Phần VI: Các Khía Cạnh Hệ Thống

Nội dung phần này:

- **Chương 27:** Ảnh hưởng của Quy mô Chương trình đến Quá trình Xây dựng
- **Chương 28:** Quản lý Quá trình Xây dựng
- **Chương 29:** Tích hợp
- **Chương 30:** Công cụ Lập trình

Chương 27

Ảnh hưởng của Quy mô Chương trình đến Quá trình Xây dựng

Chủ đề liên quan:

- Các điều kiện tiên quyết của quá trình xây dựng: Chương 3
- Xác định loại phần mềm bạn đang phát triển: Mục 3.2
- Quản lý xây dựng: Chương 28

Mở rộng quy mô trong phát triển phần mềm không đơn giản chỉ là lấy một dự án nhỏ và mở rộng từng phần của nó lên. Giả sử bạn đã viết gói phần mềm Gigatron gồm 25.000 dòng mã trong 20 nhân-tháng và phát hiện 500 lỗi trong quá trình kiểm thử thực địa. Nếu Gigatron 1.0 thành công, và cả Gigatron 2.0 cũng vậy, rồi bạn bắt đầu làm việc trên Gigatron Deluxe – một phiên bản nâng cấp vượt trội với kỳ vọng đạt 250.000 dòng mã. Dù Gigatron Deluxe lớn gấp 10 lần bản gốc, quá trình phát triển sẽ không chỉ tốn gấp 10 lần công sức; mà sẽ phải tốn đến gấp 30 lần nỗ lực. Hơn nữa, tổng nỗ lực tăng 30 lần không chỉ đồng nghĩa với số lượng công việc xây dựng cũng tăng gấp 30; mà thực tế, có thể tăng trưởng theo tỷ lệ 25 lần xây dựng, 40 lần kiến trúc và kiểm thử hệ thống.

Số lỗi cũng không chỉ tăng gấp 10 lần; mà rất có thể sẽ gấp 15 lần hoặc hơn thế nữa. Nếu bạn quen làm các dự án nhỏ, thì dự án quy mô vừa hoặc lớn đầu tiên có thể sẽ trở nên mất kiểm soát, biến thành "quái thú" khó kiểm soát thay vì thành công như mong đợi. Chương này sẽ giúp bạn hình dung đặc điểm của "quái thú" đó và cách chế ngự nó. Ngược lại, nếu bạn quen làm các dự án lớn, bạn có thể triển khai những phương pháp quá chính quy cho dự án nhỏ, khiến dự án nhỏ gục ngã dưới sức nặng của chính các thủ tục đó.

27.1 Giao tiếp và Quy mô

Nếu bạn là người duy nhất trong dự án, con đường giao tiếp duy nhất là giữa bạn và khách hàng (nếu không tính đường truyền qua corpus callosum – kết nối giữa hai bán cầu não của bạn). Khi số người tham gia dự án tăng lên, số đường giao tiếp cũng tăng theo. Không chỉ đơn giản tăng cộng; mà sẽ tăng theo bội số, tỷ lệ thuận với bình phương số thành viên. Hình 27-1 minh họa điều này:

- Dự án hai người chỉ có một kênh giao tiếp.

- Dự án năm người có 10 kênh giao tiếp.
- Dự án mười người có 45 kênh giao tiếp, nếu mỗi người giao tiếp với mọi người còn lại.
- 10% dự án có 50 thành viên trở lên sẽ có ít nhất 1.200 kênh khả dĩ.

Ý chính: Càng nhiều đường giao tiếp, bạn càng mất nhiều thời gian để trao đổi, kéo theo đó là nguy cơ hiểu nhầm hoặc sai sót cũng tăng lên. Các dự án lớn đòi hỏi kỹ thuật tổ chức để đơn giản hóa hoặc hợp lý hóa quá trình giao tiếp.

> Cách tiếp cận thường thấy để tinh giản giao tiếp là chuẩn hóa nó thông qua tài liệu (documents). Thay vì 50 người nói chuyện với nhau theo mọi tổ hợp có thể, tất cả sẽ đọc và viết tài liệu (dạng văn bản hoặc đồ họa, in ra giấy hoặc lưu điện tử).

27.2 Dải Quy mô Dự án

Liệu quy mô dự án bạn đang làm có phổ biến? Trên thực tế, phạm vi quy mô dự án rất đa dạng, khiến chúng ta không thể coi một quy mô nhất định nào là "chuẩn chung". Một cách phân loại quy mô dự án là xem theo số lượng thành viên nhóm.

Tỷ lệ dự án theo quy mô nhóm (ước lượng): | Quy mô nhóm | Tỷ lệ dự án ước lượng | |----------|----------------------| | 1–3 | 25% | | 4–10 | 30% | | 11–25 | 20% | | 26–50 | 15% | | 50+ | 10% |

Lưu ý: Các dự án lớn hơn thường sử dụng nhiều lập trình viên hơn nên chiếm tỷ lệ lớn hơn về tổng số lập trình viên.

Tỷ lệ lập trình viên theo quy mô nhóm (ước lượng): | Quy mô nhóm | Tỷ lệ lập trình viên ước lượng | |----------|----------------------| | 1–3 | 5% | | 4–10 | 10% | | 11–25 | 15% | | 26–50 | 20% | | 50+ | 50% |

27.3 Ảnh hưởng của Quy mô Dự án đến Lỗi

> Tham khảo chi tiết thêm về lỗi ở Mục 22.4, "Typical Errors".

Cả về **số lượng** lẫn **loại lỗi** đều chịu ảnh hưởng bởi quy mô dự án. Khi quy mô lớn lên, tỷ lệ lỗi liên quan đến giai đoạn yêu cầu (requirements) và thiết kế (design) cũng tăng lên đáng kể (Xem Hình 27-2). Ở dự án nhỏ, lỗi trong quá trình xây dựng (construction) chiếm khoảng 75% tổng số lỗi. Lúc này, chất lượng phương pháp (methodology) ít ảnh hưởng đến chất lượng mã, trong khi kỹ năng cá nhân lập trình viên thường là yếu tố quyết định.

Dữ liệu thực tế:
Ở các dự án lớn hơn, lỗi xây dựng có thể giảm xuống còn 50% tổng số lỗi; phần còn lại chủ yếu đến từ lỗi yêu cầu và kiến trúc (architecture). Việc này gắn liền với việc dự án lớn phải đầu tư nhiều vào phát triển yêu cầu và thiết kế kiến trúc – tạo thêm cơ hội cho lỗi xuất hiện ở các khâu này. Tuy nhiên, trong một số dự án rất lớn (ví dụ 500.000 dòng mã), tỷ lệ lỗi xây dựng vẫn có thể chiếm đến 75%.

Khi quy mô tăng lên:
Số lỗi không chỉ tăng tuyến tính. Mật độ lỗi – tức số lỗi trên mỗi 1.000 dòng mã (KLOC – Kilo Lines of Code) – cũng tăng lên:

Quy mô dự án (dòng mã) Mật độ lỗi điển hình (lỗi/KLOC)

Quy mô dự án (dòng mã)	Mật độ lỗi điển hình (lỗi/KLOC)
< 2K	0–25
2K–16K	0–40
16K–64K	0.5–50
64K–512K	2–70
> 512K	4–100

Ý chính: Số lỗi sẽ tăng vượt tỷ lệ tuyến tính, khiến các dự án lớn phức tạp hơn đáng kể về mặt kiểm soát chất lượng.

27.4 Ảnh hưởng của Quy mô đến Năng suất

Năng suất lập trình cũng giống như chất lượng phần mềm, đều chịu tác động mạnh bởi quy mô dự án. Ở quy mô nhỏ (dưới 2.000 dòng mã), kỹ năng cá nhân lập trình viên là yếu tố lớn nhất ảnh hưởng đến năng suất. Khi quy mô tăng, yếu tố tổ chức nhóm và cấu trúc dự án trở nên quan trọng hơn.

Dữ liệu thực tế:
Một số nghiên cứu cho thấy: nhóm nhỏ có năng suất cao hơn 39% so với nhóm lớn, dù chỉ khác nhau một thành viên.

Quy mô dự án (dòng mã)	Dòng mã/năm/người (Cocomo II Nominal trong ngoặc)
1K	2.500–25.000 (4.000)
10K	2.000–25.000 (3.200)
100K	1.000–20.000 (2.600)
1.000K	700–10.000 (2.000)
10.000K	300–5.000 (1.600)

Giải thích:
Năng suất bị ảnh hưởng mạnh bởi loại phần mềm, chất lượng nhân sự, ngôn ngữ lập trình, phương pháp phát triển, độ phức tạp, môi trường lập trình, công cụ hỗ trợ, và cách thống kê dòng mã. Tuy nhiên, xu hướng chung là: năng suất ở dự án nhỏ có thể gấp 2–3 lần các dự án lớn; và năng suất giữa nhỏ và lớn có thể chênh lệch đến 5–10 lần.

27.5 Ảnh hưởng của Quy mô đến Các Hoạt động Phát triển

Với dự án một người, thành bại chủ yếu do chính bạn quyết định. Ở dự án 25 người, ảnh hưởng cá nhân giảm dần; yếu tố tổ chức tổng thể sẽ chi phối thành công hoặc thất bại nhiều hơn.

Tỷ lệ hoạt động theo quy mô

Khi quy mô tăng, nhu cầu giao tiếp chính quy tăng lên, kéo theo các hoạt động khác tăng "phi tuyến". Hình 27-3 minh họa tỷ lệ các hoạt động phát triển ở các quy mô khác nhau.

- Dự án nhỏ: xây dựng (construction) chiếm đến 65% tổng thời gian.
- Dự án vừa: xây dựng chiếm khoảng 50%.

- Dự án lớn: thời gian dành cho kiến trúc (architecture), tích hợp (integration), kiểm thử hệ thống (system testing) tăng lên đáng kể, trong khi xây dựng mất dần ưu thế.

Ý chính: Khi quy mô tăng, xây dựng chỉ chiếm một phần nhỏ hơn trong tổng nỗ lực dự án.

Hình 27-4 cho thấy:
Công việc xây dựng (construction) tăng gần như tuyến tính theo quy mô, trong khi các hoạt động khác lại tăng phi tuyến, nhanh hơn rất nhiều.

Khi hai dự án có quy mô tương đồng, hoạt động phát triển tương tự nhau. Khi quy mô khác biệt lớn (ví dụ từ Gigatron gốc lên Gigatron Deluxe), nỗ lực không chỉ tăng tuyến tính:

- Cần 25 lần xây dựng,
- 25–50 lần lập kế hoạch,
- 30 lần tích hợp,
- 40 lần kiến trúc và kiểm thử hệ thống.

Danh sách các hoạt động tăng trưởng phi tuyến khi quy mô tăng:

- Giao tiếp (Communication)
- Lập kế hoạch (Planning)
- Quản lý (Management)
- Phát triển yêu cầu (Requirements development)
- Thiết kế chức năng hệ thống (System functional design)
- Thiết kế và đặc tả giao diện (Interface design and specification)
- Kiến trúc (Architecture)
- Tích hợp (Integration)
- Loại bỏ lỗi (Defect removal)
- Kiểm thử hệ thống (System testing)
- Sản xuất tài liệu (Document production)

Lưu ý:
Dù dự án thuộc bất kỳ quy mô nào, một số thực hành luôn giá trị: kỷ luật mã hóa, kiểm tra thiết kế và mã bởi người khác, hỗ trợ công cụ tốt và sử dụng ngôn ngữ bậc cao. Các thực hành này càng trở nên không thể thiếu ở dự án lớn.

Các cấp độ: Chương trình, Sản phẩm, Hệ thống, và Sản phẩm hệ thống

Dòng mã và quy mô nhóm chỉ là hai trong nhiều yếu tố ảnh hưởng đến quy mô dự án. Một yếu tố tinh tế hơn là chất lượng và độ phức tạp của sản phẩm phần mềm cuối cùng.

Ví dụ, bản Gigatron Jr chỉ gồm 2.500 dòng mã và một người viết có thể xong trong một tháng. Nhưng bản hoàn chỉnh Gigatron với 25.000 dòng mã lại cần tới 20 tháng.

Định nghĩa:

- “Chương trình” (program): Phần mềm đơn lẻ, do một người phát triển và sử dụng, hoặc chỉ chia sẻ cho vài người khác.
- “Sản phẩm phần mềm” (software product): Được thiết kế cho người dùng ngoài người phát triển, được kiểm thử kỹ, tài liệu đầy đủ, và có khả năng bảo trì bởi người khác. Chi phí phát

triển thường gấp 3 lần một chương trình cùng loại.
- "Hệ thống phần mềm" (software system): Một nhóm chương trình phối hợp với nhau – phát triển phức tạp do khâu tích hợp và quản lý giao diện, tốn kém gấp 3 lần một chương trình đơn lẻ.
- "Sản phẩm hệ thống" (system product): Kết hợp yêu cầu chất lượng của một sản phẩm, và độ phức tạp của một hệ thống – tốn gấp 9 lần phát triển một chương trình đơn lẻ.

Dữ liệu thực tế:
Việc xem nhẹ sự khác biệt giữa chương trình, sản phẩm, hệ thống và sản phẩm hệ thống thường dẫn đến lỗi ước lượng nghiêm trọng. Lập trình viên lấy kinh nghiệm viết chương trình để dự đoán lịch trình xây dựng sản phẩm hệ thống có thể ước lượng thấp hơn thực tế đến gần 10 lần.

- Nếu dựa chỉ trên kinh nghiệm xây dựng 2K dòng mã để ước lượng thời gian phát triển một chương trình 2K dòng mã, bạn mới chỉ tính được 65% tổng thời gian thực cần thiết (chưa kể các hoạt động ngoài xây dựng).
- Khi dự án mở rộng, thời gian xây dựng chỉ là một phần nhỏ, ước lượng dựa trên kinh nghiệm xây dựng càng sai lệch.
- Nếu mở rộng lên 32K dòng mã mà vẫn dùng kinh nghiệm 2K để ước tính, con số này chỉ che phủ 50% tổng thời gian cần thiết; thực tế sẽ kéo dài gấp đôi.

Phương pháp luận và Quy mô

Phương pháp luận (methodology) tồn tại ở mọi loại dự án. Ở dự án nhỏ, phương pháp luận thường cảm tính, linh hoạt. Ở dự án lớn, cần chính quy, có kế hoạch kỹ lưỡng. Đôi khi, có lập trình viên cho rằng mình không tuân theo bất kỳ quy trình nào, nhưng thực tế, mọi cách tiếp cận lập trình đều là một dạng phương pháp luận – dù tự giác hay vô thức.

Áp dụng quá máy móc các phương pháp chính quy vào dự án nhỏ sẽ tạo ra gánh nặng không cần thiết. Tuy vậy, độ phức tạp của dự án lớn đòi hỏi sự chú ý có ý thức đến phương pháp luận. Giống như xây nhà tầng và xây chọc trời – quy mô khác nhau, cách tiếp cận cũng phải khác.

Ý chính: Người quản lý dự án thành công phải lựa chọn chiến lược thích hợp khi phát triển các dự án lớn.

> Trong xã hội, càng trang trọng thì "trang phục" của bạn càng phải chỉn chu, ví dụ giày cao gót, cà vạt, v.v. – tương tự như sự chính quy trong quản lý dự án lớn.

