áp dụng ở cấp lớp (class level): các phương thức public của lớp sẽ kiểm tra và làm sạch dữ liệu đầu vào, còn các phương thức private có thể giả định dữ liệu đã hợp lệ.

Một cách ví von khác là coi đây như kỹ thuật phòng phẫu thuật: Dữ liệu phải được "tiệt trùng" trước khi đưa vào phòng mổ. Quyết định thiết kế then chốt là xác định đối tượng nào có thể "vào phòng mổ", cái nào thì không, và đặt các "cửa" (routine) ở đâu — routine nào nằm trong vùng an toàn, routine nào ngoài, routine nào chịu trách nhiệm làm sạch dữ liệu.

Thông thường, thuận tiện nhất là làm sạch dữ liệu từ bên ngoài ngay khi nhận được. Tuy nhiên, đôi khi cần làm sạch ở nhiều cấp độ khác nhau.

Chuyển đổi dữ liệu đầu vào về đúng kiểu ngay tại thời điểm nhập liệu

Thông thường, dữ liệu đầu vào dưới dạng chuỗi (string) hoặc số (number), đôi khi ánh xạ sang kiểu boolean như “yes” hoặc “no”, hoặc sang dạng liệt kê (enumerated type) như `Color_Red`, `Color_Green`, `Color_Blue`. Nếu duy trì dữ liệu chưa xác định kiểu trong thời gian dài sẽ làm tăng độ phức tạp và rủi ro (ví dụ nhập giá trị “Yes” làm cho chương trình lỗi). Vì vậy, hãy chuyển đổi dữ liệu về đúng kiểu càng sớm càng tốt.

8.6 Công cụ hỗ trợ gỡ lỗi (Debugging Aids)

Một khía cạnh then chốt khác của lập trình phòng thủ là sử dụng các công cụ hỗ trợ gỡ lỗi (debugging aids), giúp phát hiện lỗi nhanh chóng.

Không áp dụng mặc định các ràng buộc của phiên bản sản xuất vào phiên bản phát triển

Nhiều lập trình viên thường bị mắc kẹt trong quan niệm rằng các giới hạn của phần mềm sản xuất (production software) cũng áp dụng cho phiên bản phát triển. Thật ra, phiên bản sản xuất cần chạy nhanh và tiết kiệm tài nguyên, trong khi phiên bản phát triển có thể hi sinh tốc độ, tiêu thụ nhiều tài nguyên hơn để phục vụ cho các công cụ hỗ trợ phát triển.

Ví dụ: Trong một dự án, chúng tôi sử dụng danh sách liên kết bốn chiều (quadruply linked list) dễ xảy ra lỗi. Tôi đã bổ sung chức năng kiểm tra tính toàn vẹn của danh sách vào menu.

Trong chế độ gỡ lỗi (debug mode), Microsoft Word có đoạn mã kiểm tra trạng thái của đối tượng Document liên tục vài giây một lần, giúp phát hiện sớm lỗi trên dữ liệu.

> Hãy sẵn sàng đánh đổi tốc độ và tài nguyên trong giai đoạn phát triển để đổi lấy các công cụ hỗ trợ giúp quá trình phát triển dễ dàng và trơn tru hơn.

Đưa vào các công cụ hỗ trợ gỡ lỗi từ sớm

Càng sớm bổ sung các công cụ hỗ trợ gỡ lỗi, bạn càng hưởng lợi nhiều hơn. Thông thường, bạn sẽ không viết các công cụ hỗ trợ này cho đến khi gặp phải một vấn đề nào đó nhiều lần.

Ghi chú dịch thuật:

- Thuật ngữ như: exception, class, routine, assertion, debug mode, v.v. được giữ nguyên và giải thích lần đầu tiên.
- Một số lỗi đánh máy/gõ thừa của bản gốc đã được sửa hoặc bỏ qua để tăng sự mạch lạc (như các dấu câu thiếu, cách dòng lặp lại).
- Bài dịch được trình bày ngắn gọn, đúng chuẩn truyền đạt học thuật, nhất quán về thuật ngữ và cấu trúc.

Sử Dụng Offensive Programming (Lập trình Tấn công)

> **Tham khảo chéo:** Để biết thêm chi tiết về xử lý các trường hợp không lường trước, xem mục "Tips for Using case Statements" trong Phần 15.2.

Những tình huống ngoại lệ (exceptional cases) cần được xử lý sao cho chúng trở nên rõ ràng trong quá trình phát triển và có thể khôi phục được khi code vận hành thực tế. Michael Howard và David LeBlanc gọi phương pháp này là “offensive programming” (lập trình tấn công) (Howard và LeBlanc, 2003).

Giả sử bạn có một câu lệnh `case` dự kiến sẽ xử lý chỉ năm loại sự kiện. Trong quá trình phát triển, nhánh `default` nên được sử dụng để tạo ra cảnh báo, ví dụ: “Có một trường hợp mới! Hãy sửa chương trình!”. Tuy nhiên, trong môi trường vận hành (production), nhánh `default` nên thực hiện một hành động nhẹ nhàng hơn, như ghi thông báo vào file nhật ký lỗi (error-log file).

> Một chương trình bị dừng hoàn toàn (dead program) thường gây ra ít thiệt hại hơn một chương trình hoạt động chập chờn (crippled program).
>
> — Andy Hunt và Dave Thomas

Dưới đây là một số cách bạn có thể lập trình theo hướng tấn công (offensive):

- **Đảm bảo các lệnh xác nhận (assert) sẽ dừng chương trình.** Đừng để lập trình viên chỉ việc nhấn phím Enter để bỏ qua một vấn đề đã biết. Hãy làm cho vấn đề này đủ khó chịu để bắt buộc phải sửa.
- **Ghi đầy bộ nhớ đã được cấp phát** để có thể phát hiện lỗi cấp phát bộ nhớ (memory allocation errors).
- **Ghi đầy các file hoặc luồng dữ liệu được cấp phát** để phát hiện lỗi định dạng file (file-format errors).
- **Đảm bảo mã lệnh ở nhánh `default` hoặc `else` trong mỗi câu lệnh `case` phải “fail hard” (dừng chương trình), hoặc gây chú ý lớn, không thể bị bỏ sót.**
- **Làm đầy đối tượng với dữ liệu "rác" trước khi xóa** để dễ phát hiện sử dụng nhầm dữ liệu đã xóa.
- **Cấu hình chương trình gửi file log lỗi đến email của bạn**, nếu phù hợp với loại phần mềm bạn phát triển, để bạn biết các loại lỗi đang diễn ra khi phần mềm đã phát hành.

Đôi khi, phòng ngự tốt nhất là tấn công mạnh mẽ. Hãy làm chương trình "fail hard" (lỗi lớn) trong giai đoạn phát triển, nhằm giúp chương trình "fail soft" (lỗi nhẹ nhàng) khi đã đưa vào sản xuất.

Lập Kế Hoạch Loại Bỏ Công Cụ Gỡ Lỗi (Debugging Aids)

Nếu bạn viết code cho mục đích cá nhân, việc giữ lại toàn bộ code gỡ lỗi trong chương trình có thể không sao. Tuy nhiên, khi phát triển phần mềm thương mại, cái giá phải trả về dung lượng và tốc độ sẽ rất lớn. Vì thế, hãy lên kế hoạch tránh việc liên tục thêm/bớt code gỡ lỗi khỏi chương trình. Sau đây là vài cách thực hiện:

> **Tham khảo chéo:** Để biết chi tiết về quản lý phiên bản, xem Phần 28.2 "Configuration Management".

- **Sử dụng công cụ quản lý phiên bản (version-control tools)** và công cụ build như `ant` hoặc `make`. Các công cụ quản lý phiên bản cho phép xây dựng nhiều phiên bản chương trình từ cùng một bộ mã nguồn gốc. Ở chế độ phát triển, bạn có thể cấu hình để build bao gồm toàn bộ code gỡ lỗi. Ở chế độ vận hành, code debug sẽ bị loại bỏ khỏi phiên bản thương mại.

- **Sử dụng preprocessor (tiền xử lý) có sẵn.** Nếu môi trường lập trình có hỗ trợ preprocessor, như C++, bạn có thể bao gồm hoặc loại trừ code gỡ lỗi chỉ bằng một chuyển đổi trong quá trình biên dịch. Bạn có thể dùng trực tiếp preprocessor hoặc định nghĩa macro tương ứng. Ví dụ:

```
#define DEBUG

#if defined( DEBUG )
// debugging code
#endif
```

Với chủ đề này, có thể có nhiều biến thể khác nhau. Thay vì chỉ định nghĩa `DEBUG`, bạn có thể gán cho nó một giá trị và kiểm tra giá trị đó để phân biệt các mức độ debug khác nhau. Ví dụ, một số đoạn code gỡ lỗi cần tồn tại mọi lúc thì có thể đặt điều kiện như `#if DEBUG > 0`, hoặc một số đoạn code chỉ cho mục đích nhất định thì dùng như `#if DEBUG == POINTER_ERROR`.

Nếu bạn không thích việc lặp đi lặp lại nhiều dòng `#if defined()` trong code, hãy tạo macro tiền xử lý như sau:

```
#define DEBUG
#if defined( DEBUG )
#define DebugCode( code_fragment ) { code_fragment }
#else
#define DebugCode( code_fragment )
#endif

DebugCode(
    statement 1;
    statement 2;
    statement n;
);
```

Kỹ thuật này cũng có thể được mở rộng để tinh chỉnh hơn, không chỉ hoàn toàn loại bỏ hoặc giữ lại toàn bộ code debug.

Tham khảo chéo: Xem thêm về tiền xử lý (preprocessor) tại "Macro Preprocessors" trong Phần 30.3.

- **Viết tiền xử lý riêng** nếu ngôn ngữ không hỗ trợ có sẵn. Bạn có thể đặt quy ước đánh dấu các đoạn code debug (ví dụ, trong Java dùng `//#BEGIN DEBUG` và `//#END DEBUG`) và viết script xử lý trước khi biên dịch. Như vậy, bạn sẽ tiết kiệm thời gian về lâu dài và tránh việc vô tình biên dịch code chưa được xử lý qua preprocessor.

- **Sử dụng debugging stubs (chương trình con giả để kiểm tra).** Trong nhiều trường hợp, bạn có thể gọi một routine để kiểm tra khi debug. Trong bản phát triển, routine này sẽ thực hiện nhiều phép kiểm tra, còn trong bản sản xuất, bạn chỉ cần routine trả về ngay lập tức hoặc thực hiện tối thiểu thao tác, giúp giảm thiểu tác động đến hiệu năng. Bạn nên giữ cả hai phiên bản phát triển và sản xuất để tiện hoán đổi khi cần.

Ví dụ:

```
void DoSomething(
    SOME_TYPE *pointer;
) {
    CheckPointer( pointer );
}
```

Trong phát triển, routine `CheckPointer()` sẽ kiểm tra kỹ đầu vào:

```
void CheckPointer( void *pointer ) {
    // kiểm tra pointer không phải NULL
    // kiểm tra dogtag hợp lệ
    // kiểm tra vùng trỏ tới không bị hỏng
    // ...
}
```

Ở môi trường sản xuất, bạn có thể dùng routine rỗng:

```
void CheckPointer( void *pointer ) {
    // Không làm gì, trả về ngay
}
```

Đây không phải là danh sách đầy đủ các cách loại bỏ công cụ gỡ lỗi, nhưng đủ để bạn lựa chọn giải pháp phù hợp cho môi trường phát triển của mình.

8.7 Xác Định Mức Độ Defensive Programming Cần Giữ Lại Trong Code Sản Xuất

Một nghịch lý trong defensive programming (lập trình phòng ngừa) là trong quá trình phát triển, bạn muốn lỗi xuất hiện thật rõ rệt, thậm chí "đáng ghét" để đảm bảo không bỏ sót. Nhưng khi tiến hành vận hành, bạn lại mong lỗi càng nhẹ nhàng, không ảnh hưởng lớn, chương trình có khả năng phục hồi hoặc dừng một cách "êm ái".

Sau đây là một số hướng dẫn về việc quyết định giữ/loại bỏ defensive programming trong phiên bản sản xuất:

- **Giữ lại code kiểm tra các lỗi quan trọng.** Xác định khu vực nào của chương trình cho phép bỏ sót lỗi (undetected errors), khu vực nào không. Ví dụ, trong chương trình bảng tính, bạn có thể bỏ qua lỗi không phát hiện trong phần cập nhật màn hình vì hậu quả chỉ là giao diện bị rối. Tuy nhiên, không thể bỏ qua lỗi trong engine tính toán, vì có thể dẫn đến kết quả sai ngầm, gây ra hậu quả nghiêm trọng như khai thuế bị kiểm toán.

- **Loại bỏ code kiểm tra các lỗi không quan trọng.** Nếu một lỗi có hậu quả thật sự tầm thường, hãy loại bỏ code kiểm tra lỗi đó — nghĩa là sử dụng công cụ quản lý phiên bản, preprocessor, hoặc kỹ thuật khác để không biên dịch đoạn code này vào chương trình thương mại. Nếu dung lượng không phải vấn đề, bạn có thể giữ code kiểm tra này nhưng để nó chỉ ghi lỗi một cách âm thầm vào file nhật ký.

- **Loại bỏ code gây crash mạnh (hard crashes).** Trong giai đoạn phát triển, khi chương trình phát hiện lỗi, bạn muốn lỗi xuất hiện thật rõ ràng, kể cả với các lỗi nhỏ, thường là bằng cách in ra thông báo debug rồi crash chương trình. Tuy nhiên, ở bản vận hành, người dùng cần có thời gian lưu dữ liệu trước khi chương trình gặp sự cố, và sẵn sàng chấp nhận một số sai sót nhỏ để tiếp tục sử dụng tiếp. Người dùng không thích mất dữ liệu, bất kể điều đó giúp cho việc debug về sau. Vì vậy, nếu chương trình chứa đoạn mã debug có thể gây mất dữ liệu, hãy loại bỏ chúng ở bản sản xuất.

Ví dụ từ Mars Pathfinder về việc sử dụng mã gỡ lỗi trong sản phẩm

Trong dự án Mars Pathfinder, các kỹ sư đã chủ động giữ lại một số đoạn mã gỡ lỗi (debug code) trong phần mềm. Một lỗi đã phát sinh sau khi Pathfinder hạ cánh xuống sao Hỏa. Nhờ vào các công cụ hỗ trợ gỡ lỗi còn được giữ lại, các kỹ sư tại JPL đã

có thể chẩn đoán vấn đề, sau đó tải lên phiên bản mã đã chỉnh sửa cho Pathfinder, giúp thiết bị hoàn thành nhiệm vụ một cách hoàn hảo (tháng 3 năm 1999).

Ghi nhận lỗi cho đội ngũ hỗ trợ kỹ thuật

- Hãy ghi lại (log) các lỗi cho đội ngũ hỗ trợ kỹ thuật. Xem xét việc giữ lại các công cụ hỗ trợ gỡ lỗi trong mã sản phẩm, nhưng điều chỉnh hành vi của chúng sao cho phù hợp với phiên bản chính thức.
- Nếu bạn đã thêm nhiều biểu thức xác nhận (assertion) khiến chương trình dừng lại trong quá trình phát triển, bạn nên cân nhắc thay đổi routine assertion sang ghi lại thông báo vào tệp (log) khi chạy ở môi trường sản xuất, thay vì loại bỏ hoàn toàn chúng.

Đảm bảo thông báo lỗi thân thiện

- Đảm bảo rằng các thông báo lỗi để lại trong chương trình là thân thiện với người dùng. Nếu giữ lại các thông báo lỗi nội bộ, hãy chắc chắn chúng sử dụng ngôn ngữ phù hợp với người dùng cuối.
- Trong một chương trình trước đây của tôi, một người dùng đã gọi và báo rằng cô ấy nhận được thông báo: “You’ve got a bad pointer allocation, Dog Breath!” (Bạn có một phân bổ con trỏ lỗi, Hơi Thở Chó!). May mắn là cô ấy có khiếu hài hước.
- Một cách tiếp cận phổ biến và hiệu quả là thông báo với người dùng về một “lỗi nội bộ” và cung cấp địa chỉ email hoặc số điện thoại để người dùng báo cáo lỗi đó.

8.8 Cảnh giác với lập trình phòng thủ quá mức

> "Quá nhiều bất cứ điều gì đều không tốt, nhưng quá nhiều whisky thì vừa đủ."
> — Mark Twain

Lập trình phòng thủ (defensive programming) thái quá cũng gây ra những vấn đề riêng. Nếu bạn kiểm tra dữ liệu được truyền dưới mọi hình thức, tại mọi nơi có thể, chương trình của bạn sẽ trở nên “phình to” và chậm chạp. Nghiêm trọng hơn, mã nguồn bổ sung cho mục đích phòng thủ còn làm tăng độ phức tạp phần mềm; bản thân mã lập trình phòng thủ cũng không miễn nhiễm với lỗi, thậm chí khả năng có lỗi còn cao hơn nếu bạn viết chúng một cách sơ sài. Do đó, hãy cân nhắc vị trí cần có sự phòng thủ và ưu tiên những nơi thực sự cần thiết.

Danh sách kiểm tra: Lập trình phòng thủ (Defensive Programming)

Tổng quan

- ❑ Routine có tự bảo vệ khỏi dữ liệu đầu vào xấu không?
- ❑ Bạn đã sử dụng assertion để ghi rõ các giả định, gồm tiền điều kiện (precondition) và hậu điều kiện (postcondition) chưa?
- ❑ Assertion chỉ được dùng để ghi nhận các trường hợp không bao giờ được phép xảy ra?
- ❑ Kiến trúc hoặc thiết kế cấp cao liệu có xác định tập hợp các kỹ thuật xử lý lỗi cụ thể chưa?
- ❑ Kiến trúc hoặc thiết kế cấp cao có quy định việc nên ưu tiên tính bền vững (robustness) hay tính chính xác (correctness) khi xử lý lỗi?
- ❑ Có tạo các hàng rào (barricades) để giới hạn phạm vi ảnh hưởng của lỗi và giảm số lượng mã cần quan tâm đến xử lý lỗi hay không?
- ❑ Công cụ hỗ trợ gỡ lỗi đã được sử dụng trong mã nguồn chưa?
- ❑ Công cụ hỗ trợ gỡ lỗi đã được thiết lập để có thể bật/tắt một cách linh hoạt chưa?
- ❑ Lượng mã để lập trình phòng thủ đã hợp lý — không quá nhiều cũng không quá ít?
- ❑ Bạn đã sử dụng các kỹ thuật lập trình chủ động (offensive-programming) để khiến lỗi khó bị bỏ qua khi phát triển chưa?

Xử lý ngoại lệ (Exception)

- ❑ Dự án đã định nghĩa phương pháp chuẩn cho việc xử lý ngoại lệ chưa?
- ❑ Đã xem xét các giải pháp thay thế việc sử dụng exception chưa?
- ❑ Lỗi được xử lý cục bộ ở mức routine thay vì ném (throw) exception ra ngoài không cần thiết?
- ❑ Mã tránh ném ngoại lệ trong constructor và destructor chưa?
- ❑ Mọi exception đều ở đúng mức trừu tượng đối với routine ném chúng?
- ❑ Mỗi exception đều kèm theo đầy đủ thông tin nền liên quan?
- ❑ Đã loại bỏ các khối bắt rỗng (empty catch block)? Nếu bắt rỗng là cần thiết, đã được ghi chú rõ ràng?