Lưu ý kiểm tra bản dịch:

- Các đoạn code đều được giữ nguyên (không xuất hiện trong đoạn này).
- Các thuật ngữ như "API", "system testing", "integration", "product", "program", v.v. luôn được giữ nguyên và chú thích khi lần đầu xuất hiện.
- Đã sửa lỗi định dạng, lược bỏ lỗi đánh máy (nếu có), tăng độ mạch lạc và ngắt đoạn đúng ý học thuật.
- Các bảng dữ liệu và hình minh họa đều được giải thích và chuyển sang dạng markdown cho phù hợp việc đọc hiểu.

Bản dịch hoàn chỉnh, đảm bảo đầy đủ ý nghĩa, ngữ cảnh khoa học máy tính và tính học thuật, trang trọng.

Kích thước Dự án và Tác động đến Quản lý Phát triển Phần mềm

Capers Jones chỉ ra rằng, một dự án có 1.000 dòng mã nguồn (lines of code) sẽ sử dụng trung bình khoảng 7% nỗ lực cho công tác giấy tờ (paperwork), trong khi một dự án có 100.000 dòng mã nguồn sẽ dành trung bình khoảng 26% nỗ lực cho công tác này (Jones 1998).

> Công tác giấy tờ này không được tạo ra chỉ vì niềm vui khi viết tài liệu, mà là kết quả trực tiếp của hiện tượng được minh họa trong Hình 27-1: càng có nhiều bộ óc cần phối hợp, càng cần nhiều tài liệu chính thức để điều phối công việc.
> Việc xây dựng các tài liệu này không nhằm mục đích bản thân nó, mà nhằm buộc bạn phải suy nghĩ kỹ lưỡng về các khía cạnh như quản lý cấu hình (configuration management) và giải thích kế hoạch cho tất cả mọi người khác trong dự án. Tài liệu là hệ quả hữu hình của quá trình lập kế hoạch và xây dựng hệ thống phần mềm. Nếu bạn cảm thấy mình chỉ đang làm theo hình thức và viết các tài liệu mang tính chung chung, điều đó cho thấy có điều gì đó không ổn.

Ảnh hưởng của Phương pháp luận tới Quy mô Dự án

> "Nhiều hơn" không đồng nghĩa với "tốt hơn" khi xét đến phương pháp luận (methodologies). Trong đánh giá về các phương pháp luận Agile (linh hoạt) và plan-driven (hướng kế hoạch), Barry Boehm và Richard Turner cảnh báo rằng bạn sẽ thường thành công hơn nếu bắt đầu với phương pháp nhỏ và mở rộng cho dự án lớn, thay vì sử dụng một phương pháp bao quát rồi cố gắng giảm nhẹ nó cho dự án nhỏ (Boehm và Turner 2004). Một số chuyên gia phân biệt giữa các phương pháp “lightweight” (nhẹ) và “heavyweight” (nặng), nhưng trên thực tế, điều cốt lõi là phải xem xét kích thước và loại hình cụ thể của dự án để lựa chọn phương pháp phù hợp, hay còn gọi là “right-weight”.

Tài Liệu Tham Khảo Thêm

- Boehm, Barry và Richard Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston, MA: Addison-Wesley, 2004. Trình bày tác động của quy mô dự án lên việc áp dụng phương pháp Agile và plan-driven, cùng các vấn đề liên quan.
- Cockburn, Alistair. *Agile Software Development*. Boston, MA: Addison-Wesley, 2002. Chương 4 bàn về việc chọn phương pháp phù hợp với kích thước dự án. Chương 6 giới thiệu Crystal Methodologies.
- Boehm, Barry W. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981. Đề cập sâu rộng tới tác động chi phí, năng suất và chất lượng do kích thước dự án cũng như các biến số khác.
- Jones, Capers. *Estimating Software Costs*. New York, NY: McGraw-Hill, 1998. Sách này chứa nhiều bảng biểu và đồ thị phân tích các nguồn năng suất phát triển phần mềm. Quyển *Programming Productivity* (1986) cũng có bàn luận liên quan.
- Brooks, Frederick P., Jr. *The Mythical Man-Month: Essays on Software Engineering*, Phiên bản kỷ niệm (2nd ed.). Reading, MA: Addison-Wesley, 1995. Tác giả thảo luận về các vấn đề quản lý đội nhỏ và lớn cũng như mô hình nhóm trưởng.
- DeGrace, Peter và Leslie Stahl. *Wicked Problems, Righteous Solutions: A Catalogue of Modern Software Engineering Paradigms*. Englewood Cliffs, NJ:

Yourdon Press, 1990. Sách cung cấp cách tiếp cận phát triển phần mềm dựa trên quy mô và hình thức dự án.
- Jones, T. Capers. “Program Quality and Programmer Productivity.” IBM Technical Report TR 02 764 (January 1977): 42–78. Phân tích chi tiết về lý do tại sao dự án lớn có cách tiêu tiền khác với dự án nhỏ,...

Các Ý chính

- **Khi quy mô dự án tăng lên**, nhu cầu liên lạc (communication) cần phải được hỗ trợ. Mục đích chính của đa số các phương pháp luận là giảm thiểu các vấn đề truyền đạt, và một phương pháp luận thực sự nên được đánh giá dựa trên khả năng hỗ trợ giao tiếp.
- **Năng suất (productivity) của dự án lớn**, trong điều kiện các yếu tố khác không đổi, sẽ thấp hơn dự án nhỏ.
- **Dự án phần mềm lớn** sẽ phát sinh nhiều lỗi hơn trên mỗi nghìn dòng mã nguồn (KLOC) so với dự án nhỏ, nếu các điều kiện khác không thay đổi.
- **Các hoạt động được mặc định là đơn giản với dự án nhỏ** cần phải được lập kế hoạch cẩn thận trong dự án lớn. Vai trò của hoạt động xây dựng mã sẽ giảm dần khi kích thước dự án tăng lên.
- **Mở rộng phương pháp “lightweight”** phù hợp lên dự án lớn thường hiệu quả hơn là thu hẹp phương pháp “heavyweight” cho dự án nhỏ. Lựa chọn lý tưởng là sử dụng phương pháp “right-weight” phù hợp với hoàn cảnh thực tế.

Chương 28: Quản lý Xây dựng Phần mềm

Các chủ đề liên quan:

- Điều kiện tiên quyết của xây dựng phần mềm: Chương 3.
- Xác định loại phần mềm: Mục 3.2.
- Kích thước chương trình: Chương 27.
- Chất lượng phần mềm: Chương 20.

Quản lý phát triển phần mềm là một thách thức lớn trong nhiều thập kỷ vừa qua. Như Hình 28-1 minh họa, chủ đề quản lý dự án phần mềm nằm ngoài phạm vi cuốn sách này, tuy nhiên chương này sẽ bàn đến một số chủ đề quản lý liên quan trực tiếp tới giai đoạn xây dựng mã nguồn (construction). Nếu bạn là nhà phát triển, chương này sẽ giúp bạn hiểu các vấn đề mà nhà quản lý thường quan tâm. Nếu bạn là nhà quản lý, chương này sẽ giúp bạn nhìn nhận quản lý từ góc độ của lập trình viên cũng như cung cấp các phương pháp quản lý giai đoạn xây dựng một cách hiệu quả. Do chương này đề cập nhiều chủ đề, nên trong từng mục có gợi ý nguồn tham khảo bổ sung.

28.1 Thúc đẩy Lập trình Tốt

Vì mã nguồn là sản phẩm chính của giai đoạn xây dựng, một câu hỏi lớn đặt ra là: *Làm thế nào để khuyến khích các thực hành lập trình tốt?* Nhìn chung, việc yêu cầu một bộ tiêu chuẩn kỹ thuật nghiêm ngặt từ phía quản lý không phải là ý tưởng hay. Các lập trình viên thường xem các nhà quản lý như là “chưa tiến hóa về mặt kỹ thuật”, và nếu phải tuân thủ các tiêu chuẩn lập trình, chính các lập trình viên nên là người quyết định chúng.

Nếu một ai đó sẽ xác định tiêu chuẩn, hãy để một kiến trúc sư (architect) được tôn trọng đảm nhận, thay vì quản lý. Các dự án phần mềm thường vận hành dựa trên "thứ bậc chuyên môn" thay vì thứ bậc quyền lực. Nếu kiến trúc sư thực sự được coi là người dẫn dắt về chuyên môn, đội dự án sẽ sẵn sàng tuân thủ các tiêu chuẩn do người đó đề ra.

Tuy nhiên, cần đảm bảo người kiến trúc sư này thực sự được tôn trọng và có kinh nghiệm thực tế, nếu không, lập trình viên sẽ phản đối việc bị áp đặt các tiêu chuẩn thiếu phù hợp với thực tế sản xuất mã nguồn.

Xem xét khi Đặt ra Tiêu chuẩn

- Tiêu chuẩn có ích ở một số tổ chức nhưng không có nghĩa lý ở những nơi khác.
- Nhiều nhà phát triển thích tiêu chuẩn vì giúp giảm sự tùy tiện trong dự án.
- Nếu nhóm của bạn không muốn tuân theo tiêu chuẩn nghiêm ngặt, hãy cân nhắc các giải pháp thay thế như: hướng dẫn linh hoạt, tập hợp các gợi ý, hoặc các ví dụ thực tế tiêu biểu cho các thực hành tốt nhất.

Các Kỹ thuật Khuyến khích Lập trình Tốt

Một số kỹ thuật hữu ích hơn là áp đặt tiêu chuẩn cứng nhắc:

- **Giao mỗi phần dự án cho hai người (pair programming, mentor-trainee, buddy review):** Nếu mỗi dòng mã phải do hai người kiểm tra, chất lượng và khả năng đọc sẽ được đảm bảo.
- **Xem xét từng dòng mã (code review):** Việc duyệt mã thường cần tối thiểu ba người tham gia trên mỗi dòng, đảm bảo nhiều cặp mắt kiểm tra lỗi và nâng cao chất lượng. Qua thời gian, các lần duyệt sẽ hình thành nên tiêu chuẩn nhóm.
- **Yêu cầu "ký duyệt mã" (code sign-off):** Như kỹ thuật phê duyệt bản vẽ kỹ thuật, một số công ty yêu cầu lập trình viên cấp cao ký xác nhận mã trước khi hoàn thành.
- **Giao lưu mã nguồn chất lượng:** Gửi các ví dụ mã tốt trong nhóm hoặc công khai lên bảng tin, giúp mọi thành viên hiểu rõ chuẩn mực mà tổ chức mong muốn. Việc này dễ dàng hơn biên soạn một cuốn tiêu chuẩn đầy đủ bằng lời.
- **Nhấn mạnh mã nguồn là tài sản chung (public asset):** Mã là kết quả của công sức lập trình viên, nhưng nó thuộc về dự án chung. Mọi thành viên đều nên xem, duy trì, kiểm tra mã này khi cần thiết – đây là tinh thần của mô hình Open Source Software và Extreme Programming (sở hữu tập thể).
- **Khen thưởng mã tốt:** Chỉ nên khen thưởng cho mã thực sự nổi bật và lập trình viên nhận được phần thưởng phải xứng đáng về mặt kỹ thuật, tránh "bình công" không hợp lý làm mất uy tín của người quản lý.

  Một tiêu chuẩn dễ áp dụng: Nếu bạn là quản lý có nền tảng lập trình, có thể đặt ra tiêu chí "Tôi phải đọc và hiểu được bất kỳ đoạn mã nào viết cho dự án." Điều này khuyến khích mã rõ ràng, dễ hiểu và tránh mã quá "khéo léo" hoặc phức tạp.

Vai trò của Sách này

Phần lớn cuốn sách này tập trung thảo luận các thực hành lập trình tốt, không nhằm mục đích thúc ép tiêu chuẩn cứng nhắc mà là làm cơ sở thảo luận, tra cứu và áp dụng các thực hành có lợi cho từng môi trường cụ thể.

28.2 Quản lý Cấu hình (Configuration Management)

Dự án phần mềm luôn luôn biến động: mã nguồn thay đổi, thiết kế thay đổi, yêu cầu thay đổi. Thêm nữa, thay đổi ở yêu cầu kéo theo đổi thiết kế, đổi thiết kế dẫn đến điều chỉnh mã và các ca kiểm thử.

Cấu hình là gì?

Quản lý cấu hình (configuration management, đôi khi gọi là change control) là thực hành xác định các sản phẩm của dự án và kiểm soát thay đổi một cách có hệ thống nhằm duy trì tính toàn vẹn của hệ thống theo thời gian. Nó bao gồm kỹ thuật đánh giá, theo dõi, và lưu các bản sao hệ thống tại các thời điểm khác nhau.

Nếu bạn không kiểm soát các thay đổi về yêu cầu, bạn có thể viết mã cho những phần sẽ bị loại bỏ sau này, hoặc sản sinh mã không tương thích với các phần mới. Các lỗi không phát hiện cho tới giai đoạn tích hợp gây ra việc đổ lỗi lẫn nhau, bởi không ai biết thực chất có gì thay đổi.

Nếu thay đổi mã không được kiểm soát, bạn có thể chỉnh sửa một routine cùng lúc với người khác, dẫn tới kết hợp thay đổi thất bại; phiên bản mã được kiểm thử có thể là phiên bản cũ, không phải phiên bản vừa sửa; lỗi mới khó quay lại bản ổn định trước đó v.v.

> Không có kiểm soát thay đổi, bạn chỉ như đi lung tung trong sương mù, không thể tiến tới mục tiêu rõ ràng. Quản lý cấu hình giúp tận dụng thời gian hiệu quả, thay vì lãng phí vào những vòng lặp vô bổ.

Nhiều lập trình viên vẫn tránh quản lý cấu hình trong nhiều năm. Một khảo sát hơn 20 năm trước cho thấy trên 1/3 lập trình viên còn chưa biết tới khái niệm này (Beck và Perkins 1983), và thực tế chưa cải thiện nhiều. Một nghiên cứu của Software Engineering Institute gần đây chỉ ra rằng ít hơn 20% các tổ chức sử dụng thực hành phát triển phi chính thức có quản lý cấu hình đầy đủ (SEI 2003).

> Quản lý cấu hình *không* phải do lập trình viên phát minh, nhưng bởi các dự án phần mềm biến động mạnh, nó trở nên đặc biệt quan trọng. Khi áp dụng vào phần mềm, nó thường được gọi là **Software Configuration Management (SCM)**, tập trung vào yêu cầu, mã nguồn, tài liệu và dữ liệu kiểm thử.

Một Vấn đề lớn của SCM: Quản lý quá mức (overcontrol)

Nếu kiểm soát quá mức, sự phát triển phần mềm sẽ bị "bóp nghẹt", thậm chí triệt tiêu hoàn toàn. Do đó, cần lên kế hoạch quản lý cấu hình vừa phải, biến nó thành trợ lực thay vì gánh nặng.

Quy mô dự án ảnh hưởng đến cách tổ chức SCM

- **Dự án nhỏ, một người:** Thực ra có thể chỉ cần backup định kỳ không chính thức; nhưng dù vậy, quản lý cấu hình vẫn hữu ích.
- **Dự án lớn, nhiều người:** Cần áp dụng các quy trình quản lý cấu hình chặt chẽ, kiểm soát việc sao lưu, kiểm soát thay đổi yêu cầu, thiết kế, kiểm soát tài liệu, mã nguồn, nội dung, các ca kiểm thử và các sản phẩm/deliverable khác.
- **Dự án vừa:** Độ chính thức vừa phải, theo thỏa thuận giữa các bên.

Thay đổi Yêu cầu & Thiết kế

Trong quá trình phát triển, bạn sẽ nảy sinh nhiều ý tưởng cải tiến, nhưng nếu thực hiện mỗi thay đổi ngay lập tức, bạn sẽ rơi vào "băng chuyền" phát triển: hệ thống vẫn thay đổi nhưng không

tiến gần tới hoàn thiện hơn.

Một số hướng dẫn để kiểm soát thay đổi thiết kế:

1. **Áp dụng quy trình kiểm soát thay đổi có hệ thống:** Đặt ra một quy trình rõ ràng để mọi thay đổi được cân nhắc dựa trên lợi ích cho dự án nói chung.
2. **Xử lý yêu cầu thay đổi thành từng nhóm:** Tránh thực hiện từng thay đổi nhỏ lẻ ngay khi có ý tưởng; nên thống nhất và xử lý các thay đổi hợp lý hơn khi có đầy đủ thông tin và đánh giá tổng thể.

Lưu ý: Bạn chỉ yêu cầu dịch đến đây, nếu cần dịch tiếp, vui lòng cung cấp tiếp phần văn bản.

Khi Dự Án Gần Hết Thời Gian

Khi bạn bắt đầu cạn thời gian vào cuối dự án, việc thay đổi thứ hai có chất lượng gấp 10 lần thay đổi thứ nhất cũng trở nên vô nghĩa—bạn sẽ không còn điều kiện để thực hiện các thay đổi không thiết yếu. Một số thay đổi tốt nhất có thể bị bỏ lỡ chỉ vì bạn nghĩ ra chúng muộn hơn thay vì sớm hơn.

Ghi Chép và Đánh Giá Các Ý Tưởng Thay Đổi

Một giải pháp cho vấn đề này là ghi lại tất cả các ý tưởng và đề xuất, dù việc triển khai chúng có dễ đến đâu, và lưu lại cho đến khi bạn có thời gian thực hiện. Sau đó, xem xét các thay đổi này theo nhóm và chọn ra những thay đổi mang lại lợi ích lớn nhất.

Hãy ước tính chi phí cho mỗi thay đổi. Bất cứ khi nào khách hàng, cấp trên, hoặc chính bạn muốn thay đổi hệ thống, hãy ước lượng thời gian cần thiết để thực hiện thay đổi đó, bao gồm cả việc rà soát mã lệnh (code), kiểm thử lại toàn bộ hệ thống. Bao gồm cả thời gian xử lý hiệu ứng lan tỏa của thay đổi từ yêu cầu tới thiết kế, từ thiết kế đến mã nguồn (code), từ code đến kiểm thử (test), cũng như việc cập nhật tài liệu cho người dùng. Hãy để tất cả các bên liên quan hiểu rằng phần mềm được liên kết rất chặt chẽ, và việc ước tính thời gian là cần thiết ngay cả khi thay đổi ban đầu có vẻ nhỏ.

Dù bạn lạc quan thế nào khi thay đổi được đề xuất, hãy tránh việc đưa ra ước tính cảm tính. Những ước tính vội vàng này thường sai lệch ít nhất 2 lần hoặc hơn.

> **Tham khảo chéo:** Đối với vấn đề số lượng lớn yêu cầu thay đổi, điều này là dấu hiệu cảnh báo rằng yêu cầu, kiến trúc (architecture) hoặc thiết kế cấp cao chưa được thực hiện đủ tốt để hỗ trợ quá trình xây dựng hiệu quả. Việc quay lại hoàn thiện yêu cầu hoặc kiến trúc có thể tốn kém, nhưng sẽ không tốn kém bằng việc xây dựng lại phần mềm nhiều lần hoặc phải loại bỏ những phần không cần thiết. Để biết thêm về việc xử lý thay đổi yêu cầu trong quá trình xây dựng, xem phần "Handling Requirements Changes During Construction" tại Mục 3.4. Đối với cách xử lý thay đổi mã nguồn một cách an toàn, tham khảo Chương 24, “Refactoring”.

28.2 Quản Lý Cấu Hình (Configuration Management)

Thành Lập Hội Đồng Kiểm Soát Thay Đổi

Hãy thiết lập một hội đồng kiểm soát thay đổi (change-control board) hoặc tổ chức tương đương phù hợp với dự án của bạn. Nhiệm vụ của hội đồng là lọc ra những thay đổi thực sự cần thiết từ các đề xuất thay đổi không quan trọng. Bất cứ ai muốn đề xuất thay đổi sẽ nộp yêu cầu thay đổi (change request) cho hội đồng. "Yêu cầu thay đổi" là bất kỳ đề xuất nào làm thay đổi phần mềm: ý tưởng về tính năng mới, thay đổi với tính năng hiện tại, báo cáo lỗi (error report) dù có thể đúng hay không.

Hội đồng sẽ họp định kỳ để xem xét các đề xuất thay đổi, và sẽ phê duyệt, từ chối, hoặc trì hoãn từng thay đổi. Hội đồng kiểm soát thay đổi được xem là thực hành tốt nhất để ưu tiên và kiểm soát các thay đổi về yêu cầu; tuy nhiên, chúng vẫn còn khá hiếm trong môi trường thương mại (Jones 1998, 2000).

Phòng Tránh Quan Liêu Nhưng Không Thiếu Kiểm Soát

Cần đề phòng chủ nghĩa quan liêu, nhưng đừng để nỗi sợ quan liêu ngăn cản việc kiểm soát thay đổi hiệu quả. Thiếu kiểm soát thay đổi kỷ luật là một trong những vấn đề quản lý nghiêm trọng nhất trong ngành phần mềm hiện nay. Một tỷ lệ lớn các dự án bị coi là trễ thực chất là đúng tiến độ nếu tính đến tác động của các thay đổi không được theo dõi nhưng đã được chấp thuận.

Kiểm soát thay đổi kém cho phép các thay đổi tích tụ mà không ghi nhận chính thức, làm suy yếu khả năng theo dõi, dự báo dài hạn, lập kế hoạch dự án, quản lý rủi ro và điều hành dự án nói chung.

Quy trình kiểm soát thay đổi dễ trở nên quan liêu, do đó, cần tìm cách tối giản quy trình. Nếu bạn không muốn sử dụng các yêu cầu thay đổi truyền thống, hãy tạo một địa chỉ email “ChangeBoard” và yêu cầu mọi người gửi đề xuất thay đổi đến địa chỉ này. Hoặc, cho phép nêu ý kiến thay đổi trực tiếp tại các cuộc họp của hội đồng. Một phương pháp mạnh mẽ là ghi nhận các đề xuất thay đổi như là các lỗi (defect) trong phần mềm theo dõi lỗi (defect-tracking software). Bạn có thể phân loại các thay đổi này là “requirements defects” hoặc chỉ là thay đổi.

Bạn có thể xây dựng Change-Control Board một cách chính thức, hoặc định nghĩa một Nhóm Hoạch Định Sản Phẩm (Product Planning Group) hay Hội đồng Chiến tranh (War Council) thực hiện vai trò tương tự, hoặc chỉ định một cá nhân chịu trách nhiệm—dù là tên gọi gì, nhất định phải thực hiện chức năng này.

> **KEY POINT:** Nội dung thực chất của kiểm soát thay đổi là quan trọng, đừng để nỗi sợ quan liêu ngăn cản bạn hưởng lợi từ nó.

Tôi thỉnh thoảng thấy các dự án gặp khó khăn vì thực hiện kiểm soát thay đổi quá máy móc, nhưng thường xuyên hơn nhiều là các dự án không có bất kỳ sự kiểm soát thay đổi hiệu quả nào cả.

Thay Đổi Mã Nguồn Phần Mềm

Một vấn đề khác trong quản lý cấu hình là kiểm soát mã nguồn (source code). Nếu bạn thay đổi mã và xuất hiện lỗi mới không liên quan tới thay đổi vừa thực hiện, bạn có thể muốn so sánh phiên bản mới với phiên bản cũ để tìm nguồn gốc lỗi. Nếu vẫn chưa rõ nguyên nhân, bạn có thể phải kiểm tra các phiên bản cũ hơn.

Hành trình này dễ dàng hơn rất nhiều nếu bạn có công cụ quản lý phiên bản (version-control tools) theo dõi nhiều phiên bản khác nhau của mã nguồn.

Phần mềm Quản Lý Phiên Bản (Version-control Software)

Phần mềm quản lý phiên bản tốt vận hành rất trơn tru đến mức bạn hầu như không nhận ra mình đang sử dụng nó, đặc biệt hữu ích khi làm việc nhóm. Một kiểu kiểm soát phiên bản khóa tệp nguồn để chỉ một người được chỉnh sửa tại một thời điểm. Khi bạn muốn làm việc với mã nguồn trong một tệp bất kỳ, bạn phải "check out" tệp đó. Nếu người khác đã "check out", bạn sẽ được báo và không thể lấy tệp đó. Nếu không, bạn chỉnh sửa như bình thường cho đến khi "check in". Một kiểu khác cho phép nhiều người cùng làm việc trên tệp và giải quyết xung đột khi nộp lại (check in).

Khi bạn check in, phần mềm quản lý phiên bản sẽ hỏi lý do thay đổi, và bạn cần cung cấp lý do.

Những lợi ích lớn đạt được từ khoản đầu tư nhỏ này:

- Bạn không “giẫm chân” lên công việc của người khác khi cùng chỉnh sửa một tệp, hoặc ít nhất bạn sẽ nhận được cảnh báo nếu điều này xảy ra.
- Bạn có thể dễ dàng cập nhật tập tin dự án của mình với phiên bản mới nhất chỉ bằng một lệnh.
- Bạn có thể quay lại bất kỳ phiên bản nào đã được lưu trữ.
- Bạn có thể có danh sách các thay đổi đã thực hiện đối với bất kỳ phiên bản nào.
- Bạn không phải lo lắng về sao lưu cá nhân, vì hệ thống quản lý phiên bản đóng vai trò như một mạng an toàn dự phòng.

Quản lý phiên bản là điều không thể thiếu khi làm việc nhóm. Giá trị của nó càng tăng lên khi quản lý phiên bản, theo dõi lỗi, và kiểm soát thay đổi được tích hợp với nhau. Bộ phận ứng dụng của Microsoft nhận thấy công cụ kiểm soát phiên bản độc quyền của họ là một “lợi thế cạnh tranh lớn” (Moore 1992).

Quản Lý Phiên Bản Công Cụ

Đối với một số dự án, có thể cần khả năng tái tạo chính xác môi trường dùng để tạo ra từng phiên bản phần mềm, bao gồm cả trình biên dịch (compiler), liên kết (linker), thư viện mã (code libraries), v.v. Trong trường hợp này, bạn cũng nên đặt toàn bộ các công cụ này vào hệ thống quản lý phiên bản.

Cấu Hình Máy Tính

Nhiều công ty (bao gồm cả công ty tôi) đã thu được kết quả tốt khi tạo các cấu hình tiêu chuẩn cho máy phát triển. Một ảnh đĩa của máy trạm lập trình viên chuẩn sẽ bao gồm tất cả công cụ lập trình phổ biến, ứng dụng văn phòng, v.v. Ảnh này được cài trên từng máy của lập trình viên. Nhờ vậy, có thể tránh được nhiều vấn đề phát sinh từ các cấu hình khác nhau, các phiên bản công cụ khác nhau, và đồng thời rút ngắn thời gian thiết lập máy mới hơn rất nhiều so với việc cài đặt thủ công từng phần mềm.

Kế Hoạch Sao Lưu (Backup Plan)

Kế hoạch sao lưu không phải là ý tưởng mới mẻ; đó chỉ đơn thuần là việc sao lưu công việc định kỳ. Nếu bạn đang viết một cuốn sách bằng tay, bạn sẽ không để các trang bản thảo ở ngoài hiên. Nếu làm vậy, chúng có thể bị mưa ướt, bị gió cuốn đi, hoặc bị chó nhà hàng xóm tha mất. Bạn sẽ cất chúng ở nơi an toàn. Phần mềm thì "vô hình" hơn nên dễ bị lãng quên, dù giá trị lớn tác phẩm của bạn đang nằm trên một chiếc máy tính.

Có rất nhiều rủi ro với dữ liệu máy tính: ổ đĩa có thể hỏng; bạn hoặc ai đó có thể vô tình xóa nhầm tệp quan trọng; nhân viên bất mãn có thể phá hoại; hoặc bạn có thể mất máy do trộm, cháy, hoặc lũ lụt. Hãy chủ động bảo vệ công việc. Kế hoạch sao lưu cần gồm các yếu tố:

- Sao lưu định kỳ tất cả tài liệu dự án
- Định kỳ chuyển bản sao lưu ra khỏi nơi làm việc (off-site storage)
- Sao lưu tất cả tài liệu quan trọng—bao gồm cả mã nguồn (source code), tài liệu, đồ họa, và ghi chú quan trọng

Một mặt thường bị bỏ quên là kiểm tra lại quy trình sao lưu khôi phục (backup-recovery). Hãy thử phục hồi dữ liệu để đảm bảo mọi thứ đều được lưu và có thể khôi phục bình thường.

Khi kết thúc dự án, hãy tạo bản lưu trữ (archive) cho dự án. Lưu lại bản sao của mọi thứ: mã nguồn, trình biên dịch, công cụ, yêu cầu, thiết kế, tài liệu—tất cả những gì cần để tái tạo sản phẩm. Giữ lại ở nơi an toàn.

Danh Sách Kiểm Tra: Quản Lý Cấu Hình

Tổng Quát

- ❑ Kế hoạch quản lý cấu hình phần mềm của bạn có nhằm hỗ trợ lập trình viên và giảm thiểu phí tổn không?
- ❑ Cách tiếp cận quản lý cấu hình phần mềm (SCM) có tránh kiểm soát quá mức dự án không?
- ❑ Bạn có nhóm các yêu cầu thay đổi, dù bằng các phương tiện phi chính thức (ví dụ: danh sách các thay đổi chờ xử lý) hoặc bằng phương pháp hệ thống (như hội đồng kiểm soát thay đổi)?
- ❑ Bạn có ước tính hệ thống chi phí, lịch trình và ảnh hưởng chất lượng của mỗi thay đổi đề xuất không?
- ❑ Bạn có coi các thay đổi lớn là dấu hiệu cho thấy quá trình phát triển yêu cầu chưa hoàn chỉnh?

Công Cụ

- ❑ Bạn có sử dụng phần mềm quản lý phiên bản để quản lý cấu hình không?
- ❑ Bạn có dùng phần mềm quản lý phiên bản nhằm giảm vấn đề phối hợp trong làm việc nhóm không?

Sao Lưu

- ❑ Bạn có sao lưu tất cả tài liệu dự án định kỳ không?
- ❑ Sao lưu dự án có được chuyển định kỳ ra nơi lưu trữ bên ngoài không?
- ❑ Mọi tài liệu, bao gồm mã nguồn, tài liệu, đồ họa và ghi chú quan trọng đều được sao lưu?
- ❑ Bạn đã kiểm thử quy trình phục hồi sau khi sao lưu?