Vấn đề bảo mật

- ❑ Mã kiểm tra dữ liệu đầu vào có kiểm soát tràn bộ đệm (buffer overflow), SQL injection, HTML injection, tràn số nguyên (integer overflow) và các dạng đầu vào độc hại khác không?
- ❑ Mọi mã trả về báo lỗi (error-return code) đều được kiểm tra?
- ❑ Mọi exception đều được bắt (caught)?
- ❑ Thông báo lỗi không tiết lộ thông tin có thể giúp kẻ tấn công xâm nhập hệ thống?

Tài liệu tham khảo về Lập trình phòng thủ (Defensive Programming)

Bảo mật

- Howard, Michael và David LeBlanc. *Writing Secure Code*, lần thứ 2. Microsoft Press, 2003. Tác giả phân tích các hệ lụy bảo mật của việc tin tưởng đầu vào, minh họa nhiều cách một chương trình có thể bị tấn công. Nội dung bao gồm yêu cầu về bảo mật, thiết kế, lập trình và kiểm thử.

Assertion

- Maguire, Steve. *Writing Solid Code*. Microsoft Press, 1993. Chương 2 bàn rất sâu về cách dùng assertion, cùng nhiều ví dụ thực tế từ sản phẩm của Microsoft.
- Stroustrup, Bjarne. *The C++ Programming Language*, lần thứ 3. Addison-Wesley, 1997. Mục 24.3.7.2 trình bày một số biến thể của triển khai assertion trong C++ cũng như mối quan hệ giữa assertion với precondition và postcondition.
- Meyer, Bertrand. *Object-Oriented Software Construction*, lần thứ 2. Prentice Hall PTR, 1997. Sách này đề cập sâu sắc đến tiền/hậu điều kiện.

Ngoại lệ (Exception)

- Meyer, Bertrand. *Object-Oriented Software Construction*, lần thứ 2. Prentice Hall PTR, 1997. Chương 12 trình bày chi tiết về xử lý ngoại lệ.
- Stroustrup, Bjarne. *The C++ Programming Language*, lần thứ 3. Addison-Wesley, 1997. Chương 14 thảo luận sâu về xử lý ngoại lệ trong C++. Mục 14.11 tóm lược 21 lưu ý khi xử lý ngoại lệ trong C++.
- Meyers, Scott. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley, 1996. Các mục 9–15 đề cập nhiều khía cạnh tinh tế của xử lý ngoại lệ trong C++.
- Arnold, Ken, James Gosling, và David Holmes. *The Java Programming Language*, lần thứ 3. Addison-Wesley, 2000. Chương 8 bàn về xử lý ngoại lệ trong Java.
- Bloch, Joshua. *Effective Java Programming Language Guide*. Addison-Wesley, 2001. Các mục 39–47 nói về các khía cạnh tinh tế của xử lý ngoại lệ trong Java.
- Foxall, James. *Practical Standards for Microsoft Visual Basic.NET*. Microsoft Press, 2003. Chương 10 nói về xử lý ngoại lệ trong Visual Basic.

Những điểm mấu chốt

- **Mã sản xuất phải xử lý lỗi tinh vi hơn nhiều so với kiểu "rác vào, rác ra" (garbage in, garbage out).**
- **Kỹ thuật lập trình phòng thủ giúp phát hiện, sửa lỗi dễ dàng và giảm hậu quả lỗi với hệ thống khi vận hành thực tế.**
- **Assertion giúp phát hiện lỗi sớm, đặc biệt hiệu quả trong hệ thống lớn, đòi hỏi độ tin cậy cao hoặc có mã thường xuyên thay đổi.**
- **Việc quyết định xử lý dữ liệu đầu vào xấu như thế nào là hai quyết định then chốt — cả trong xử lý lỗi lẫn thiết kế cấp cao.**
- **Ngoại lệ (exception) cung cấp một phương thức xử lý lỗi khác với luồng điều khiển thông thường, là công cụ hữu ích nếu sử dụng đúng mức và nên được cân nhắc so với các kỹ thuật khác.**
- **Những ràng buộc áp dụng với hệ thống sản xuất không nhất thiết phải dùng cho phiên bản phát triển. Bạn nên tận dụng để thêm mã hỗ trợ nhận diện lỗi cho phiên bản phát triển.**

Chương 9: Quá trình Lập trình Pseudocode (Pseudocode Programming Process)

Nội dung

- **9.2 Pseudocode cho lập trình viên chuyên nghiệp:** trang 218
- **9.3 Xây dựng routine với PPP:** trang 220
- **9.4 Các phương án thay thế PPP:** trang 232

Chủ đề liên quan

- Tạo lớp chất lượng cao: Chương 6
- Đặc điểm routine chất lượng cao: Chương 7
- Thiết kế trong quá trình kiến tạo: Chương 5
- Phong cách chú thích code: Chương 32

Mặc dù toàn bộ cuốn sách này có thể được xem là mô tả mở rộng về quy trình lập trình cho việc xây dựng lớp và routine, chương này đặt các bước đó vào một bối cảnh cụ thể. Trọng tâm của chương là việc lập trình "ở tầm nhỏ" – các bước cụ thể để xây dựng một lớp riêng lẻ cùng các routine, các bước này đều có vai trò quan trọng ở mọi quy mô dự án. Chương còn trình bày về Quá trình Lập trình Pseudocode (Pseudocode Programming Process - PPP), giúp giảm khối lượng công việc khi thiết kế và tài liệu hóa, đồng thời nâng cao chất lượng cho cả hai.

Nếu bạn là lập trình viên chuyên nghiệp, bạn có thể chỉ cần đọc lướt chương này, nhưng hãy xem tóm tắt các bước và các lời khuyên xây routine với PPP ở Mục 9.3. Ít lập trình viên khai thác triệt để sức mạnh của PPP, dù nó mang lại rất nhiều lợi ích.

PPP không phải là quy trình duy nhất để xây dựng lớp và routine. Mục 9.4 ở cuối chương sẽ giới thiệu thêm về các lựa chọn phổ biến khác, như phát triển theo hướng kiểm thử trước (test-first development) hoặc thiết kế theo hợp đồng (design by contract).

9.1 Tóm tắt các bước xây dựng lớp và routine

Việc xây dựng lớp có thể được tiếp cận bằng nhiều cách, nhưng thông thường, đó là một quy trình lặp lại giữa việc tạo một thiết kế tổng quát cho lớp, liệt kê các routine cụ thể bên trong lớp, xây dựng các routine chi tiết, kiểm tra và đánh giá tổng thể lớp. Như Hình 9-1 minh họa, quá trình xây dựng lớp có thể khá "lộn xộn", do bản chất "lộn xộn" sẵn có của quy trình thiết kế (như được trình bày ở Mục 5.1, "Các thách thức thiết kế").

```
Begin
Tạo thiết kế tổng quát
cho lớp
Xem xét và Kiểm tra
toàn bộ lớp trong khi
xây dựng routine chi tiết
Kết thúc
```

Hình 9-1: Các chi tiết trong xây dựng lớp có thể khác nhau, nhưng các hoạt động thường diễn ra theo trình tự như trên.

Các bước then chốt khi xây dựng lớp

1. **Tạo thiết kế tổng quát cho lớp:** Thiết kế lớp bao gồm nhiều yếu tố cụ thể.
2. **Xác định trách nhiệm cụ thể của lớp, "bí mật" mà lớp che giấu cùng các trừu tượng giao diện lớp đại diện.**
3. **Xác định xem lớp có kế thừa từ lớp khác hay không, và các lớp khác có thể kế thừa lại lớp này không.**

Lưu ý: Một số lỗi đánh máy, dấu câu thiếu hoặc sai ở bản gốc đã được chỉnh sửa lại trong bản dịch nhằm đảm bảo tính mạch lạc và dễ hiểu.

Lặp lại các chủ đề này nhiều lần nếu cần để xây dựng một thiết kế rõ ràng cho routine

Những cân nhắc này, cùng nhiều vấn đề khác, sẽ được thảo luận chi tiết hơn trong **Chương 6, "Làm việc với Classes"**.

9.1 Tóm tắt Các Bước Xây Dựng Classes và Routines

Xây dựng từng routine trong class
Sau khi đã xác định các routine chính của class ở bước đầu tiên, bạn cần triển khai từng routine cụ thể. Quá trình xây dựng mỗi routine thường sẽ làm phát sinh nhu cầu thiết kế thêm các routine bổ sung (bao gồm cả routine nhỏ và lớn), và các vấn đề liên quan đến việc tạo ra các routine này thường sẽ tác động ngược trở lại thiết kế tổng thể của class.

Rà soát và kiểm thử toàn bộ class
Thông thường, mỗi routine sẽ được kiểm thử ngay khi được tạo ra. Sau khi toàn bộ class đã vận hành được, việc rà soát và

kiểm thử tổng thể cho class là cần thiết nhằm phát hiện các vấn đề không thể kiểm tra ở cấp độ routine riêng lẻ.

Các Bước Xây Dựng một Routine

Nhiều routine trong một class sẽ rất đơn giản và dễ thực hiện—chẳng hạn như **accessor routines** (routine truy xuất), hoặc các routine chuyển tiếp tới routine của đối tượng khác. Tuy nhiên, cũng có những routine yêu cầu triển khai phức tạp hơn; việc xây dựng chúng càng được lợi nhiều từ một quy trình hệ thống.