Tài Nguyên Bổ Sung về Quản Lý Cấu Hình

Vì cuốn sách này tập trung vào xây dựng phần mềm, nên phần này chỉ đề cập đến kiểm soát thay đổi từ góc độ xây dựng. Tuy nhiên, thay đổi ảnh hưởng đến dự án ở tất cả các mức, vì vậy cần một chiến lược kiểm soát thay đổi toàn diện.

Tài liệu tham khảo:

- **Hass, Anne Mette Jonassen** – *Configuration Management Principles and Practices*, Addison-Wesley, 2003.
- **Berczuk, Stephen P. & Brad Appleton** – *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*, Addison-Wesley, 2003.
- **SPMN** – *Little Book of Configuration Management*, Software Program Managers Network, 1998. Tải bản miễn phí tại www.spmn.com/products_guidebooks.html
- **Bays, Michael** – *Software Release Methodology*, Prentice Hall, 1999.
- **Bersoff, Edward H. & Alan M. Davis** – "Impacts of Life Cycle Models on Software Configuration Management," *Communications of the ACM*, 34(8), 1991.

28.3 Ước Lượng Lịch Trình Xây Dựng Phần Mềm

Quản lý một dự án phần mềm là một trong những thách thức lớn của thế kỷ 21, và việc ước lượng kích thước và nỗ lực cần thiết để hoàn thành dự án là một trong những khía cạnh khó nhất của quản lý dự án phần mềm. Trung bình, các dự án phần mềm lớn đều bị trễ khoảng một năm và vượt ngân sách khoảng 100% (Standish Group 1994, Jones 1997, Johnson 1999). Ở cấp độ cá nhân, khảo sát về sự chênh lệch giữa ước lượng và thực tế cho thấy ước lượng của lập trình viên thường lạc quan 20–30% (van Genuchten 1991). Lý do không chỉ nằm ở quy trình phát triển kém, mà còn vì ước lượng về kích thước và công sức không chính xác.

Các Cách Tiếp Cận Ước Lượng

Bạn có thể ước lượng kích thước và công sức cần thiết để hoàn thành dự án bằng nhiều cách, ví dụ:

- Sử dụng phần mềm chuyên ước lượng
- Sử dụng phương pháp thuật toán như Cocomo II – mô hình ước lượng của Barry Boehm (Boehm et al., 2000)
- Mời chuyên gia bên ngoài ước lượng dự án
- Tổ chức họp rà soát (walk-through) để cùng nhau ước lượng
- Ước lượng từng phần rồi tổng hợp lại
- Cho các thành viên tự ước lượng công việc của mình rồi cộng lại
- Tham chiếu kinh nghiệm các dự án trước đó
- So sánh các ước lượng trước đó với kết quả thực tế để điều chỉnh cho ước lượng mới

Tham khảo thêm trong "Additional Resources on Software Estimation" ở cuối phần này.

Cách Tiếp Cận Ước Lượng Khuyến Nghị

1. **Xác định mục tiêu:** Tại sao bạn cần ước lượng? Bạn đang ước lượng những công đoạn nào? Chỉ riêng xây dựng, hay toàn bộ quá trình phát triển? Chỉ công việc dự án, hay cả thời gian nghỉ phép, ngày lễ, đào tạo,...?
2. **Cho phép đủ thời gian lập kế hoạch ước lượng:** Nếu bạn làm một dự án lớn, hãy xem ước lượng như một dự án nhỏ có kế hoạch riêng để thực hiện tốt.
3. **Làm rõ yêu cầu phần mềm:** Giống như kiến trúc sư không thể báo giá cho một căn nhà "khá to", bạn cũng không thể ước lượng dự án phần mềm "khá lớn" nếu chưa xác định rõ đặc tả. Hãy làm rõ yêu cầu hoặc lên kế hoạch nghiên cứu sơ bộ trước khi ước lượng.
4. **Ước lượng càng chi tiết càng tốt:** Càng kiểm tra chi tiết các hoạt động dự án, ước lượng càng sát thực tế. *Định luật số lớn* cho thấy các sai số cộng gộp ở nhiều phần nhỏ sẽ được triệt tiêu chứ không cộng dồn như khi ước lượng một khối lớn.
5. **Áp dụng nhiều kỹ thuật ước lượng và so sánh kết quả:** Sử dụng nhiều cách tiếp cận sẽ cho ra các kết quả khác nhau, đối chiếu và đánh giá sự chênh lệch có thể cho bạn thêm góc nhìn.
6. **Ước lượng lại định kỳ:** Các yếu tố của dự án sẽ thay đổi sau ước lượng ban đầu. Lập kế hoạch cập nhật lại ước lượng thường xuyên; so sánh kết quả thực tế với dự kiến và áp dụng vào những lần ước lượng tiếp theo.

   Hình dưới đây minh họa: ước lượng vào đầu dự án thường thiếu chính xác; càng tiến triển, ước lượng càng sát thực tế. Hãy cập nhật ước lượng trong suốt quá trình, tận dụng kinh nghiệm thực tế để làm mới dự báo cho phần còn lại.

Ước Lượng Khối Lượng Xây Dựng

Phạm vi xây dựng có ảnh hưởng lớn tới lịch trình dự án tùy vào tỷ trọng của công đoạn này (bao gồm thiết kế chi tiết, lập trình, kiểm thử đơn vị). Hãy giữ ghi chép kinh nghiệm các dự án của tổ chức để sử dụng dự đoán thời gian cho các dự án tiếp theo.

Các Yếu Tố Ảnh Hưởng Đến Lịch Trình

Ảnh hưởng lớn nhất tới lịch trình một dự án phần mềm là kích thước chương trình cần triển khai. Tuy nhiên, còn nhiều yếu tố khác cũng đóng vai trò quan trọng và thường đã được định lượng trong các nghiên cứu thương mại.

Bản dịch kết thúc tại đây. Nếu bạn cần tiếp tục phần bảng các yếu tố ảnh hưởng hoặc các phần tiếp theo, vui lòng cung cấp nội dung bổ sung.

Một số yếu tố khó định lượng ảnh hưởng đến tiến độ phát triển phần mềm

Các yếu tố dưới đây, được tổng hợp từ *Software Cost Estimation with Cocomo II* của Barry Boehm (2000) và *Estimating Software Costs* của Capers Jones (1998), là những nhân tố có thể tác động đáng kể đến tiến độ của một dự án phát triển phần mềm, dù chúng khó định lượng:

- Kinh nghiệm và năng lực của người phát triển yêu cầu (requirements developer)
- Kinh nghiệm và năng lực của lập trình viên (programmer)
- Động lực của nhóm thực hiện

- Chất lượng quản lý dự án
- Mức độ mã nguồn được tái sử dụng (code reuse)
- Tỷ lệ thay thế nhân sự (personnel turnover)
- Sự biến động của yêu cầu (requirements volatility)
- Chất lượng mối quan hệ với khách hàng
- Sự tham gia của người dùng vào quá trình xác định yêu cầu
- Kinh nghiệm của khách hàng đối với loại ứng dụng
- Mức độ lập trình viên tham gia vào quá trình phát triển yêu cầu
- Mức độ bảo mật được phân loại cho máy tính, chương trình, và dữ liệu liên quan
- Khối lượng tài liệu (documentation)
- Mục tiêu dự án (ưu tiên tiến độ so với chất lượng, khả năng sử dụng hay các mục tiêu khác)

Mỗi yếu tố kể trên đều có ảnh hưởng đáng kể, do vậy cần xem xét kết hợp các yếu tố này cùng với những yếu tố trong Bảng 28-1 (trong đó đã bao gồm một số yếu tố phía trên).

Ước lượng vs Kiểm soát

> Câu hỏi quan trọng là: bạn muốn **dự đoán** (prediction) hay **kiểm soát** (control)? — Tom Gilb

Ước lượng là một phần quan trọng trong kế hoạch nhằm đảm bảo hoàn thành dự án phần mềm đúng hạn. Khi ngày giao hàng và các thông số kỹ thuật đã cố định, vấn đề chủ yếu là làm sao kiểm soát sử dụng nguồn lực (con người, kỹ thuật) để đảm bảo sản phẩm được hoàn thành đúng thời hạn. Theo nghĩa đó, mức độ chính xác của ước lượng ban đầu ít quan trọng hơn so với việc kiểm soát nguồn lực trong quá trình thực hiện nhằm đáp ứng tiến độ.

Làm gì khi tiến độ bị trễ?

Các dự án phần mềm trung bình thường vượt tiến độ dự kiến khoảng 100%, như đã đề cập trước đó. Khi dự án bị trễ, việc gia hạn thời gian thường không phải là lựa chọn khả thi. Nếu có thể, hãy thực hiện, còn không, bạn có thể thử một hoặc nhiều giải pháp sau:

1. **Hy vọng sẽ bắt kịp tiến độ**
   Đây là phản ứng phổ biến đối với các dự án bị trễ: lạc quan hy vọng. Lập luận thường là: "Giai đoạn xác định yêu cầu mất nhiều thời gian hơn dự đoán, nhưng giờ phần yêu cầu đã vững chắc, nên chắc chắn sẽ tiết kiệm thời gian ở các giai đoạn sau." Tuy nhiên, khảo sát hơn 300 dự án phần mềm cho thấy, các chậm trễ và vượt tiến độ thường có xu hướng tăng về cuối dự án (van Genuchten 1991). Các dự án hiếm khi lấy lại được thời gian đã mất, mà thường còn trễ thêm.

2. **Mở rộng nhóm thực hiện**
   Theo định luật của Fred Brooks (“Brooks’s law”), việc thêm người vào một dự án phần mềm bị trễ chỉ làm dự án càng trễ hơn (Brooks 1995). Bởi những thành viên mới cần thời gian làm quen, đồng thời quá trình dẫn dắt họ sẽ tốn nguồn lực của những người đã quen việc. Việc tăng số lượng người cũng làm tăng độ phức tạp và chi phí cho giao tiếp nhóm. Brooks ví von rằng một phụ nữ có thể sinh con sau 9 tháng không có nghĩa là 9 phụ nữ có thể sinh ra một đứa trẻ chỉ trong một tháng.

   ***Ghi chú:* Cảnh báo từ định luật Brooks cần được cân nhắc hơn trong thực tế. Không nên cho rằng cứ tăng người sẽ đẩy nhanh tiến độ, vì phát triển phần mềm không giống như các công việc thủ công, nơi nhiều người làm việc đơn giản hóa được quy**

trình. Tuy nhiên, trong một số trường hợp, nếu nhiệm vụ có thể phân chia độc lập, việc thêm người vào có thể rút ngắn thời gian, nhưng điều này phải được xem xét cẩn thận (Abdel-Hamid 1989, McConnell 1999).

3. **Giảm phạm vi dự án**
   Việc giảm phạm vi (project scope) là một kỹ thuật mạnh mẽ nhưng thường bị bỏ qua. Việc loại bỏ một chức năng đồng nghĩa với loại bỏ sung song thiết kế, lập trình, kiểm thử và tài liệu liên quan đến chức năng đó, cũng như giảm khối lượng giao diện với các chức năng khác. Khi lập kế hoạch sản phẩm, hãy phân loại các chức năng thành "phải có" (must haves), "ưu tiên thêm" (nice to haves) và "tùy chọn" (optionals). Khi bị trễ tiến độ, hãy ưu tiên các chức năng "tùy chọn" và "ưu tiên thêm", và loại bỏ các chức năng ít quan trọng nhất.

   Nếu không thể loại bỏ chức năng, có thể triển khai một phiên bản đơn giản hơn, ví dụ, một phiên bản đảm bảo tiến độ nhưng chưa tối ưu về hiệu năng, hoặc đường tính năng phụ ít quan trọng có thể lập trình đơn giản hơn. Bạn cũng có thể giảm yêu cầu về tốc độ hoặc bộ nhớ nếu cần.

   Ngoài ra, hãy xem xét **ước lượng lại thời gian phát triển cho các chức năng kém quan trọng hơn.** Xác định bạn có thể hoàn thành những gì trong 2 giờ, 2 ngày hay 2 tuần, và lựa chọn giữa các phương án dựa vào lợi ích tương ứng.

Tài liệu tham khảo về Ước lượng phần mềm

- *Boehm, Barry, et al. Software Cost Estimation with Cocomo II.* Boston, MA: Addison-Wesley, 2000.
  Giới thiệu chi tiết về mô hình Cocomo II, mô hình ước lượng phổ biến nhất hiện nay.

- *Boehm, Barry W. Software Engineering Economics.* Englewood Cliffs, NJ: Prentice Hall, 1981.
  Cuốn sách này cung cấp một cái nhìn tổng quát hơn về ước lượng dự án phần mềm so với sách mới hơn của Boehm.

- *Humphrey, Watts S. A Discipline for Software Engineering.* Reading, MA: Addison-Wesley, 1995.
  Chương 5 trình bày phương pháp Probe của Humphrey dùng để ước lượng công việc ở mức cá nhân lập trình viên.

- *Conte, S. D., H. E. Dunsmore, and V. Y. Shen. Software Engineering Metrics and Models.* Menlo Park, CA: Benjamin/Cummings, 1986.
  Chương 6 khảo sát các kỹ thuật ước lượng, từ lịch sử, mô hình thống kê, lý thuyết đến mô hình tổng hợp, đồng thời so sánh với các dự án thực tế.

- *Gilb, Tom. Principles of Software Engineering Management.* Wokingham, England: Addison-Wesley, 1988.
  Chương 16 nhấn mạnh quan điểm nên tập trung vào kiểm soát dự án hơn là dự báo chính xác, đưa ra 10 nguyên tắc giúp đạt được các mục tiêu về thời hạn, chi phí, v.v.

Đo lường trong phát triển phần mềm

Dự án phần mềm có thể được đo lường bằng nhiều cách. Dưới đây là hai lý do thuyết phục để thực hiện đo lường quy trình:

1. **Với bất kỳ thuộc tính nào của dự án, việc đo lường luôn tốt hơn là không đo lường.** Mặc dù số liệu có thể chưa hoàn toàn chính xác, có thể khó thu thập và cần cải tiến theo thời gian, nhưng đo lường giúp bạn kiểm soát quy trình phát triển phần mềm tốt hơn (Gilb 2004).

2. **Nếu dữ liệu dùng trong thực nghiệm khoa học, nó phải được lượng hóa (quantified).** Ví dụ, một nhà khoa học không thể kiến nghị cấm một loại thức ăn mới chỉ vì "chuột trắng có vẻ ốm yếu hơn", mà phải đưa ra số liệu định lượng, như “Chuột ăn sản phẩm mới ốm 3,7 ngày/tháng nhiều hơn nhóm đối chứng”. Để đánh giá phương pháp phát triển phần mềm, bạn cũng phải đo lường.

Những phát biểu như “Phương pháp mới này có vẻ hiệu quả hơn” là không đủ.

Tác động và vấn đề của đo lường

> “Cái gì được đo lường thì sẽ được chú ý.”
> — Tom Peters

- Đo lường có tác động thúc đẩy tâm lý: mọi người chú ý tới những gì bị đo và thường tập trung vào phần việc đó, bỏ qua những việc không bị đo.
- Hãy lựa chọn yếu tố đo lường một cách cẩn trọng, vì nhân viên sẽ định hướng công việc dựa trên các chỉ số này.
- Phản đối đo lường tức cũng như từ chối biết thực trạng dự án. Dù chỉ đo lường một góc rất nhỏ nhưng vẫn còn hơn không đo lường gì (như có cửa sổ dù mờ cũng còn hơn không có cửa sổ).