Các hoạt động chính trong quá trình tạo routine—**thiết kế routine, kiểm tra thiết kế, viết code cho routine, kiểm tra code**—thường được thực hiện theo trình tự minh họa ở Hình 9-2.

```
Bắt đầu
Thiết kế routine
Kiểm tra thiết kế
Lặp lại nếu cần thiết
Viết và kiểm thử code của routine
Hoàn thành
```

Hình 9-2: Các hoạt động chính để xây dựng một routine thường được thực hiện theo thứ tự trên.

Các chuyên gia đã phát triển nhiều cách tiếp cận khác nhau để xây dựng routines, và phương pháp yêu thích của tôi là **Pseudocode Programming Process (PPP)**, sẽ được trình bày ở phần tiếp theo.

9.2 Pseudocode dành cho Chuyên gia

Pseudocode đề cập đến một ký hiệu xác định thuật toán, routine, class hoặc chương trình dưới dạng phi chính thức, gần với tiếng Anh tự nhiên. **Pseudocode Programming Process** (PPP) xác định một phương pháp sử dụng pseudocode để tối ưu hóa quá trình tạo code cho các routine.

Vì pseudocode giống với tiếng Anh, bạn có thể nghĩ rằng bất kỳ mô tả nào mang tính chất tiếng Anh đều mang lại hiệu quả tương tự. Tuy nhiên, thực tiễn chứng minh một số phong cách pseudocode hiệu quả hơn những phong cách khác. Để sử dụng pseudocode hiệu quả, hãy tuân theo các hướng dẫn sau:

- **Sử dụng câu tiếng Anh mô tả chính xác các thao tác cụ thể.**
- **Tránh các thành phần cú pháp của ngôn ngữ lập trình đích.** Pseudocode cho phép bạn thiết kế ở cấp độ trừu tượng cao hơn so với code thực tế. Nếu bạn sử dụng các cấu trúc của ngôn ngữ lập trình, bạn sẽ giảm cấp độ thiết kế, đánh mất lợi ích của việc thiết kế ở cấp cao, đồng thời tự đặt giới hạn bởi các quy tắc cú pháp không cần thiết.
- **Viết pseudocode ở mức độ mong muốn (intent).** Miêu tả ý nghĩa của phương pháp thay vì trình bày quá trình sẽ được hiện thực hóa trong ngôn ngữ đích.
- **Viết pseudocode chi tiết ở mức mà quá trình chuyển hóa thành code gần như tự động.** Nếu pseudocode quá trừu tượng, có thể bỏ sót các chi tiết quan trọng trong code. Tiếp tục bổ sung chi tiết cho đến khi pseudocode trở nên dễ dàng chuyển đổi thành code.

Sau khi pseudocode được viết xong, bạn phát triển code dựa trên nó và pseudocode này sẽ trở thành phần chú thích trong code. Nhờ đó, công việc chú thích được tối thiểu hóa—nếu pseudocode tuân thủ đúng hướng dẫn, các chú thích sẽ đầy đủ và có ý nghĩa.

Ví dụ: Pseudocode Kém

Dưới đây là một pseudocode vi phạm hầu hết các nguyên tắc trên:

```
increment resource number by 1
allocate a dlg struct using malloc
if malloc() returns NULL then return 1
invoke OSrsrc_init to initialize a resource for the operating system
*hRsrcPtr = resource number
return 0
```

Vấn đề:
Khối pseudocode này khó hiểu do sử dụng các chi tiết đặc thù của ngôn ngữ lập trình như *hRsrcPtr (cú pháp trỏ của C) và malloc() (hàm cụ thể của C). Pseudocode này tập trung vào cách viết code hơn là ý định thiết kế.

Ví dụ: Pseudocode Tốt

Thiết kế cho cùng nhiệm vụ ở mức pseudocode tốt hơn:

```
Theo dõi số lượng tài nguyên đang được sử dụng
Nếu còn tài nguyên khả dụng
```

```
    Cấp phát một cấu trúc hộp thoại
    Nếu cấu trúc hộp thoại được cấp phát thành công
        Ghi nhận thêm một tài nguyên được sử dụng
        Khởi tạo tài nguyên
        Lưu số hiệu tài nguyên vào vị trí do người gọi cung cấp
    Kết thúc nếu
Kết thúc nếu
Trả về true nếu đã tạo được tài nguyên mới; ngược lại trả về false
```

Pseudocode này tốt hơn ví dụ trước vì chỉ dùng tiếng Anh thông thường, không sử dụng bất kỳ thành phần nào của ngôn ngữ lập trình đích. Nó cũng được trình bày ở cấp độ ý định (intent). Dễ theo dõi hơn, và đủ chi tiết để dùng làm nền tảng chuyển sang code.

Lợi ích của Pseudocode

- **Pseudocode giúp quá trình rà soát (review) dễ dàng hơn:** Có thể xem xét thiết kế chi tiết mà không cần nhìn vào mã nguồn. Các cuộc kiểm tra thiết kế ở cấp thấp trở nên nhẹ nhàng, giảm nhu cầu kiểm tra từng dòng code.
- **Pseudocode hỗ trợ quy trình lặp tinh chỉnh (iterative refinement):** Bắt đầu bằng thiết kế tổng quát, dần chi tiết hóa thành pseudocode, rồi tiếp tục đến mã nguồn. Qua từng bước, bạn sẽ phát hiện lỗi ở đúng cấp độ và sửa chữa kịp thời trước khi lan sang các phần khác.
- **Pseudocode giúp việc thay đổi dễ dàng hơn:** Sửa đổi một vài dòng pseudocode đơn giản hơn nhiều so với việc chỉnh sửa cả trang mã đã viết. Nguyên lý sửa đổi sản phẩm ở giai đoạn "giá trị thấp nhất" (ít chi phí nhất) luôn đúng trong phát triển phần mềm.
- **Pseudocode giảm công sức chú thích:** Thay vì vừa viết code vừa bổ sung chú thích, các phát biểu pseudocode trở thành chú thích, dễ duy trì hơn và tiết kiệm công sức.
- **Pseudocode dễ duy trì hơn so với các dạng tài liệu thiết kế khác:** Với các phương pháp khác, thiết kế thường bị tách rời khỏi code; một khi một trong hai thay đổi, sự đồng bộ sẽ bị phá vỡ. Trong PPP, pseudocode được giữ trong code dưới dạng chú thích, luôn đồng nhất với code nếu được cập nhật.

  Lưu ý quan trọng: Theo khảo sát (Ramsey, Atwood và Van Doren, 1983), lập trình viên ưa thích pseudocode vì nó giúp triển khai code dễ dàng, dễ phát hiện các thiết kế thiếu chi tiết, và thuận tiện cho việc tài liệu hóa, bảo trì.

Pseudocode không phải là công cụ duy nhất, nhưng nó và PPP là những lựa chọn hiệu quả nên có trong bộ công cụ của lập trình viên.

9.3 Xây Dựng Routine Bằng PPP

Phần này mô tả các hoạt động khi xây dựng một routine, bao gồm:

- Thiết kế routine
- Viết code cho routine
- Kiểm tra code
- Hoàn thiện các vấn đề còn tồn đọng
- Lặp lại quy trình nếu cần

Thiết kế routine

Sau khi đã xác định các routine cho class, bước đầu tiên khi xây dựng bất kỳ routine phức tạp nào là thiết kế routine đó. Giả sử bạn cần viết một routine để xuất thông điệp lỗi dựa trên mã lỗi, với tên gọi `ReportErrorMessage()`. Sau đây là đặc tả không chính thức cho `ReportErrorMessage()`:

> `ReportErrorMessage()` nhận một mã lỗi dưới dạng đối số đầu vào và xuất ra thông điệp lỗi tương ứng với mã đó. Routine này chịu trách nhiệm xử lý các mã lỗi không hợp lệ. Nếu chương trình đang hoạt động ở chế độ tương tác, `ReportErrorMessage()` hiển thị thông báo tới người dùng; nếu ở chế độ dòng lệnh, routine sẽ ghi thông báo vào file thông điệp. Sau khi xuất thông báo, `ReportErrorMessage()` trả về giá trị trạng thái, chỉ ra thành công hay thất bại.

Phần còn lại của chương sẽ sử dụng routine này làm ví dụ minh họa và mô tả cách thiết kế routine.

Xác định vấn đề và đo lường yêu cầu

> Xem Chương 3, “Đo hai lần, cắt một lần: Các điều kiện tiên quyết ở mức thượng nguồn (Upstream Prerequisites),” và Chương 4, “Các quyết định then chốt về xây dựng (Key Construction Decisions)” để đảm bảo rằng routine thực sự được gọi theo yêu cầu dự án, ít nhất là gián tiếp.

Xác định vấn đề mà routine sẽ giải quyết

Hãy trình bày vấn đề mà routine sẽ giải quyết với đủ chi tiết để có thể xây dựng routine đó. Nếu thiết kế cấp cao (high-level design) đã đủ chi tiết thì bước này có thể đã được thực hiện. Thiết kế cấp cao nên thể hiện tối thiểu các nội dung sau:

- **Thông tin mà routine sẽ che giấu**
- **Dữ liệu đầu vào (inputs) cho routine**
- **Dữ liệu đầu ra (outputs) từ routine**
- **Điều kiện tiên quyết (preconditions) đảm bảo đúng trước khi gọi routine**
  (ví dụ: giá trị đầu vào trong phạm vi xác định, stream đã được khởi tạo, file đã mở hoặc đóng, buffer đã đầy hoặc đã làm sạch, v.v.)
- **Điều kiện hậu mãn (postconditions) mà routine đảm bảo đúng trước khi trả quyền điều khiển về cho routine gọi nó**
  (ví dụ: giá trị đầu ra trong phạm vi xác định, stream đã được khởi tạo, file đã mở hoặc đóng, buffer đã đầy hoặc đã làm sạch, v.v.)

  Tham khảo chéo: Để biết chi tiết về điều kiện tiên quyết (preconditions) và điều kiện hậu mãn (postconditions), xem mục "Sử dụng assert để mô tả và kiểm tra các điều kiện trước và sau" (Use assertions to document and verify preconditions and postconditions) tại Mục 8.2.

Ví dụ về xử lý các yếu tố trên trong ReportErrorMessage():

- Routine này che giấu hai thông tin: văn bản thông báo lỗi (error message text) và phương thức xử lý hiện tại (interactive hay command line).
- Không có điều kiện tiên quyết đảm bảo cho routine.
- Đầu vào của routine là mã lỗi (error code).
- Hai loại đầu ra được yêu cầu: thứ nhất là thông báo lỗi, thứ hai là trạng thái (status) mà `ReportErrorMessage()` trả về cho routine gọi nó.
- Routine đảm bảo giá trị trạng thái (status value) sẽ là Success hoặc Failure.

Đặt tên cho routine

Tham khảo chéo: Để biết chi tiết về cách đặt tên routine, xem Mục 7.3, "Tên routine tốt" (Good Routine Names).

Việc đặt tên routine có thể tưởng như đơn giản, nhưng một cái tên rõ ràng, không mơ hồ là dấu hiệu của một chương trình xuất sắc và đây là điều không dễ thực hiện. Nếu gặp khó khăn khi đặt tên, đó thường là do mục đích của routine chưa rõ ràng. Một tên gọi mơ hồ giống như một chính trị gia trên đường vận động tranh cử – nghe có vẻ như đang nói điều gì đó, nhưng khi kiểm tra kỹ, chẳng rõ ràng ý nghĩa. Nếu có thể làm tên gọi trở nên rõ ràng hơn, hãy thực hiện điều đó. Nếu tên gọi mơ hồ xuất phát từ thiết kế chung chung, bạn nên xem đó là cảnh báo và cải thiện lại thiết kế.

Trong ví dụ này, `ReportErrorMessage()` là tên không gây nhầm lẫn và là một tên tốt.

Quyết định cách kiểm thử routine

Đọc thêm: Đối với một phương pháp xây dựng chú trọng viết case kiểm thử trước, xem "Test-Driven Development: By Example" (Beck 2003).

Khi viết routine, hãy cân nhắc cách kiểm thử routine đó. Điều này hữu ích cho cả lập trình viên khi kiểm thử đơn vị (unit testing) và người kiểm thử độc lập. Trong ví dụ này, do dữ liệu đầu vào đơn giản, bạn có thể kiểm thử `ReportErrorMessage()` với tất cả mã lỗi hợp lệ và nhiều mã lỗi không hợp lệ.

Tìm hiểu các chức năng sẵn có trong thư viện chuẩn

Phương pháp quan trọng nhất để nâng cao cả chất lượng code lẫn năng suất là tái sử dụng code tốt có sẵn. Nếu nhận thấy routine bạn định thiết kế quá phức tạp, hãy tự hỏi liệu một phần hoặc toàn bộ chức năng bạn cần đã có sẵn trong thư viện của ngôn ngữ, nền tảng hoặc các công cụ mà bạn đang sử dụng hay chưa. Hãy kiểm tra cả thư viện code nội bộ của công ty. Nhiều thuật toán đã được phát minh, kiểm thử, thảo luận, đánh giá và cải thiện từ lâu. Thay vì tự phát minh lại bánh xe, hãy dành chút thời gian rà soát các thư viện sẵn có để đảm bảo bạn không làm việc dư thừa.

Suy nghĩ về xử lý lỗi

Hãy cân nhắc kỹ mọi vấn đề có thể xảy ra trong routine: giá trị đầu vào xấu, giá trị không hợp lệ được trả về từ routine khác, v.v. Routine có thể xử lý lỗi theo nhiều cách khác nhau, bạn cần lựa chọn phương pháp phù hợp. Nếu kiến trúc hệ thống đã

quy định chiến lược xử lý lỗi, bạn chỉ cần tuân theo chiến lược đó. Ngược lại, hãy chọn phương pháp phù hợp với routine cụ thể đó.

Suy nghĩ về hiệu suất (efficiency)

Tùy vào từng trường hợp, có hai hướng để giải quyết vấn đề hiệu suất:

1. **Trong đa số hệ thống**: hiệu suất không phải là yếu tố quan trọng nhất. Hãy đảm bảo giao diện routine trừu tượng tốt, code dễ đọc, để khi cần tối ưu sau này có thể thực hiện dễ dàng nhờ tính đóng gói (encapsulation). Bạn có thể thay thế một routine cũ thực hiện chậm hoặc tiêu tốn tài nguyên bằng một thuật toán tốt hơn hoặc bằng cài đặt ở ngôn ngữ cấp thấp, và không ảnh hưởng đến routine khác.
2. **Trong một số ít hệ thống**: hiệu suất là yếu tố quan trọng sống còn (ví dụ: kết nối cơ sở dữ liệu hạn chế, bộ nhớ hạn chế, số lượng handle hạn chế, thời gian đáp ứng ngắn, hoặc tài nguyên khan hiếm khác). Kiến trúc nên chỉ rõ số lượng tài nguyên mỗi routine (hoặc class) được phép sử dụng và tốc độ thực hiện mong muốn.

Hãy thiết kế routine sao cho đáp ứng yêu cầu về tài nguyên và tốc độ. Nếu một trong hai yếu tố này quan trọng hơn, hãy cân nhắc đánh đổi tài nguyên lấy tốc độ hoặc ngược lại. Trong giai đoạn xây dựng ban đầu, chỉ cần tối ưu vừa đủ sao cho routine đạt được mục tiêu về tài nguyên và tốc độ.

> Ngoài các phương án kể trên, thông thường không nên quá tập trung tối ưu hoá ở cấp độ routine riêng lẻ. Những cải thiện lớn thường đến từ tối ưu thiết kế cấp cao, chứ không phải routine riêng lẻ. Thường chỉ thực hiện tối ưu nhỏ lẻ (micro-optimization) khi thiết kế cấp cao không đáp ứng được yêu cầu hiệu năng và chỉ khi hệ thống đã hoàn thành mới xác định điều này. Đừng tốn thời gian cho các cải thiện vụn vặt khi chưa cần thiết.

Tìm hiểu thuật toán (algorithm) và kiểu dữ liệu (data type)

Nếu chức năng của routine chưa có trong thư viện sẵn có, hãy kiểm tra trong các sách về thuật toán. Trước khi bắt đầu viết code phức tạp từ đầu, hãy xem thử có thuật toán nền tảng nào có thể áp dụng không. Nếu dùng thuật toán có sẵn, đảm bảo bạn điều chỉnh nó đúng cho ngôn ngữ lập trình đang sử dụng.

Viết mã giả (pseudocode)

Quá trình chuẩn bị ở các bước trên chủ yếu giúp bạn có nhận thức tốt về routine trước khi bắt tay vào viết.

> *Tham khảo chéo*: Giả định rằng bạn đang sử dụng phương pháp thiết kế tốt để tạo ra phiên bản mã giả (pseudocode) của routine. Để biết thêm về thiết kế, xem Chương 5, “Thiết kế trong quá trình xây dựng” (Design in Construction).

Sau các bước chuẩn bị, bạn có thể bắt đầu viết routine bằng mã giả ở mức cao. Hãy sử dụng trình biên tập lập trình hoặc môi trường tích hợp (IDE) để viết mã giả, vì mã giả sẽ sớm trở thành nền tảng cho code thực sự.

Bắt đầu từ tổng quát rồi chuyển dần tới chi tiết. Phần tổng quát nhất là phần chú thích đầu (header comment) nêu rõ mục đích của routine. Trước tiên, hãy viết mô tả ngắn gọn về routine – điều này cũng giúp bạn làm rõ ý tưởng và mục đích của routine. Nếu gặp khó khăn trong việc viết chú thích tổng quát, bạn cần làm rõ hơn vai trò của routine trong chương trình.

> Nói chung, nếu khó tóm tắt vai trò của routine, bạn nên giả định rằng có điều gì đó chưa đúng trong thiết kế.

Ví dụ chú thích đầu cho routine

> Routine này xuất ra thông điệp lỗi dựa trên mã lỗi (error code) do routine gọi nó truyền vào. Cách xuất thông điệp tùy thuộc vào trạng thái xử lý hiện tại, và routine này tự xác định trạng thái đó. Routine trả về giá trị biểu thị thành công hoặc thất bại.

Sau khi viết chú thích tổng quát, hãy bổ sung mã giả ở mức cao cho routine.

Ví dụ mã giả cho routine

```
Routine này xuất ra thông điệp lỗi dựa trên mã lỗi do routine gọi truyền vào.
Cách xuất thông điệp phụ thuộc vào trạng thái xử lý hiện tại, routine sẽ tự xác định trạng thái này.
Routine trả về giá trị thể hiện thành công hoặc thất bại.

Đặt giá trị trạng thái (status) mặc định là "fail"
Tra cứu thông điệp dựa trên mã lỗi
Nếu mã lỗi hợp lệ
    Nếu đang xử lý tương tác (interactive), hiển thị thông điệp lỗi trên giao diện và xác nhận thành công
    Nếu đang xử lý dòng lệnh (command line), ghi lại thông điệp lỗi lên dòng lệnh và xác nhận thành công
```

```
Nếu mã lỗi không hợp lệ, thông báo cho người dùng biết có lỗi nội bộ
Trả về thông tin trạng thái
```

Một lần nữa, lưu ý rằng mã giả được viết ở mức khá cao, không phải là ngôn ngữ lập trình cụ thể mà dùng tiếng Anh chính xác để biểu đạt yêu cầu của routine.