Một số chỉ số đo lường hữu ích cho phát triển phần mềm

Bảng 28-2: Các chỉ số hữu ích trong phát triển phần mềm

Kích thước (Size)

- Tổng số dòng mã (lines of code)
- Tổng số dòng chú thích (comment lines)
- Tổng số lớp (class) hoặc hàm/thủ tục (routine)
- Tổng số khai báo dữ liệu
- Tổng số dòng trống (blank lines)

Chất lượng tổng thể (Overall Quality)

- Tổng số lỗi (defect)
- Số lượng lỗi trong từng lớp hoặc hàm
- Số lỗi trung bình trên 1.000 dòng mã
- Thời gian trung bình giữa các lỗi (mean time between failures)
- Số lỗi phát hiện qua biên dịch (compiler-detected errors)

Theo dõi lỗi (Defect Tracking)

- Mức độ nghiêm trọng của mỗi lỗi
- Vị trí (lớp/hàm) xuất hiện lỗi
- Nguyên nhân phát sinh lỗi (yêu cầu, thiết kế, lập trình, kiểm thử, v.v.)
- Cách thức khắc phục mỗi lỗi
- Người chịu trách nhiệm với mỗi lỗi
- Số dòng bị ảnh hưởng bởi việc sửa lỗi
- Số giờ làm việc khắc phục lỗi
- Thời gian trung bình để tìm/sửa lỗi
- Số lần thử khắc phục lỗi
- Số lỗi mới phát sinh do sửa lỗi

Khả năng bảo trì (Maintainability)

- Số hàm công khai trên mỗi lớp
- Số tham số truyền vào mỗi hàm
- Số hàm/biến riêng trên mỗi lớp
- Số biến cục bộ trên mỗi hàm
- Số hàm được gọi bởi mỗi lớp/hàm
- Số điểm quyết định (decision point) trong mỗi hàm
- Độ phức tạp luồng điều khiển (control-flow complexity)
- Số dòng mã, chú thích, khai báo dữ liệu, dòng trống, số lệnh goto, số lệnh nhập/xuất trong mỗi lớp/hàm

Năng suất (Productivity)

- Số giờ làm việc cho dự án/từng lớp/hàm
- Số lần mỗi lớp/hàm được chỉnh sửa
- Chi phí cho dự án, mỗi dòng mã, mỗi lỗi

Nhiều chỉ số kể trên có thể thu thập tự động bằng các công cụ phần mềm hiện nay. Tuy nhiên, tác giả khuyến cáo không nên ghi nhận tất cả mọi yếu tố ngay từ đầu, vì có thể bị ngập trong dữ liệu phức tạp. Hãy bắt đầu đo đếm đơn giản với số lỗi, số người/tháng, tổng chi phí và số dòng mã. Chuẩn hóa phương pháp đo trên toàn bộ dự án và mở rộng dần khi cần thiết (Pietrasanta 1990).

Hãy thu thập dữ liệu với mục đích rõ ràng – đặt ra mục tiêu, xác định rõ cần đo gì để trả lời câu hỏi gì, và chỉ thu thập lượng thông tin vừa phải, cân nhắc ưu tiên hoàn thành dự án hơn là thu thập dữ liệu (Basili và Weiss 1984, Basili et al. 2002).

Tài liệu tham khảo về đo lường phần mềm

- *Oman, Paul and Shari Lawrence Pfleeger, eds. Applying Software Metrics.* IEEE Computer Society Press, 1996.
  Tuyển tập hơn 25 bài viết tiêu biểu về đo lường phần mềm.

- *Jones, Capers. Applied Software Measurement: Assuring Productivity and Quality, 2nd ed.* New York, NY: McGraw-Hill, 1997.
  Tổng kết lý thuyết và thực tiễn hiện đại về đo lường năng suất và chất lượng phần mềm, giới thiệu “function-point metrics”.

- *Grady, Robert B. Practical Software Metrics for Project Management and Process Improvement.* Englewood Cliffs, NJ: Prentice Hall PTR, 1992.

Chia sẻ bài học từ chương trình đo lường phần mềm tại Hewlett-Packard và chỉ dẫn phương pháp thiết lập đo lường tại cơ quan.

- *Conte, S. D., H. E. Dunsmore, and V. Y. Shen. Software Engineering Metrics and Models.* Menlo Park, CA: Benjamin/Cummings, 1986.
  Tổng hợp kiến thức về đo lường phần mềm, kỹ thuật thực nghiệm và tiêu chí đánh giá ở thời điểm 1986.

- *Basili, Victor R., et al. "Lessons learned from 25 years of process improvement: The Rise and Fall of the NASA Software Engineering Laboratory." Proceedings of the 24th International Conference on Software Engineering. Orlando, FL, 2002.*
  Tổng kết bài học về đo lường của một trong những tổ chức phát triển phần mềm hàng đầu thế giới.

- NASA Software Engineering Laboratory. "Software Measurement Guidebook," 1995, NASA-GB-001-94.
  Sổ tay thực tế khoảng 100 trang về xây dựng, vận hành chương trình đo lường.

- *Gilb, Tom. Competitive Engineering.* Boston, MA: Addison-Wesley, 2004.
  Trình bày cách tiếp cận quản lý lấy đo lường làm trung tâm.

Đối xử với lập trình viên như con người

Hoạt động lập trình có tính trừu tượng cao, vì vậy môi trường làm việc cần sự tự nhiên và tăng cường tương tác giữa các đồng nghiệp. Nhiều công ty công nghệ hàng đầu xây dựng môi trường làm việc thân thiện, văn hóa doanh nghiệp "hữu cơ" và không gian tiện nghi để cân bằng với công việc đôi khi khô khan và đậm chất trí tuệ. Thành công nhất là những tổ chức kết hợp được "công nghệ cao" với "tinh thần nhân văn" (Naisbitt 1982).

Lập trình viên dành thời gian như thế nào?

Lập trình viên không chỉ lập trình; họ còn họp hành, đào tạo, đọc thư/công văn, và suy nghĩ. Một nghiên cứu tại Bell Laboratories năm 1964 cho thấy lập trình viên phân bổ thời gian như trong Bảng 28-3:

Hoạt động	Mail/Khác	Kỹ thuật	Thủ tục, Tài liệu, Hướng dẫn	Công việc cá nhân	Họp	Đào tạo	Test	Tổng cộng
Đàm thoại/lắng nghe	4%	17%	7%	3%	1%	32%		
Nói chuyện trực tiếp	1%	1%						
Điện thoại	2%	1%		3%				
Đọc tài liệu	14%	2%	2%		18%			
Ghi chép/ghi âm	13%	1%		14%				
Đi ra ngoài	4%	1%	4%	6%	15%			
Đi bộ	2%	2%	1%	1%	6%			
Khác	2%	3%	3%	1%	1%	11%		

Kết quả dựa trên việc quan sát 70 lập trình viên, nhấn mạnh rằng khoảng 30% thời gian của lập trình viên dành cho các hoạt động không liên quan trực tiếp đến kỹ thuật hoặc dự án, ví dụ đi bộ, công việc cá nhân, v.v. Đáng chú ý, họ dành thời gian đi bộ mỗi năm ngang bằng với thời gian đào tạo, gấp ba lần thời gian đọc tài liệu kỹ thuật và gấp sáu lần thời gian trao đổi với quản lý. Tác giả nhận xét rằng mô hình này hầu như không thay đổi cho đến hiện nay.

Sự khác biệt về năng suất và chất lượng giữa các lập trình viên

Năng lực và nỗ lực của các lập trình viên có sự khác biệt rất lớn, tương tự như các ngành nghề khác. Một nghiên cứu cho thấy, trong nhiều ngành nghề (viết lách, bóng đá, sáng chế, cảnh sát, phi công), 20% người đứng đầu tạo ra 50% thành quả (Augustine 1979). Nghiên cứu này dựa trên phân tích dữ liệu năng suất như số bàn thắng, bằng sáng chế, vụ án được phá, v.v. Do những người không tạo ra đóng góp hữu hình không tính vào nghiên cứu, dữ liệu này thực tế còn đánh giá thấp sự chênh lệch.

Khác biệt giữa các cá nhân

Nghiên cứu kinh điển của Sackman, Erikson, và Grant (1968) chỉ ra sự chênh lệch năng suất rất lớn giữa các lập trình viên có trung bình 7 năm kinh nghiệm:

- Thời gian lập trình (initial coding time) chênh lệch 20:1
- Thời gian gỡ lỗi (debugging time) chênh lệch 25:1
- Kích thước chương trình 5:1
- Tốc độ thực thi chương trình 10:1

Đặc biệt, nghiên cứu không tìm thấy mối liên hệ giữa số năm kinh nghiệm và chất lượng cũng như năng suất lập trình.

Nhiều nghiên cứu khác cũng xác nhận sự khác biệt theo cấp số nhân về năng suất giữa các lập trình viên chuyên nghiệp (Curtis 1981, Mills 1983, DeMarco và Lister 1985, Card 1987, Boehm và Papaccio 1988, v.v.).

Khác biệt giữa các nhóm

Nhóm lập trình cũng có sự khác biệt lớn về chất lượng phần mềm và năng suất. Lập trình viên giỏi thường tập trung thành nhóm tốt, và ngược lại nhóm yếu cũng tập trung nhiều lập trình viên yếu (Demarco và Lister 1999).

Một nghiên cứu trên bảy dự án giống hệt nhau cho thấy, tổng nỗ lực bỏ ra chênh lệch 3,4:1, còn kích thước chương trình chênh lệch 3:1 (Boehm, Gray, và Seewaldt 1984). Dù có sự chênh lệch lớn, nhưng các lập trình viên trong nghiên cứu đều là chuyên gia dày kinh nghiệm, tham gia chương trình cao học ngành khoa học máy tính.

Lưu ý: Một số lỗi đánh máy và định dạng trong bản gốc đã được làm rõ để đảm bảo bản dịch dễ hiểu và nhất quán.

Đối xử Với Lập Trình Viên Như Những Con Người

Sự Khác Biệt Giữa Các Nhóm Lập Trình

Một nghiên cứu trước đây về các nhóm lập trình đã quan sát thấy sự khác biệt 5 lần về kích thước chương trình và sự biến động 2,6 lần về thời gian cần thiết để một nhóm hoàn thành cùng một dự án (Weinberg and Schulman 1974).

Sau khi xem xét dữ liệu hơn 20 năm trong quá trình xây dựng mô hình Cocomo II, Barry Boehm và các nhà nghiên cứu khác kết luận rằng phát triển một chương trình với một nhóm ở phân vị thứ 15 (tức nằm trong 15% lập trình viên có năng lực thấp nhất) thường đòi hỏi khoảng **3,5 lần** số tháng-làm-việc so với một nhóm ở phân vị thứ 90 (tức nằm trong 10% lập trình viên xuất sắc hàng đầu) (Boehm et al, 2000). Boehm và các đồng nghiệp cũng phát hiện ra rằng 80% giá trị đóng góp đến từ 20% thành viên (Boehm 1987b).

Ý Nghĩa Cho Tuyển Dụng và Đãi Ngộ

> Nếu bạn phải trả lương cao hơn để thu hút một lập trình viên trong nhóm 10% xuất sắc nhất thay vì nhóm 10% thấp nhất, hãy làm ngay. Bạn sẽ nhận được lợi ích tức thì về chất lượng và năng suất của người bạn tuyển dụng, cũng như hiệu ứng lan toả tới những lập trình viên khác vì lập trình viên giỏi thường tập hợp lại cùng nhau.

Những Vấn Đề "Tôn Giáo" Trong Lập Trình

Các nhà quản lý dự án lập trình không phải lúc nào cũng nhận ra rằng một số vấn đề về lập trình mang tính "tôn giáo". Nếu bạn cố gắng áp đặt các quy tắc về những khía cạnh này, bạn có thể khiến lập trình viên của mình không hài lòng. Danh sách những vấn đề "tôn giáo" gồm có:

- Ngôn ngữ lập trình
- Kiểu thụt lề (indentation style)
- Cách đặt dấu ngoặc nhọn
- Lựa chọn IDE (Integrated Development Environment – Môi trường Phát triển Tích hợp)
- Phong cách chú thích (commenting style)
- Cân nhắc giữa hiệu suất và tính dễ đọc (efficiency vs readability tradeoffs)
- Lựa chọn phương pháp luận (methodology) – ví dụ, Scrum so với Extreme Programming hoặc evolutionary delivery
- Công cụ lập trình (programming utilities)
- Quy ước đặt tên (naming conventions)
- Sử dụng goto
- Sử dụng biến toàn cục (global variables)
- Các phép đo, đặc biệt là các phép đo năng suất như số dòng mã mỗi ngày

Điểm chung là quan điểm về mỗi chủ đề nói lên phong cách cá nhân của lập trình viên. Nếu bạn nghĩ cần kiểm soát họ ở các vấn đề này, hãy cân nhắc:

- Nhận thức đây là lĩnh vực nhạy cảm. Hỏi ý kiến lập trình viên về mỗi chủ đề trước khi ra quyết định.
- Sử dụng "gợi ý" hoặc "hướng dẫn" thay cho những "quy tắc" cứng nhắc.

- Xử lý khéo léo các vấn đề có thể bằng cách tránh đưa ra các quy định quá chi tiết. Ví dụ, với kiểu thụt lề hay cách đặt dấu ngoặc, yêu cầu mã nguồn phải được chạy qua công cụ định dạng (pretty-printer) trước khi hoàn thành. Với phong cách chú thích, yêu cầu phải kiểm tra mã nguồn và chỉnh sửa cho đến khi rõ ràng.

Hãy để lập trình viên xây dựng tiêu chuẩn của riêng mình thay vì áp đặt, tuy nhiên hãy yêu cầu chuẩn hóa về những khía cạnh bạn thực sự thấy quan trọng.

> Những vấn đề về phong cách nhỏ nhặt rất ít khi mang lại lợi ích đủ lớn để bù đắp cho việc làm giảm tinh thần của lập trình viên. Tuy nhiên, nếu bạn nhận thấy việc sử dụng goto hoặc biến toàn cục một cách tuỳ tiện, hay phong cách mã hoá khó đọc, việc kiểm soát sẽ cần thiết để cải thiện chất lượng. Nếu lập trình viên có trách nhiệm, điều này hiếm khi gây ra vấn đề lớn; các tranh cãi chủ yếu xoay quanh các sắc thái về phong cách mã hóa và bạn có thể tránh mà không ảnh hưởng tới dự án.

Môi Trường Vật Lý Làm Việc

Hãy làm thử phép so sánh: ra ngoại ô hỏi một nông dân về giá trị thiết bị trên mỗi công nhân, câu trả lời có thể lên tới trên 100.000 đôla/người. Ở thành phố, một quản lý phòng lập trình thường chỉ có thiết bị dưới 25.000 đôla/người (bàn, ghế, sách, máy tính).