Suy nghĩ về dữ liệu

Bạn có thể thiết kế dữ liệu cho routine ở nhiều điểm khác nhau trong quá trình trên. Trong ví dụ này, dữ liệu đơn giản và xử lý dữ liệu không phải là phần trọng tâm. Tuy nhiên, nếu xử lý dữ liệu là chủ đạo, bạn nên xác định các dữ liệu quan trọng trước khi thiết kế logic cho routine. Định nghĩa rõ các kiểu dữ liệu then chốt sẽ rất hữu ích khi bạn xây dựng logic của routine.

> *Tham khảo*: Để sử dụng biến hiệu quả, xem các Chương 10 đến 13.

Lùi lại và suy ngẫm

Chương 21: “Cộng tác Xây dựng” (Collaborative Construction)

Hãy suy nghĩ về cách bạn sẽ giải thích quá trình này cho người khác. Hãy nhờ ai đó xem xét hoặc lắng nghe bạn trình bày. Có thể bạn sẽ nghĩ rằng việc nhờ người khác xem xét 11 dòng mã giả (pseudocode) là điều ngớ ngẩn, nhưng bạn sẽ ngạc nhiên vì mã giả giúp làm rõ các giả định và sai sót ở cấp độ cao dễ dàng hơn so với mã lập trình thực tế. Bên cạnh đó, mọi người thường sẵn lòng xem xét vài dòng mã giả hơn là 35 dòng mã C++ hoặc Java.

9.3 Xây dựng các routine bằng PPP (Pseudocode Programming Process)

Hãy đảm bảo bạn hiểu rõ một cách dễ dàng và thoải mái về mục đích và cách thực hiện của routine. Nếu bạn chưa hiểu rõ về routine ở cấp độ mã giả, thì xác suất bạn hiểu tại cấp độ mã lập trình sẽ càng thấp. Và nếu bạn không hiểu, thì ai sẽ hiểu?

Tham khảo chéo

> Để biết thêm về lặp lại, hãy xem Mục 34.8, “Lặp đi lặp lại, nhiều lần liên tiếp” (Iterate, Repeatedly, Again and Again).

Hãy thử nghiệm ý tưởng trên mã giả, và giữ lại giải pháp tốt nhất (lặp lại). Hãy thử càng nhiều ý tưởng càng tốt trên mã giả trước khi bắt đầu lập trình thực tế. Một khi bạn đã viết mã, bạn sẽ có xu hướng gắn bó về mặt cảm xúc với mã của mình, khiến việc loại bỏ một thiết kế kém hiệu quả và bắt đầu lại trở nên khó khăn hơn.

Ý tưởng chung là hãy lặp lại routine trên mã giả cho đến khi các câu lệnh mã giả đơn giản đủ để bạn có thể triển khai mã thực phía dưới mỗi câu lệnh đó, đồng thời giữ nguyên mã giả để làm tài liệu. Một phần mã giả ở lần thử đầu tiên của bạn có thể vẫn còn ở mức độ quá cao, yêu cầu bạn phải phân rã thêm. Hãy chắc chắn thực hiện việc đó. Nếu bạn chưa chắc chắn cách viết mã cho một phần nào đó, hãy tiếp tục làm việc với mã giả cho đến khi bạn thật sự chắc chắn. Tiếp tục tinh chỉnh và phân rã mã giả cho đến khi việc viết mã giả thay vì mã thực trở nên không còn cần thiết nữa.

Bắt đầu xây dựng routine

Khi bạn đã thiết kế routine, hãy tiến hành xây dựng nó. Bạn có thể thực hiện các bước xây dựng theo một trình tự gần như tiêu chuẩn, nhưng cũng có thể điều chỉnh linh hoạt khi cần thiết. Hình 9-3 trình bày các bước xây dựng một routine.

- **Bắt đầu với mã giả**
- **Viết khai báo routine**
- **Viết câu lệnh đầu và cuối, chuyển mã giả thành chú thích cấp cao**
- **Lặp lại nếu cần thiết**
- **Hoàn thiện mã phía dưới mỗi chú thích**
- **Kiểm tra mã**
- **Dọn dẹp các phần còn sót lại**
- **Hoàn thành**

> **Hình 9-3**: Bạn sẽ thực hiện tất cả các bước này khi thiết kế routine, nhưng không nhất thiết phải theo trình tự cố định.

Ghi chú về khai báo routine

Hãy viết câu lệnh khai báo interface của routine—tuyên bố hàm trong C++, khai báo method trong Java, hoặc khai báo function/sub procedure trong Microsoft Visual Basic, hoặc theo các yêu cầu của ngôn ngữ sử dụng. Chuyển phần chú thích

đầu (header comment) gốc thành chú thích trong ngôn ngữ lập trình, đặt chúng phía trên mã giả đã viết. Dưới đây là ví dụ về khai báo interface và phần chú thích đầu routine trong C++:

Ví dụ C++ về interface của routine và phần đầu chú thích

```
/* This routine outputs an error message based on an error code
   supplied by the calling routine. The way it outputs the message
   depends on the current processing state, which it retrieves
   on its own. It returns a value indicating success or failure
*/
Status ReportErrorMessage(
    ErrorCode errorToReport
)
set the default status to "fail"
look up the message based on the error code
if the error code is valid
    if doing interactive processing, display the error message
        interactively and declare success
    if doing command line processing, log the error message to the
        command line and declare success
if the error code isn't valid, notify the user that an
    internal error has been detected
return status information
```

Ở giai đoạn này, đây là thời điểm thích hợp để ghi chú về các giả định interface. Trong trường hợp này, biến interface `error` khá rõ ràng và đã được gán kiểu dữ liệu cụ thể, nên không cần tài liệu bổ sung.

Chuyển mã giả thành chú thích cấp cao

Tiếp tục viết câu lệnh đầu và cuối: `{` và `}` trong C++. Sau đó, chuyển mã giả thành chú thích. Ví dụ minh họa như sau:

Ví dụ C++: Viết câu lệnh đầu và cuối quanh mã giả

```
/* This routine outputs an error message based on an error code
   supplied by the calling routine. The way it outputs the message
   depends on the current processing state, which it retrieves
   on its own. It returns a value indicating success or failure
*/
Status ReportErrorMessage(
    ErrorCode errorToReport
) {
    // set the default status to "fail"
    // look up the message based on the error code
    // if the error code is valid
        // if doing interactive processing, display the error message
            // interactively and declare success
        // if doing command line processing, log the error message to the
            // command line and declare success
    // if the error code isn't valid, notify the user that an
        // internal error has been detected
    // return status information
}
```

Ở thời điểm này, bạn đã có thể cảm nhận được cách hoạt động của routine, ngay cả khi chưa có dòng mã thực nào. Khi chuyển đổi mã giả sang mã lập trình nên là một quá trình tự nhiên, cơ học và dễ dàng. Nếu chưa đạt được cảm giác này, hãy tiếp tục thiết kế trên mã giả cho tới khi bạn cảm thấy bản thiết kế đã vững chắc.

Hoàn thiện mã phía dưới mỗi chú thích

> Đây là trường hợp phù hợp với phép ẩn dụ viết văn—trong phạm vi nhỏ. Bạn viết đề cương, sau đó phát triển từng đoạn chi tiết dựa trên các ý chính; tương tự, mỗi chú thích từ mã giả mô tả một khối hoặc đoạn mã. Độ dài của từng khối mã cũng tương tự như độ dài các đoạn văn, phụ thuộc vào nội dung và mức độ tập trung của ý tưởng.

Trong ví dụ sau, hai chú thích mã giả đầu tiên được hiện thực hóa thành hai dòng mã:

Ví dụ C++: Chuyển các chú thích mã giả thành code

```
/* This routine outputs an error message based on an error code
   supplied by the calling routine. The way it outputs the message
   depends on the current processing state, which it retrieves
   on its own. It returns a value indicating success or failure
*/
Status ReportErrorMessage(
```

```
    ErrorCode errorToReport
) {
    // set the default status to "fail"
    Status errorMessageStatus = Status_Failure;

    // look up the message based on the error code
    Message errorMessage = LookupErrorMessage(errorToReport);

    // if the error code is valid
    // if doing interactive processing, display the error message
        // interactively and declare success
    // if doing command line processing, log the error message to the
        // command line and declare success
    // if the error code isn't valid, notify the user that an
        // internal error has been detected
    // return status information
}
```

Ở bước này, biến `errorMessage` được sử dụng nên bạn cần khai báo nó. Nếu bạn chỉ viết chú thích sau khi mã đã hoàn thành thì hai dòng chú thích cho hai dòng mã có thể là dư thừa. Tuy nhiên, trong phương pháp này, điều quan trọng là nội dung ý nghĩa của từng chú thích, không phải số lượng dòng. Các chú thích giúp giải thích rõ ý định của code, vì vậy hãy giữ chúng lại.

Bạn cần hoàn thiện mã dưới mỗi chú thích còn lại.