Các nhà nghiên cứu DeMarco và Lister đã khảo sát 166 lập trình viên từ 35 tổ chức về chất lượng môi trường làm việc. Hầu hết đánh giá nơi làm việc là không đạt yêu cầu. Trong một kỳ thi lập trình, nhóm 25% xuất sắc nhất có văn phòng rộng hơn, yên tĩnh hơn, riêng tư hơn và ít bị gián đoạn hơn. Dưới đây là bảng tóm tắt:

Yếu tố môi trường	Top 25%	Bottom 25%
Diện tích riêng biệt	78 sq ft	46 sq ft
Không gian đủ yên tĩnh	57% có	29% có
Không gian đủ riêng tư	62% có	19% có
Có thể tắt điện thoại	52% có	10% có
Có thể chuyển tiếp cuộc gọi	76% có	19% có
Bị gián đoạn không cần thiết thường xuyên	38% có	76% có
Không gian làm cho lập trình viên cảm thấy được trân trọng	57% có	29% có

Nguồn: *Peopleware* (DeMarco and Lister 1999)

Các dữ liệu này chỉ ra sự tương quan mạnh giữa năng suất và chất lượng môi trường làm việc. Nhóm 25% trên cùng có năng suất cao hơn nhóm 25% dưới cùng tới 2,6 lần. Sau khi kiểm tra, DeMarco và Lister xác nhận rằng điều kiện văn phòng không phải do được thăng chức mà có được.

Các tổ chức lớn như Xerox, TRW, IBM, Bell Labs cũng ghi nhận năng suất tăng lên đáng kể khi đầu tư từ 10.000 đến 30.000 đôla mỗi người vào cải thiện môi trường làm việc – số tiền này nhanh chóng được bù lại nhờ hiệu quả tăng lên (Boehm 1987a).

> Tóm lại, nếu nơi làm việc của bạn đang thuộc nhóm 25% kém nhất, bạn có thể tăng gấp đôi năng suất bằng cách cải thiện lên mức top 25%. Nếu nơi làm việc trung bình, bạn vẫn có thể tăng ít nhất 40% năng suất nếu nâng nó lên top 25%.

Tài Nguyên Tham Khảo Thêm

- **Weinberg, Gerald M.** *The Psychology of Computer Programming,* 2nd ed. New York, NY: Van Nostrand Reinhold, 1998.
    - Đây là cuốn sách đầu tiên nhận diện lập trình viên như con người với nhiều nhận xét sâu sắc về bản chất con người và tác động tới nghề nghiệp lập trình.
- **DeMarco, Tom and Timothy Lister.** *Peopleware: Productive Projects and Teams,* 2nd ed. New York, NY: Dorset House, 1999.
    - Bàn về yếu tố con người trong các dự án lập trình; nhiều câu chuyện thực tế về môi trường làm việc, tuyển dụng, phát triển đội nhóm.
- **McCue, Gerald M.** "IBM's Santa Teresa Laboratory—Architectural Design for Program Development," *IBM Systems Journal* 17, no 1 (1978): 4–25.
    - Mô tả quá trình IBM thiết kế văn phòng tối ưu cho lập trình viên, với sự tham gia của chính lập trình viên vào toàn bộ quá trình.
- **McConnell, Steve.** *Professional Software Development.* Boston, MA: Addison-Wesley, 2004 – Chương 7 "Orphans Preferred" tổng kết về nhân khẩu học của lập trình viên.
- **Carnegie, Dale.** *How to Win Friends and Influence People*, Revised Edition. New York, NY: Pocket Books, 1981.
    - Sách kinh điển về quản lý con người và tạo dựng mối quan hệ; nhấn mạnh tới tầm quan trọng của việc quan tâm chân thành tới người khác.

28.6 Quản Lý Người Quản Lý Của Bạn

Trong phát triển phần mềm, không hiếm các quản lý không có nền tảng kỹ thuật, hoặc có kinh nghiệm nhưng lạc hậu về công nghệ. Quản lý giỏi, cập nhật xu hướng kỹ thuật là rất hiếm; nếu bạn có một người như vậy, hãy giữ lấy vị trí đó.

> "Trong một hệ thống phân cấp, mỗi nhân viên có xu hướng thăng tiến tới mức độ bất tài của mình."
> — *The Peter Principle*

Nếu quản lý của bạn thuộc số đông còn lại, bạn sẽ phải đối mặt với nhiệm vụ khó chịu là "quản lý người quản lý". Tức là, bạn phải chỉ dẫn cho quản lý làm điều đúng, mà không làm ảnh hưởng tới vai trò quản lý. Một số chiến thuật:

- Gợi ý các ý tưởng bạn mong muốn thực hiện, rồi chờ quản lý "nảy ra ý" giống bạn.
- Liên tục cập nhật cho quản lý về cách làm đúng.
- Chú trọng lợi ích thật sự của quản lý, tránh đưa ra chi tiết kỹ thuật không cần thiết (hãy xem đó như việc "đóng gói" (encapsulation) công việc của bạn).
- Từ chối các yêu cầu sai, kiên quyết làm việc theo cách đúng đắn.
- Hoặc, tìm một công việc khác.

Giải pháp lâu dài nhất là cố gắng giáo dục lại quản lý của bạn, tham khảo cuốn *How to Win Friends and Influence People* của Dale Carnegie để cải thiện kỹ năng.

Tài Nguyên Tham Khảo về Quản Lý Xây Dựng Phần Mềm

- **Gilb, Tom.** *Principles of Software Engineering Management*. Addison-Wesley, 1988.

    - Giới thiệu sớm về phát triển tiến hóa (evolutionary development), quản lý rủi ro, kiểm tra chính thức; nhiều ý tưởng nay thuộc nhóm Agile.
- **McConnell, Steve.** *Rapid Development*. Microsoft Press, 1996.
    - Các vấn đề quản lý dự án, đặc biệt trong tình huống tiến độ gấp rút.
- **Brooks, Frederick P. Jr.** *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition (2nd ed), Addison-Wesley, 1995.
    - Đầy ắp các ẩn dụ và kinh nghiệm về quản lý dự án lập trình; cần đọc kỹ để tách biệt quan sát thực tế với phỏng đoán.

Các Tiêu Chuẩn Liên Quan

- **IEEE Std 1058-1998**: Standard for Software Project Management Plans
- **IEEE Std 12207-1997**: Information Technology—Software Life Cycle Processes
- **IEEE Std 1045-1992**: Standard for Software Productivity Metrics
- **IEEE Std 1062-1998**: Recommended Practice for Software Acquisition
- **IEEE Std 1540-2001**: Standard for Software Life Cycle Processes—Risk Management
- **IEEE Std 828-1998**: Standard for Software Configuration Management Plans
- **IEEE Std 1490-1998**: Guide—Adoption of PMI Standard—A Guide to the Project Management Body of Knowledge

---

Các Ý Chính

- **Thực hành lập trình tốt** có thể đạt được thông qua các tiêu chuẩn bắt buộc hoặc các hướng tiếp cận linh hoạt hơn.
- **Quản lý cấu hình (configuration management)**, nếu thực hiện đúng, sẽ giúp công việc của lập trình viên nhẹ nhàng hơn, nhất là kiểm soát thay đổi (change control).
- **Ước lượng phần mềm** là một thách thức lớn; thành công dựa trên việc sử dụng nhiều phương pháp, củng cố ước lượng khi bước vào dự án và tận dụng dữ liệu thực tế.
- **Đo lường** là chìa khoá cho việc quản lý xây dựng hiệu quả; đo lường chính xác giúp lập lịch, kiểm soát chất lượng và cải tiến quy trình phát triển.
- **Lập trình viên và quản lý đều là con người** và họ làm tốt hơn khi được đối xử như vậy.

Chương 29: Tích Hợp (Integration)

Nội dung

- **29.1 Tầm Quan Trọng của Phương Pháp Tích Hợp**
- **29.2 Tần Suất Tích Hợp – Tích Hợp Theo Pha hay Gia Tăng?**
- **29.3 Chiến lược Tích Hợp Gia Tăng**
- **29.4 Daily Build và Smoke Test**

Liên quan:

- Kiểm thử của nhà phát triển: Chương 22
- Gỡ lỗi: Chương 23
- Quản lý xây dựng: Chương 28

Định nghĩa “Tích Hợp” (Integration)

“Tích hợp” trong phát triển phần mềm là hoạt động kết hợp các thành phần riêng lẻ thành một hệ thống hoàn chỉnh. Với dự án nhỏ, có thể chỉ mất một buổi sáng để nối các class lại với nhau; với dự án lớn, có thể mất nhiều tuần, nhiều tháng để ghép các nhóm chương trình. Dù quy mô ra sao, các nguyên tắc chung vẫn luôn áp dụng.

Tích hợp luôn gắn liền với trình tự xây dựng (construction sequence). Thứ tự bạn xây dựng các class hoặc thành phần sẽ quyết định thứ tự bạn có thể tích hợp – bạn không thể tích hợp những gì chưa được xây dựng! Cả tích hợp lẫn trình tự xây dựng đều quan trọng; chương này tập trung từ góc nhìn tích hợp.

29.1 Tầm Quan Trọng của Phương Pháp Tích Hợp

Trong kỹ thuật ngoài ngành phần mềm, vai trò của tích hợp đúng cách luôn được nhận biết. Ở vùng Tây Bắc nước Mỹ, đã từng ghi nhận một sự cố nghiêm trọng: sân vận động bóng đá của Đại học Washington bị sập trong quá trình xây dựng, do tích hợp sai trình tự – một lỗi tích hợp điển hình (xem Hình 29-1).

> Sân vận động có thể đủ vững khi hoàn thiện, nhưng từng giai đoạn phải đảm bảo ổn định. Nếu xây dựng hoặc tích hợp phần mềm sai thứ tự, sẽ khó lập trình, kiểm thử, và gỡ lỗi. Nếu mọi thứ chỉ hoạt động khi tất cả đã ghép lại, bạn sẽ có cảm giác dự án không bao giờ hoàn thành. Dự án có thể “sụp đổ” vì số lượng lỗi không kiểm soát, tiến độ không thấy rõ, hoặc độ phức tạp quá sức chịu đựng – dẫu sản phẩm cuối cùng có thể vận hành tốt.

Tích hợp thường diễn ra sau kiểm thử của nhà phát triển (developer testing) và song song với kiểm thử hệ thống (system testing), nên đôi khi bị nhầm là chỉ là một hoạt động kiểm thử. Thực ra, tích hợp đủ phức tạp để cần xem xét như một hoạt động độc lập.

Lợi Ích của Tích Hợp Cẩn Thận

Bạn có thể kỳ vọng những lợi ích sau từ tích hợp được chuẩn bị kỹ:

- **Dễ xác định nguyên nhân lỗi**
- **Ít lỗi hơn**
- **Giảm mã bổ trợ (scaffolding)**
- **Rút ngắn thời gian để có sản phẩm đầu tiên hoạt động**
- **Rút ngắn tổng thời gian phát triển**
- **Mối quan hệ khách hàng tốt hơn**
- **Tinh thần làm việc cao hơn**
- **Tăng khả năng hoàn thành dự án**
- **Ước lượng lịch trình đáng tin cậy hơn**
- **Báo cáo tiến độ chính xác hơn**
- **Nâng cao chất lượng mã nguồn**
- **Giảm khối lượng tài liệu cần thiết**

Những lợi ích này dù có vẻ lớn nhưng thường bị bỏ qua, do vậy tích hợp xứng đáng có một chương riêng.

29.2 Tần Suất Tích Hợp – Theo Pha hay Gia Tăng?

Tích Hợp Theo Pha (Phased Integration)

Trước đây, phương pháp phổ biến là tích hợp theo pha, bao gồm các bước:

1. Thiết kế, lập trình, kiểm thử và sửa lỗi từng class riêng lẻ (unit development).
2. Kết hợp các class thành hệ thống lớn (system integration).
3. Kiểm thử, sửa lỗi toàn hệ thống ("system dis-integration" – một nhận xét hài hước từ Meilir Page-Jones).

Vấn đề của tích hợp theo pha: Khi các class được kết hợp lần đầu, nhiều vấn đề mới xuất hiện mà nguyên nhân có thể nằm ở bất kỳ đâu: một class kiểm thử chưa kỹ, lỗi giao tiếp giữa hai class, hoặc do tương tác khó lường giữa các class. Tình trạng càng trầm trọng khi nhiều vấn đề xuất hiện cùng lúc, khiến việc xác định nguyên nhân trở nên khó khăn hơn.

Do đó, tích hợp kiểu này còn được gọi là **"big bang integration"**, như minh họa dưới đây:

```
Global
variables
Different error-
handling
assumptions Big Bang
Integration
Poorly
documented
interfaces
Weak
encapsulation
```

Tích hợp theo pha chỉ có thể bắt đầu ở giai đoạn cuối dự án, sau khi tất cả các class đã kiểm thử xong. Khi các class được ghép lại, lỗi xuất hiện hàng loạt khiến các lập trình viên phải "chữa cháy" thay vì xử lý bài bản.

Với các chương trình nhỏ – thật sự nhỏ – tích hợp theo pha có thể hiệu quả. Nếu chỉ có hai hoặc ba class, có thể tiết kiệm thời gian. Tuy nhiên, phần lớn trường hợp, **phương pháp khác sẽ tốt hơn**.

Tích Hợp Gia Tăng (Incremental Integration)

Ở phương pháp tích hợp gia tăng, bạn viết, kiểm thử chương trình theo từng phần nhỏ rồi lần lượt tích hợp từng phần đó vào hệ thống chung.

Dịch thuật học thuật: Tích hợp từng phần (Incremental Integration) trong phát triển phần mềm

1. Phát triển phần chức năng nhỏ của hệ thống

Bạn hãy phát triển một phần nhỏ, có chức năng cụ thể của hệ thống. Đây có thể là phần nhỏ nhất có thể thực thi được, phần khó nhất, phần quan trọng, hoặc sự kết hợp của những lựa chọn này.

Cần kiểm thử (test) và gỡ lỗi (debug) kỹ càng cho phần này. Phần này sẽ đóng vai trò là "bộ khung" (skeleton) để sau này bổ sung thêm các thành phần còn lại của hệ thống.

2. Thiết kế, lập trình, kiểm thử và gỡ lỗi một lớp (class)

3. Tích hợp lớp mới với bộ khung

Hãy tích hợp lớp vừa phát triển vào bộ khung, sau đó kiểm thử và gỡ lỗi tổ hợp bộ khung và lớp mới. Đảm bảo rằng bộ khung cùng lớp vừa tích hợp hoạt động đúng trước khi bổ sung thêm các lớp mới khác. Nếu vẫn còn phải tiếp tục phát triển, lặp lại quá trình từ bước 2.

Thỉnh thoảng, bạn có thể muốn tích hợp các đơn vị lớn hơn một lớp. Ví dụ, nếu một thành phần (component) đã được kiểm thử kỹ càng và mỗi lớp của thành phần đó đã được kiểm thử tích hợp nhỏ lẻ, bạn có thể tích hợp toàn bộ thành phần mà vẫn thực hiện tích hợp từng bước (incremental integration). Khi bổ sung từng thành phần, hệ thống sẽ dần hoàn thiện và tăng trưởng dần về chức năng, tương tự cách một quả cầu tuyết lớn dần và lăn nhanh hơn khi lăn xuống dốc (xem Hình 29-3).

> **Hình 29-3:** Tích hợp từng bước (incremental integration) giúp dự án phát triển mạnh mẽ, giống như một quả cầu tuyết lăn xuống dốc.

29.2 Tần suất tích hợp—Từng pha hay từng bước?

Lợi ích của tích hợp từng bước (Incremental Integration)

Phương pháp tích hợp từng bước mang lại nhiều ưu điểm so với phương pháp tích hợp theo từng pha (phased approach), bất kể bạn sử dụng chiến lược nào:

- **Dễ xác định vị trí lỗi:** Khi xuất hiện lỗi trong quá trình tích hợp từng bước, lớp mới vừa được tích hợp thường là nơi xảy ra vấn đề, dù lỗi nằm ở giao diện (interface) với chương trình hoặc tương tác với các lớp đã tích hợp trước đó. Như gợi ý ở Hình 29-4, bạn biết chính xác cần kiểm tra ở đâu. Ngoài ra, do số lượng vấn đề xuất hiện cùng lúc giảm, bạn hạn chế được rủi ro các lỗi tương tác với nhau hoặc che giấu lẫn nhau. Với dự án nhiều lỗi giao diện, hiệu quả gỡ lỗi của thực tiễn này sẽ càng rõ rệt. Ví dụ, một dự án ghi nhận 39% lỗi liên quan tới giao diện giữa các mô-đun (Basili và Perricone, 1984). Do đó, việc tối đa hóa hiệu quả gỡ lỗi bằng cách khiến lỗi dễ xác định sẽ giúp nâng cao chất lượng và năng suất.

  Hình 29-4: Ở mô hình tích hợp theo pha, bạn tích hợp nhiều thành phần cùng lúc nên không dễ xác định lỗi nằm ở đâu—có thể ở bất kỳ thành phần hoặc kết nối nào. Với tích hợp từng bước, lỗi thường nằm ở thành phần mới hoặc kết nối giữa thành phần mới và hệ thống.

- **Hệ thống có thể vận hành sớm:** Khi mã nguồn được tích hợp và chạy, dù ban đầu hệ thống chưa hoàn chỉnh, bạn sẽ sớm thấy nó khả dụng. Nhờ đó lập trình viên có động lực hơn so với việc lo ngại dự án của họ sẽ không bao giờ được nghiệm thu.
- **Tiến độ dự án được theo dõi sát sao hơn:** Tích hợp thường xuyên giúp các chức năng đã và chưa hoàn thành lộ rõ. Quản lý dự án sẽ có cái nhìn trực quan hơn về tiến độ, ví dụ nhìn thấy 50% chức năng đã vận hành còn có ý nghĩa hơn việc nghe báo cáo "đã hoàn thành 99%".

- **Cải thiện mối quan hệ với khách hàng:** Tích hợp định kỳ giúp duy trì tinh thần tích cực không chỉ với lập trình viên mà cả với khách hàng, nhờ các dấu hiệu tiến bộ thường xuyên.
- **Các đơn vị hệ thống được kiểm thử đầy đủ hơn:** Tích hợp được thực hiện từ sớm và liên tục, mỗi lớp vừa xây dựng sẽ được vận hành như một phần của hệ thống tổng thể nhiều lần, khác với việc tích hợp ồ ạt ở cuối dự án.
- **Có thể rút ngắn lịch trình phát triển:** Nếu tích hợp được lên kế hoạch kỹ càng, một phần hệ thống có thể được thiết kế trong lúc phần khác đang lập trình. Điều này tạo điều kiện thực hiện công việc song song, giúp giảm thời gian lịch trình tổng thể.
- **Khuyến khích các chiến lược phát triển từng bước khác:** Lợi ích của tích hợp từng bước chỉ là phần nổi của tảng băng chìm về các lợi ích của chia nhỏ và phát triển theo từng bước.

29.3 Các chiến lược tích hợp từng bước (Incremental Integration Strategies)

Sự Khác Biệt Về Lập Kế Hoạch Giữa Tích Hợp Theo Pha Và Tích Hợp Từng Bước

Với tích hợp theo pha (phased integration), bạn không cần phải lập kế hoạch thứ tự phát triển các thành phần—mọi thành phần sẽ được tích hợp cùng lúc, miễn là tất cả đều sẵn sàng.

Với tích hợp từng bước (incremental integration), bạn phải lập kế hoạch kỹ hơn. Phần lớn các hệ thống sẽ yêu cầu tích hợp một số thành phần trước những thành phần khác. Do đó, trình tự tích hợp tác động trực tiếp tới trình tự xây dựng và thiết kế các thành phần.

Các chiến lược chọn thứ tự tích hợp đa dạng và không có một phương án tối ưu cho mọi trường hợp. Giá trị thực tiễn của các phương pháp này là bạn có thêm góc nhìn để chọn giải pháp phù hợp với dự án cụ thể.

Tích hợp từ trên xuống (Top-Down Integration)

Trong phương pháp tích hợp từ trên xuống, lớp (class) ở đỉnh của hệ thống phân cấp sẽ được phát triển và tích hợp trước. Đỉnh này có thể là cửa sổ chính (main window), vòng lặp điều khiển ứng dụng (application control loop), đối tượng chứa hàm `main()` (trong Java), `WinMain()` (với lập trình Microsoft Windows), hay tương tự.

Cần viết các đoạn mã giả lập (stub) để kiểm thử lớp ở đỉnh này. Khi dần tích hợp các lớp từ trên xuống, các lớp giả lập sẽ dần được thay thế bởi lớp thực tế. Phương pháp này được minh họa ở Hình 29-5.

> **Hình 29-5:** Trong tích hợp từ trên xuống, bạn bổ sung các lớp ở phía trên trước và các lớp thấp nhất sau cùng.

Một điểm quan trọng khi dùng tích hợp từ trên xuống là phải xác định rõ ràng giao diện giữa các lớp. Những lỗi khó xử lý nhất không phải do một lớp đơn lẻ mà thường do tương tác phức tạp giữa các lớp. Việc đặc tả giao diện cẩn thận có thể giảm thiểu rủi ro này.

Ưu điểm:

- Kiểm thử logic điều khiển hệ thống từ rất sớm.

- Nếu thiết kế hợp lý, có thể hoàn thành một hệ thống đơn giản khả dụng từ đầu dự án, đặc biệt nếu các thành phần giao diện người dùng nằm ở phía trên, giúp động viên cả nhóm phát triển và người dùng.

Nhược điểm:

- Các lỗi hoặc vấn đề hiệu năng liên quan tới giao diện hệ thống sâu thường bị phát hiện muộn—có thể gây ra việc phải thay đổi lớn ở cấp cao dù đã tích hợp gần hoàn tất.
- Cần số lượng lớn các mã giả lập (stub) trong giai đoạn trung gian, dễ sinh lỗi nhiều hơn mã sản xuất thật.

Thông thường, khó hoặc không khả thi để thực hiện tích hợp từ trên xuống một cách "thuần túy". Đa số dự án sử dụng cách "lát dọc" (vertical-slice)—tức là thực hiện tích hợp từ trên xuống theo từng phần chức năng (xem Hình 29-6).

> **Hình 29-6:** Thay vì tích hợp hoàn toàn từ trên xuống, bạn có thể tích hợp hệ thống theo từng lát dọc chức năng.

Tích hợp từ dưới lên (Bottom-Up Integration)

Ngược lại, tích hợp từ dưới lên bắt đầu từ các lớp thấp nhất trong hệ thống phân cấp. Cần phát triển các trình kiểm thử (test driver) để kiểm thử các lớp này. Khi đã hoàn tất các lớp cao hơn, các test driver được thay thế dần bằng các lớp thực. Hình 29-7 minh họa phương pháp này.

> **Hình 29-7:** Với tích hợp từ dưới lên, các lớp thấp nhất được tích hợp trước, lớp cao nhất sau cùng.

Ưu điểm:

- Giới hạn nguồn lỗi có khả năng xuất hiện ở lớp vừa tích hợp.
- Có thể phát hiện và xử lý sớm các giới hạn của hệ thống ở tầng thấp—các vấn đề ảnh hưởng trực tiếp đến khả năng đạt mục tiêu kỹ thuật.

Nhược điểm:

- Các lỗi lớn trong thiết kế ở tầng trên chỉ được phát hiện sau khi hoàn thiện tất cả chi tiết tầng dưới, có thể dẫn đến việc phải loại bỏ hoặc sửa lại khối lượng lớn công sức đã bỏ ra.
- Yêu cầu phải hoàn chỉnh thiết kế tổng thể từ đầu dự án, nếu không các chi tiết tầng dưới rất dễ trở thành rào cản hoặc gây khó khăn cho thiết kế tầng trên về sau—trái với các nguyên lý giấu thông tin (information hiding) và thiết kế hướng đối tượng (object-oriented design).

Cũng giống như phương pháp tích hợp từ trên xuống, cách tiếp cận thuần túy này ít được sử dụng hơn các phương pháp lai/hỗn hợp, ví dụ tích hợp từ dưới lên theo từng phần (xem Hình 29-8).

> **Hình 29-8:** Thay vì tích hợp toàn bộ từ dưới lên, bạn có thể tích hợp theo từng khu vực chức năng—đây là phương án trung gian giữa bottom-up integration và feature-oriented integration.

Tích hợp kiểu "bánh mì kẹp" (Sandwich Integration)

Các nhược điểm của hai phương pháp thuần túy nêu trên đã dẫn đến việc một số chuyên gia đề xuất cách tiếp cận "bánh mì kẹp" (sandwich approach):

- Đầu tiên, tích hợp các lớp nghiệp vụ (business-object classes) ở đầu trên của hệ thống phân cấp.
- Tiếp theo, tích hợp các lớp giao diện thiết bị (device-interface classes) và các lớp tiện ích dùng chung (utility classes) ở tầng dưới cùng (hai tầng này giống "vỏ bánh mì").
- Các lớp tầng giữa (middle-level classes)—chính là "nhân bánh mì"—được thực hiện sau (xem Hình 29-9).

  Hình 29-9: Trong sandwich integration, các lớp ở đỉnh và đáy hệ thống được tích hợp trước, các lớp tầng giữa để lại sau.

Ưu điểm của phương pháp này:

- Tránh sự cứng nhắc của bottom-up hay top-down thuần tuý.
- Ưu tiên giải quyết các lớp thường xuyên gây vấn đề trước, đồng thời giảm khối lượng mã giả lập/kiểm thử cần thiết trong giai đoạn trung gian.
- Là lựa chọn thực tế, linh hoạt trong nhiều trường hợp dự án phần mềm.

Kết luận

Phương pháp tích hợp từng bước và các hình thức linh hoạt của nó giúp phát hiện lỗi nhanh hơn, rút ngắn lịch trình, nâng cao chất lượng sản phẩm cũng như động lực cho cả đội phát triển lẫn khách hàng. Việc cân nhắc lựa chọn phương pháp và trình tự tích hợp phù hợp với tính chất dự án sẽ quyết định hiệu quả tổng thể của quá trình phát triển phần mềm.

Tích hợp hướng rủi ro (Risk-Oriented Integration)

Khái niệm

Tích hợp hướng rủi ro còn được gọi là "tích hợp phần khó trước" (hard part first integration). Phương pháp này tương tự tích hợp kiểu sandwich ở chỗ nó nhằm tránh các vấn đề vốn có của các phương pháp tích hợp từ trên xuống (top-down) hoặc từ dưới lên (bottom-up). Tình cờ, phương pháp này cũng thường tiến hành tích hợp các lớp (class) ở cấp cao nhất và thấp nhất trước, để lại các lớp ở giữa cho giai đoạn cuối cùng. Tuy nhiên, động cơ của phương pháp này lại khác biệt.

Cách tiếp cận

Trong tích hợp hướng rủi ro, bạn xác định mức độ rủi ro liên quan đến từng lớp. Bạn quyết định phần nào là khó khăn nhất để triển khai, và thực hiện chúng trước. Thực tiễn cho thấy, các interface (giao diện) cấp cao thường có mức rủi ro lớn, vì vậy chúng thường nằm trong danh sách ưu tiên. Các interface hệ thống (system interfaces), thường ở cấp thấp nhất của hệ thống phân cấp, cũng tiềm ẩn rủi ro cao và cần được xử lý ngay từ đầu. Ngoài ra, bạn có thể nhận biết một số lớp ở giữa hệ thống sẽ rất thách thức, ví dụ như một lớp thực hiện một thuật toán chưa được hiểu rõ hoặc có yêu cầu hiệu năng cao. Những lớp như vậy cũng được xác định là rủi ro cao và được tích hợp sớm.

Phần còn lại của mã nguồn, những phần dễ dàng hơn, có thể để lại giải quyết sau. Tuy nhiên, một số trong đó có thể khó hơn dự đoán, nhưng đây là điều không thể tránh khỏi. Hình 29-10 minh họa cho quá trình tích hợp hướng rủi ro.

> **Minh họa:** Trong tích hợp hướng rủi ro, bạn tích hợp các lớp dự kiến có nhiều rắc rối nhất trước và triển khai các lớp dễ hơn sau.

Tích hợp hướng tính năng (Feature-Oriented Integration)

Một phương pháp khác là tích hợp từng *feature* (chức năng đặc trưng) một. Thuật ngữ "feature" ở đây không mang ý nghĩa phức tạp, mà chỉ đơn giản là một chức năng nhận diện được của hệ thống cần tích hợp. Ví dụ, nếu bạn phát triển một trình xử lý văn bản, một feature có thể là khả năng hiển thị chữ gạch dưới trên màn hình hoặc tự động định dạng lại tài liệu.

Khi feature cần tích hợp lớn hơn một lớp duy nhất, "mỗi increment" (gia tăng) trong tích hợp gia tăng (incremental integration) sẽ lớn hơn một lớp. Điều này ảnh hưởng phần nào đến lợi ích của tính gia tăng, vì làm giảm sự chắc chắn về nguồn gốc của các lỗi mới. Tuy nhiên, nếu bạn đã kiểm thử toàn diện các lớp triển khai feature mới trước khi tích hợp, thì nhược điểm này là không đáng kể. Bạn có thể sử dụng các chiến lược tích hợp gia tăng một cách đệ quy: tích hợp các thành phần nhỏ để hình thành feature, sau đó tích hợp các feature thành hệ thống hoàn chỉnh.

Thông thường, bạn sẽ bắt đầu với một khung xương (skeleton) đã được chọn trước vì có khả năng hỗ trợ các feature khác. Trong một hệ thống tương tác, feature đầu tiên có thể là hệ thống menu tương tác. Sau đó, các feature khác sẽ được triển khai dựa trên feature đầu tiên đã được tích hợp. Hình 29-11 minh họa phương pháp này dưới dạng đồ họa.

> **Minh họa:** Trong tích hợp hướng tính năng, bạn tích hợp các lớp theo nhóm tạo thành các feature nhận dạng được—thường là nhiều lớp cùng lúc.

Các thành phần được thêm vào theo mô hình "feature tree" (cây chức năng)—tập hợp phân cấp các lớp cấu thành feature. Việc tích hợp sẽ dễ dàng hơn nếu mỗi feature khá độc lập, chỉ gọi các mã thư viện cấp thấp chung cho các feature khác, và không chia sẻ mã với các lớp ở mức trung gian. (Các lớp thư viện cấp thấp dùng chung không được thể hiện trong hình minh hoạ.)