Ví dụ C++: Một routine hoàn chỉnh theo PPP

```
/* This routine outputs an error message based on an error code
   supplied by the calling routine. The way it outputs the message
   depends on the current processing state, which it retrieves
   on its own. It returns a value indicating success or failure
*/
Status ReportErrorMessage(
    ErrorCode errorToReport
) {
    // set the default status to "fail"
    Status errorMessageStatus = Status_Failure;

    // look up the message based on the error code
    Message errorMessage = LookupErrorMessage(errorToReport);

    // if the error code is valid
    if (errorMessage.ValidCode()) {
        // determine the processing method
        ProcessingMethod errorProcessingMethod = CurrentProcessingMethod();

        // if doing interactive processing, display the error message
        // interactively and declare success
        if (errorProcessingMethod == ProcessingMethod_Interactive) {
            DisplayInteractiveMessage(errorMessage.Text());
            errorMessageStatus = Status_Success;
        }
        // if doing command line processing, log the error message to the
        // command line and declare success
        else if (errorProcessingMethod == ProcessingMethod_CommandLine) {
            CommandLine messageLog;
            if (messageLog.Status() == CommandLineStatus_Ok) {
                messageLog.AddToMessageQueue(errorMessage.Text());
                messageLog.FlushMessageQueue();
                errorMessageStatus = Status_Success;
            }
        }
        else {
            // can't do anything because the routine is already error processing
        }
    }
    else {
        // can't do anything because the routine is already error processing
    }
    // if the error code isn't valid, notify the user that an
    // internal error has been detected
    else {
        DisplayInteractiveMessage(
            "Internal Error: Invalid error code in ReportErrorMessage()"
        );
    }
    // return status information
    return errorMessageStatus;
}
```

Mỗi chú thích dẫn tới một hoặc nhiều dòng mã, và mỗi khối mã là một ý tưởng hoàn chỉnh dựa trên chú thích đó. Việc giữ lại các chú thích giúp giải thích rõ ràng ở mức độ tổng quát hơn cho code bên dưới.

Mỗi chú thích nên mở rộng thành khoảng 2 đến 10 dòng mã

Thông thường, mỗi chú thích nên mở rộng thành khoảng từ 2 đến 10 dòng code. (*Vì ví dụ này chỉ mang tính minh họa, việc mở rộng mã code ở đây thấp hơn so với thực tế mà bạn sẽ thường gặp.*)

Hãy xem lại phần đặc tả (specification) ở trang 000 và mã giả (pseudocode) ban đầu ở trang 000. Đặc tả ban đầu gồm năm câu đã được mở rộng thành 15 dòng mã giả (phụ thuộc vào cách bạn đếm dòng), và sau đó lại được phát triển thành một thủ tục có độ dài gần một trang. Mặc dù đặc tả khá chi tiết, việc tạo ra thuật toán đòi hỏi phải thiết kế đáng kể trong cả mã giả lẫn code thực tế. Thiết kế ở mức thấp như vậy là một trong những lý do khiến "việc lập trình" không phải là một nhiệm vụ tầm thường, và cũng vì vậy mà chủ đề của cuốn sách này trở nên quan trọng.

Kiểm tra xem mã có cần tiếp tục phân tách hay không

Trong một số trường hợp, bạn sẽ thấy lượng code phát sinh dưới một trong các dòng mã giả ban đầu là khá lớn. Trong trường hợp như vậy, bạn nên cân nhắc một trong hai hướng xử lý sau:

- **Tách mã dưới chú thích thành một routine (thủ tục) mới**
  Nếu bạn thấy một dòng mã giả mở rộng thành nhiều code hơn mong đợi, hãy tách đoạn code đó thành một routine riêng. Viết phần code gọi đến routine vừa tách, bao gồm tên routine. Nếu bạn đã áp dụng tốt Quy trình lập trình bằng mã giả (PPP - Pseudocode Programming Process), tên routine mới thường sẽ phát sinh tự nhiên từ mã giả. Khi hoàn thiện routine ban đầu, bạn có thể tiếp tục áp dụng PPP cho routine mới này.

- **Áp dụng PPP một cách đệ quy**
  Thay vì viết vài chục dòng code phía dưới một dòng mã giả, hãy dành thời gian phân rã dòng mã giả đó thành nhiều dòng mã giả nhỏ hơn. Sau đó, tiếp tục viết code cho từng dòng mã giả mới này.

  Tham khảo:
  Để tìm hiểu kỹ hơn về tái cấu trúc (refactoring), xem Chương 24, "Refactoring".

Kiểm tra mã

Sau khi thiết kế và triển khai routine, bước lớn thứ ba trong quá trình xây dựng là kiểm tra xem những gì bạn tạo ra có chính xác không. Bất kỳ lỗi nào bị bỏ sót ở giai đoạn này đều sẽ chỉ được phát hiện khi kiểm thử sau này, và chi phí phát hiện cùng sửa chữa sẽ tăng cao hơn nhiều. Vì thế, bạn nên cố gắng tìm ra mọi lỗi ngay ở giai đoạn này.

> **Tham khảo:**
> Để biết chi tiết về kiểm tra lỗi trong kiến trúc và yêu cầu, xem Chương 3, "Measure Twice, Cut Once: Upstream Prerequisites."

Một số lỗi chỉ xuất hiện khi routine đã được lập trình đầy đủ vì nhiều lý do: lỗi ở mã giả có thể trở nên dễ thấy hơn khi triển khai chi tiết; một thiết kế trông có vẻ tinh tế trên mã giả lại có thể vụng về khi hiện thực hóa; làm việc với hiện thực chi tiết có thể bộc lộ các lỗi từ kiến trúc, thiết kế cấp cao, hoặc yêu cầu; cuối cùng là các lỗi coding thông thường—không ai hoàn hảo cả! Vì vậy, hãy xem lại mã (code) trước khi chuyển sang giai đoạn tiếp theo.

Kiểm tra tinh thần đối với routine

Bước kiểm tra hình thức đầu tiên đối với routine là kiểm tra tinh thần (mental checking). Các bước dọn dẹp và kiểm tra không chính thức đã nhắc đến trước đó đều là các loại kiểm tra tinh thần. Một loại khác là tự thực thi từng đường đi (path) trong đầu. Việc kiểm tra này khá khó, đó cũng là lý do nên giữ routine nhỏ gọn. Hãy đảm bảo bạn kiểm tra cả các đường đi chính (nominal path), các điểm kết thúc, và mọi tình huống ngoại lệ. Thực hiện việc này cả một mình ("desk checking") và cùng đồng nghiệp ("peer review", "walk-through", hoặc "inspection" tùy cách thức tiến hành).

> Một trong những khác biệt lớn nhất giữa những người tự học lập trình (hobbyists) và lập trình viên chuyên nghiệp là việc chuyển từ niềm tin cảm tính (superstition) sang sự hiểu biết thực sự. Trong ngữ cảnh này, "superstition" (niềm tin cảm tính) không ám chỉ những chương trình kỳ dị, mà là việc thay thế sự hiểu biết bằng cảm xúc đối với code. Nếu bạn thường xuyên nghi ngờ compiler (bộ biên dịch) hoặc phần cứng gây ra lỗi, bạn vẫn còn ở mức độ "superstition". Một nghiên cứu nhiều năm trước cho thấy chỉ khoảng 5% lỗi đến từ phần cứng, compiler hoặc hệ điều hành (Ostrand và Weyuker, 1984); ngày nay, con số này còn thấp hơn nữa. Lập trình viên chuyên nghiệp luôn nghi ngờ chính công việc của họ đầu tiên, vì họ biết 95% lỗi đến từ chính lập trình viên. Hãy hiểu vai trò của từng dòng code và lý do tồn tại của nó. Không có gì "đúng" chỉ vì nó đang chạy. Nếu bạn không biết tại sao nó chạy được, có lẽ thực ra nó chưa chạy đúng—chỉ là bạn chưa nhận ra mà thôi.

Tóm lại: Một routine chạy được là chưa đủ. Nếu bạn chưa hiểu rõ vì sao nó hoạt động tốt, hãy nghiên cứu, thảo luận và thử nghiệm các thiết kế thay thế cho đến khi bạn nắm chắc nguyên tắc vận hành.

Biên dịch routine

Sau khi đã review routine, hãy biên dịch (compile) nó. Có thể bạn sẽ cho là chờ tới lúc này mới biên dịch thì hơi không hiệu quả, bởi code có thể đã hoàn thiện cách đây vài trang. Thật ra, bạn có thể tiết kiệm được một chút thời gian nếu biên dịch sớm để máy kiểm tra các biến chưa khai báo, xung đột tên, v.v.

Bạn sẽ thu được nhiều lợi ích nếu trì hoãn biên dịch cho đến các bước sau cùng. Nguyên nhân chủ yếu là sau lần biên dịch đầu tiên, một chiếc "đồng hồ bấm giờ" bên trong bạn sẽ bắt đầu đếm ngược. Khi đó, bạn sẽ có xu hướng "Chỉ biên dịch thêm lần nữa là xong", dẫn đến sự hấp tấp, thay đổi vội vàng và phát sinh thêm nhiều lỗi cũng như tốn nhiều thời gian hơn về lâu về dài. Hãy tránh vội vàng kết thúc bằng cách chỉ biên dịch khi bạn đã thật sự tin tưởng routine đã đúng.

Điểm mấu chốt của cuốn sách này là giúp bạn vượt ra khỏi vòng luẩn quẩn "nối ghép vội vàng rồi chạy thử xem có được không". Việc biên dịch quá sớm khi bạn chưa dám chắc đầu ra hợp lý là dấu hiệu của tư duy hacker chưa chuyên nghiệp. Nếu bạn đã thoát khỏi vòng lặp "hacking, compiling, and fixing", hãy biên dịch tại thời điểm phù hợp, nhưng hãy luôn ý thức sức ép tự nhiên hướng tới việc "hacking, compiling and fixing" một cách thiếu bài bản.

Một số hướng dẫn để tận dụng tối đa quá trình biên dịch routine:

- Đặt mức cảnh báo (warning level) của compiler lên mức nghiêm ngặt nhất. Bạn có thể phát hiện được rất nhiều lỗi tinh vi chỉ bằng cách để compiler tìm chúng.
- Sử dụng các công cụ xác thực (validator). Ở các ngôn ngữ như C, việc kiểm tra của compiler có thể được bổ trợ bởi các công cụ như lint. Ngay cả với các loại code không biên dịch như HTML và JavaScript, cũng nên sử dụng công cụ xác thực (validator).
- Loại bỏ hết các lỗi và cảnh báo. Hãy chú ý đến thông điệp cảnh báo từ compiler. Nếu có nhiều cảnh báo, đó thường là dấu hiệu chất lượng code thấp; bạn nên cố gắng hiểu lý do của từng cảnh báo. Thực tế, những cảnh báo hay gặp sẽ dẫn tới hai hệ quả: bạn bỏ qua chúng và vô tình bỏ sót các cảnh báo nguy hiểm hơn, hoặc chúng trở nên phiền phức. Giải pháp an toàn và hiệu quả là viết lại mã để khắc phục tận gốc, loại bỏ hoàn toàn các cảnh báo.

Thực thi từng bước với trình gỡ lỗi (debugger)

Khi routine đã biên dịch thành công, hãy đưa nó vào debugger và thực thi từng bước một. Đảm bảo rằng từng dòng code chạy đúng như mong đợi. Chỉ với thao tác đơn giản này, bạn có thể phát hiện ra rất nhiều lỗi.

> **Tham khảo:**
> Để biết chi tiết, xem Chương 22, "Developer Testing". Xem thêm mục "Building Scaffolding to Test Individual Classes" tại Mục 22.5.

Kiểm thử routine

Kiểm thử routine bằng các bộ test case (trường hợp kiểm thử) đã được lên kế hoạch hoặc xây dựng trong quá trình phát triển routine. Bạn có thể cần phát triển các đoạn mã bổ trợ (scaffolding) để hỗ trợ quá trình kiểm thử—đây là những đoạn mã dùng trong quá trình kiểm thử routine nhưng không được đưa vào sản phẩm cuối cùng. Scaffolding có thể là các routine kiểm thử gọi routine của bạn với dữ liệu kiểm thử (test harness) hoặc các stub được gọi bởi routine của bạn.

> **Tham khảo:**
> Để biết chi tiết, xem Chương 23, "Debugging".

Loại bỏ lỗi khỏi routine

Sau khi phát hiện ra lỗi, bạn cần loại bỏ chúng. Nếu routine đang phát triển xuất hiện nhiều lỗi ở giai đoạn này, khả năng cao là nó sẽ tiếp tục mắc lỗi trong tương lai. Nếu routine quá nhiều lỗi, hãy bắt đầu lại từ đầu; đừng "vá víu" tạm bợ. Việc "hack" chỉ phản ánh sự hiểu biết chưa đầy đủ và sẽ đảm bảo tiếp diễn lỗi trong cả hiện tại lẫn tương lai. Thiết kế lại hoàn toàn một routine nhiều lỗi thực sự đáng giá. Ít điều gì thỏa mãn hơn việc viết lại một routine từng gây rắc rối, và sau đó không còn thấy lỗi nào nữa.

Dọn Dẹp Sau Cùng (Clean Up Leftovers)

Khi bạn đã kiểm tra code để phát hiện vấn đề, hãy kiểm tra nó thêm một lần nữa theo các tiêu chí chất lượng tổng quát đã trình bày trong toàn cuốn sách. Bạn có thể thực hiện một số bước dọn dẹp để đảm bảo routine đạt đến những tiêu chuẩn mong muốn:

- **Kiểm tra interface của routine:** Đảm bảo tất cả dữ liệu vào/ra đều đã được xét đến và tất cả các tham số (parameter) đều được sử dụng. Để biết thêm chi tiết, xem Mục 7.5, “How to Use Routine Parameters”.
- **Kiểm tra chất lượng thiết kế tổng thể:** Đảm bảo routine chỉ thực hiện một việc và thực hiện tốt việc đó, liên kết lỏng lẻo (loose coupling) với các routine khác, và có thiết kế phòng thủ (defensive design). Tham khảo Chương 7, “High-Quality Routines”.
- **Kiểm tra biến của routine:** Xem lại các tên biến không chính xác, đối tượng không dùng đến, biến chưa khai báo, đối tượng khởi tạo không đúng, v.v. Xem chi tiết trong các chương 10 đến 13, về sử dụng biến.
- **Kiểm tra các lệnh (statement) và logic:** Tìm lỗi “lệch một đơn vị” (off-by-one), vòng lặp vô hạn, lồng ghép không hợp lý, rò rỉ tài nguyên, v.v. Tham khảo các chương từ 14 đến 19.
- **Kiểm tra bố cục (layout) routine:** Đảm bảo bạn sử dụng khoảng trắng để làm rõ cấu trúc logic của routine, biểu thức, và danh sách tham số. Xem thêm Chương 31, “Layout and Style”.
- **Kiểm tra tài liệu chú thích:** Đảm bảo phần mã giả đã chuyển thành chú thích trong code vẫn còn chính xác; kiểm tra mô tả thuật toán, tài liệu về giả định giao diện (interface assumption) và các phụ thuộc không rõ ràng, lý do cho các thói quen lập trình không rõ ràng, v.v. Tham khảo Chương 32, “Self-Documenting Code”.
- **Loại bỏ các chú thích thừa.**

Lặp lại các bước khi cần thiết

Nếu chất lượng của routine (thủ tục) kém, hãy quay lại bước viết pseudocode (mã giả). **Lập trình chất lượng cao là một quá trình lặp**, do đó, đừng ngần ngại thực hiện lại các hoạt động xây dựng.

9.4 Các phương pháp thay thế PPP

Theo quan điểm cá nhân, phương pháp PPP (Pseudocode Programming Process – Quy trình Lập trình bằng Mã giả) là phương pháp tốt nhất để tạo ra các class (lớp) và routine (thủ tục/chức năng). Tuy nhiên, dưới đây là một số phương pháp khác được các chuyên gia khuyến nghị. Bạn có thể sử dụng các phương pháp này như những lựa chọn thay thế, hoặc bổ sung cho PPP.

Test-first development (Phát triển hướng kiểm thử trước)

Test-first development là một phong cách phát triển phổ biến, trong đó các test case (trường hợp kiểm thử) được viết **trước** khi viết bất kỳ đoạn code nào. Cách tiếp cận này được mô tả chi tiết hơn trong mục “Test First or Test Last?” tại Phần 22.2. Một cuốn sách hay về *test-first programming* là **Test-Driven Development: By Example** của Kent Beck (Beck 2003).

Refactoring (Tái cấu trúc mã nguồn)

Refactoring là một phương pháp phát triển, trong đó bạn cải tiến code thông qua loạt các biến đổi giữ nguyên ý nghĩa (semantic preserving transformations). Lập trình viên sử dụng các mẫu nhận diện code kém hay còn gọi là “smells” (mùi mã nguồn yếu kém), để xác định các phần code cần cải thiện. Chương 24, “Refactoring”, mô tả chi tiết phương pháp này. Một cuốn sách tốt về chủ đề này là **Refactoring: Improving the Design of Existing Code** của Martin Fowler (Fowler 1999).

Design by contract (Thiết kế theo hợp đồng)

Design by contract là một phương pháp phát triển, trong đó mỗi routine được xem xét theo các điều kiện tiên quyết (preconditions) và điều kiện hậu kiểm (postconditions). Cách tiếp cận này được trình bày trong mục “Use assertions to document and verify preconditions and postconditions” tại Phần 8.2. Nguồn thông tin tốt nhất về *design by contract* là cuốn **Object-Oriented Software Construction** của Bertrand Meyer (Meyer 1997).

Hacking? (“Code chay” không có hệ thống)

Một số lập trình viên cố gắng “hack” để có code hoạt động thay vì sử dụng các phương pháp hệ thống như PPP. Nếu bạn từng rơi vào tình huống phải viết lại một routine do lâm vào ngõ cụt, đó chính là dấu hiệu cho thấy PPP có thể hữu ích hơn. Nếu bạn thường xuyên bị mất luồng suy nghĩ khi đang code một routine, đó cũng là một dấu hiệu cho thấy PPP sẽ mang lại lợi ích. Bạn đã bao giờ “quên” viết một phần của class hoặc routine chưa? Nếu sử dụng PPP, điều này hiếm khi xảy ra. Nếu bạn cảm thấy lúng túng không biết bắt đầu từ đâu trước màn hình máy tính, đó là dấu hiệu rõ ràng PPP sẽ giúp công việc lập trình của bạn dễ dàng hơn.

---

cc2e.com/0943 CHECKLIST: The Pseudocode Programming Process

> Mục đích của danh sách này là kiểm tra xem bạn có tuân thủ đúng các bước cần thiết để xây dựng một routine hay không. Đối với một checklist tập trung vào chất lượng của routine, tham khảo thêm mục “High-Quality Routines” ở Chương 7, trang 185.

Checklist:

- ❑ Bạn đã kiểm tra liệu các điều kiện tiên quyết đã được đáp ứng chưa?
- ❑ Bạn đã xác định vấn đề mà class sẽ giải quyết chưa?
- ❑ Thiết kế cấp cao có đủ rõ ràng để đặt tên hợp lý cho class và từng routine chưa?
- ❑ Bạn đã suy nghĩ về cách kiểm thử class và từng routine chưa?
- ❑ Bạn đã cân nhắc hiệu năng chủ yếu dưới góc độ giao diện ổn định và triển khai dễ đọc, hay chủ yếu dựa theo tiêu chí tài nguyên và tốc độ?
- ❑ Bạn đã kiểm tra các thư viện chuẩn và thư viện mã nguồn khác để tìm các routine hoặc component (thành phần) phù hợp chưa?
- ❑ Bạn đã tra cứu các sách tham khảo để tìm các thuật toán hữu ích chưa?