Ưu điểm

Tích hợp hướng tính năng mang lại ba lợi ích chính:

1. Loại bỏ hầu hết nhu cầu về *scaffolding* (mã tạm hỗ trợ) cho mọi thành phần ngoại trừ các lớp thư viện cấp thấp. Khung xương có thể cần một chút scaffolding, hoặc một số phần đơn giản chưa hoạt động cho đến khi feature tương ứng được thêm vào, nhưng khi mỗi feature đã được gắn vào cấu trúc, không cần thêm scaffolding nào nữa.
2. Mỗi feature được tích hợp đồng nghĩa với việc bổ sung chức năng thực tế, chứng minh rằng dự án đang tiến triển đều đặn. Bạn có thể cung cấp phần mềm có chức năng cụ thể cho khách hàng đánh giá hoặc phát hành sớm hơn dù chức năng chưa đầy đủ.
3. Phương pháp này phù hợp tự nhiên với *object-oriented design* (thiết kế hướng đối tượng), bởi vì các object (đối tượng) thường tương ứng tốt với feature.

Lưu ý

Tích hợp hoàn toàn theo hướng feature cũng khó thực hiện như tích hợp thuần top-down hoặc bottom-up; một số mã cấp thấp nhất định phải được tích hợp trước khi một số feature quan trọng có thể vận hành.

Tích hợp hình chữ T (T-Shaped Integration)

Một tiếp cận cuối cùng thường được dùng để khắc phục các vấn đề của tích hợp top-down và bottom-up là "tích hợp hình chữ T" (T-shaped integration). Theo phương pháp này, một nhánh dọc (vertical slice) cụ thể được chọn để phát triển và tích hợp sớm. Nhánh này nên kiểm thử hệ thống theo chiều dọc (end-to-end) và có khả năng phát hiện sớm các vấn đề lớn trong giả định thiết kế của hệ thống. Sau khi nhánh dọc tiêu biểu này được triển khai (và các vấn đề liên quan đã được xử lý), bề rộng tổng thể của hệ thống có thể được phát triển tiếp (ví dụ như hệ thống menu trong ứng dụng desktop). Cách tiếp cận này thường được kết hợp với tích hợp hướng rủi ro hoặc hướng tính năng.

> **Minh hoạ:** Trong tích hợp hình chữ T, bạn xây dựng và tích hợp một nhánh dọc sâu để xác minh các giả định kiến trúc, sau đó phát triển bề rộng hệ thống để tạo khung phát triển các chức năng còn lại.

Tóm tắt các phương pháp tích hợp

Các phương pháp như bottom-up, top-down, sandwich, risk-oriented, feature-oriented, hay T-shaped—liệu bạn có cảm thấy chúng như những cái tên được "sáng tạo" ngẫu nhiên? Quả thật là vậy. Không có phương pháp nào trong số này là quy trình chuẩn mà bạn cần tuân thủ rập khuôn từ bước 1 đến bước 47 rồi kết luận là hoàn thành. Cũng giống như các phương pháp thiết kế phần mềm, đây là những heuristics (phép thử nghiệm), không phải thuật toán, và bạn nên linh hoạt tạo ra chiến lược độc đáo phù hợp với dự án cụ thể của mình thay vì tuân thủ máy móc.

Tích hợp và kiểm thử smoke hằng ngày (Daily Build and Smoke Test)

Khái niệm

Bất kể bạn chọn chiến lược tích hợp nào, một phương pháp hiệu quả là "daily build and smoke test" (xây dựng và kiểm thử smoke mỗi ngày). Mỗi file được biên dịch, liên kết và kết hợp thành chương trình thực thi hằng ngày, sau đó chương trình được chạy qua "smoke test"—một kiểm tra tương đối đơn giản để đảm bảo sản phẩm không "bốc khói" khi chạy.

Lợi ích

Quy trình này mang lại nhiều lợi ích quan trọng:

- Giảm thiểu rủi ro về chất lượng thấp, vốn liên quan đến rủi ro tích hợp không thành công hoặc có vấn đề. Kiểm thử smoke tất cả mã nguồn mỗi ngày giúp ngăn chặn các vấn đề chất lượng kiểm soát dự án. Bạn duy trì hệ thống ở trạng thái ổn định, không cho phép nó xuống cấp tới mức khó kiểm soát về chất lượng trước khi phát hiện ra vấn đề.
- Dễ dàng chẩn đoán lỗi: Khi sản phẩm được build và test hằng ngày, rất dễ xác định tại sao sản phẩm bị lỗi bất kỳ ngày nào. Nếu sản phẩm hoạt động vào ngày 17 nhưng lỗi vào ngày 18, lỗi do thay đổi nào đó giữa hai lần build.
- Tăng tinh thần làm việc: Việc thấy sản phẩm hoạt động mang lại động lực lớn cho đội ngũ phát triển, kể cả chỉ là hiển thị một hình chữ nhật! Khi build hằng ngày, mỗi ngày lại có chút chức năng hoạt động, giúp duy trì tinh thần tích cực.
- Hiện ra những việc chưa hoàn tất, tránh việc dồn đống đến cuối dự án (khiến dự án bị trì hoãn nhiều tuần hoặc thậm chí nhiều tháng). Đội không quen với daily build có thể thấy bị chậm đi, nhưng thực chất là vì công việc được phân bổ đều suốt dự án, giúp mọi người có cái nhìn chính xác về tiến độ thực sự.

Một số lưu ý khi áp dụng daily build:

- **Build hằng ngày:** Phần quan trọng nhất của daily build là tính "hằng ngày". Hãy xem daily build như nhịp đập của dự án. Nếu không có nhịp đập, dự án xem như đã chết.
- **Kiểm tra build lỗi:** Để quy trình hiệu quả, phần mềm build ra phải hoạt động được. Nếu không dùng được, build được xem là "lỗi" và việc sửa trở thành ưu tiên số một. Chuẩn mực về "lỗi build" nên đủ chặt để loại trừ các lỗi nghiêm trọng nhưng đủ linh hoạt để bỏ qua lỗi vụn vặt.
    - Một build "tốt" tối thiểu phải:
        - Biên dịch thành công tất cả các file, thư viện, thành phần.
        - Liên kết thành công mọi thành phần.
        - Không có lỗi nghiêm trọng khiến chương trình không thể khởi động hoặc gây nguy hại khi vận hành (tức là pass smoke test).
- **Smoke test mỗi ngày:** Phải kiểm thử toàn bộ hệ thống từ đầu đến cuối (end-to-end). Không cần phải kiểm thử kỹ lưỡng, chỉ cần đủ để phát hiện lỗi lớn, đảm bảo build này đủ ổn định để kiểm thử sâu hơn.
- **Giữ smoke test luôn cập nhật:** Smoke test phải phát triển cùng hệ thống. Ban đầu có thể chỉ kiểm tra đơn giản như "Hello, World!", về sau càng nhiều chức năng sẽ bổ sung vào smoke test. Nếu không cập nhật, smoke test sẽ đánh lừa người phát triển về chất lượng thực tế.
- **Tự động hóa build và smoke test:** Nếu không tự động hóa, việc build và chạy smoke test mỗi ngày sẽ rất mất thời gian và khó duy trì.
- **Thành lập nhóm chuyên trách build:** Trên các dự án lớn, chăm sóc daily build và cập nhật smoke test có thể thành công việc toàn thời gian của nhiều người.
- **Chỉ bổ sung mã khi hợp lý:** Phần lớn lập trình viên không tạo ra increment có ý nghĩa mỗi ngày; nên làm việc với khối lượng mã đồng nhất rồi tích hợp khi phù hợp, nhưng không để quá lâu.
- **Không kiểm tra code quá chậm:** Nếu một lập trình viên quá lâu mới tích hợp mã thì giá trị của daily build sẽ bị giảm sút.
- **Bắt buộc lập trình viên kiểm thử smoke trước khi thêm mã:** Phát triển nên kiểm thử local build trước khi thêm vào hệ thống, có thể nhờ một "bạn kiểm thử" kiểm riêng phần mã đó.
- **Tạo khu vực chờ để thêm mã mới:** Cần biết build nào là "tốt", lập trình viên cần làm việc với hệ thống tốt đã kiểm thử. Thường dùng version-control system (hệ thống quản lý phiên bản) để quản lý, đánh dấu ngày của build "tốt" cuối cùng. Nếu không có công cụ này, có thể phải quản lý thủ công.
- **Áp dụng hình phạt khi làm lỗi build:** Nhiều nhóm áp dụng hình phạt nhẹ nhàng hoặc vui vẻ để nhấn mạnh tầm quan trọng của việc giữ build ổn định. Ví dụ, ai làm lỗi build phải

dán hình nộm lên cửa, đội mũ "dê" hoặc đóng góp vào quỹ đội nhóm. Một số nhóm áp dụng biện pháp nghiêm khắc hơn như buộc lập trình viên đeo máy nhắn tin và phải sửa lỗi ngay khi phát hiện build hỏng.
- **Phát hành build vào buổi sáng:** Nên hoàn thiện build và kiểm thử vào sáng sớm, phát hành ngay trong ngày. Như vậy, tester có thể làm việc với bản build mới và trao đổi, xử lý vấn đề với lập trình viên còn đang trực.
- **Giữ daily build kể cả khi bị áp lực deadline:** Khi áp lực tiến độ lớn, phát triển thường dễ bỏ qua discipline, nhưng chính lúc này daily build càng quan trọng để giữ dự án ổn định.

  Hạn chế: Một số nhà phát triển cho rằng không thể build mỗi ngày vì dự án quá lớn. Nhưng các dự án như Microsoft Windows 2000 (gồm hàng chục ngàn file, tới 50 triệu dòng code, hoàn thiện build trong 19 giờ liên tục trên nhiều máy) vẫn thực hiện được. Thực tế, dự án càng lớn càng phải chú trọng tích hợp và kiểm thử gia tăng.

Nghiên cứu ở các quốc gia khác nhau cho thấy chỉ 20–25% dự án áp dụng daily build vào giai đoạn đầu hoặc giữa dự án, đây là cơ hội lớn để cải thiện.

Tích hợp liên tục (Continuous Integration)

Một số chuyên gia phần mềm đã nâng cao khái niệm daily build thành tích hợp liên tục (continuous integration). Phần lớn tài liệu đề cập "continuous" nghĩa là "ít nhất mỗi ngày"; một số người hiểu "liên tục" theo nghĩa đen là tích hợp từng thay đổi chỉ sau vài giờ. Đối với đa số dự án, tích hợp liên tục tuyệt đối là không cần thiết. Tuy nhiên, kiểm thử và tích hợp thường xuyên, dù chỉ là mỗi ngày, vẫn mang lại hiệu quả rõ rệt.

[Lưu ý: Một số lỗi đánh máy, định dạng trong bản gốc đã được sửa/thống nhất để đảm bảo mạch lạc.]

Giá trị của việc tạm thời không đồng bộ mã nguồn

Trong các dự án có quy mô vừa và lớn, có giá trị nhất định khi để mã nguồn tạm thời không đồng bộ trong những khoảng thời gian ngắn. **Dữ liệu thực tế cho thấy** các lập trình viên thường xuyên bị mất đồng bộ khi thực hiện các thay đổi lớn về cấu trúc. Sau đó, họ có thể đồng bộ lại sau một thời gian ngắn. Việc xây dựng (build) hằng ngày cung cấp các điểm hẹn (rendezvous points) thường xuyên đủ cho nhóm dự án. Miễn là nhóm đồng bộ mỗi ngày, họ không cần phải liên tục cập nhật lẫn nhau.

Checklist: Chiến lược tích hợp (Integration Strategy)

- **Chiến lược tích hợp có xác định thứ tự tối ưu để tích hợp các subsystem (hệ thống con), class (lớp), và routine (thủ tục) không?**
- **Thứ tự tích hợp đã được phối hợp với thứ tự xây dựng (construction order) để đảm bảo các class sẵn sàng cho việc tích hợp vào đúng thời điểm chưa?**
- **Chiến lược có giúp dễ dàng chẩn đoán defect (khiếm khuyết/phát hiện lỗi) không?**
- **Chiến lược có hạn chế sử dụng scaffolding (mã bổ trợ tạm thời) đến mức tối thiểu không?**
- **Chiến lược được chọn có thực sự tốt hơn các phương án khác không?**

- **Các interface (giao diện) giữa các thành phần đã được xác định rõ ràng chưa? (Việc xác định interface không phải nhiệm vụ của tích hợp nhưng việc kiểm tra lại là cần thiết.)**

Checklist: Xây dựng và Kiểm thử nhanh (Daily Build and Smoke Test)

- **Dự án có được xây dựng thường xuyên—lý tưởng là hằng ngày—để hỗ trợ tích hợp tăng dần không?**
- **Mỗi lần build có thực hiện smoke test (kiểm thử tổng quát nhanh) để biết build có hoạt động không?**
- **Build và smoke test đã được tự động hóa chưa?**
- **Lập trình viên có thường xuyên check-in (nộp) mã nguồn—không quá một hoặc hai ngày giữa các lần check-in không?**
- **Smoke test có được cập nhật liên tục cùng với sự phát triển của mã nguồn không?**
- **Xảy ra build lỗi có phải là điều hiếm gặp không?**
- **Bạn có tiếp tục thực hiện build và smoke test phần mềm ngay cả khi chịu áp lực tiến độ không?**

Tài nguyên bổ sung (Additional Resources)

cc2e.com/2999
Sau đây là các tài nguyên liên quan đến chủ đề tích hợp và phát triển phần mềm trong chương này:

Tích hợp (Integration)

- **Lakos, John. Large-Scale C++ Software Design.** Boston, MA: Addison-Wesley, 1996.
  Lakos cho rằng "physical design" (thiết kế vật lý)—tức là cấu trúc phân cấp file, thư mục, và thư viện—ảnh hưởng đáng kể tới khả năng xây dựng phần mềm của nhóm phát triển. Nếu không chú ý tới thiết kế vật lý, thời gian build sẽ trở nên kéo dài, làm giảm hiệu quả tích hợp thường xuyên. Tuy luận điểm này tập trung vào C++, nhưng các quan điểm về "physical design" vẫn áp dụng cho dự án ở các ngôn ngữ khác.
- **Myers, Glenford J. The Art of Software Testing.** New York, NY: John Wiley & Sons, 1979.
  Cuốn sách kiểm thử kinh điển này bàn về tích hợp như một hoạt động kiểm thử.

Phát triển gia tăng (Incrementalism)

- **McConnell, Steve. Rapid Development.** Redmond, WA: Microsoft Press, 1996.
  Chương 7 ("Lifecycle Planning") trình bày chi tiết về những điểm đánh đổi giữa các mô hình vòng đời linh hoạt và không linh hoạt. Các chương 20, 21, 35, và 36 thảo luận các mô hình vòng đời hỗ trợ nhiều mức độ phát triển gia tăng khác nhau. Chương 19 mô tả về "designing for change" (thiết kế cho thay đổi), hoạt động quan trọng để hỗ trợ mô hình phát triển lặp và gia tăng.
- **Boehm, Barry W. "A Spiral Model of Software Development and Enhancement."** Computer, May 1988: 61–72.
  Bài báo này mô tả "spiral model" (mô hình xoắn ốc) của phát triển phần mềm, trình bày như một cách tiếp cận quản lý rủi ro trong dự án phát triển phần mềm. Nội dung chủ yếu về phát triển nói chung hơn là chỉ về tích hợp. Boehm là chuyên gia hàng đầu về các vấn đề

tổng thể của phát triển phần mềm, và cách giải thích của ông phản ánh chất lượng kiến thức sâu sắc.

- **Gilb, Tom. Principles of Software Engineering Management.** Wokingham, England: Addison-Wesley, 1988.
  Các chương 7 và 15 phân tích sâu về evolutionary delivery (phát triển tiến hóa)—một trong những cách tiếp cận phát triển gia tăng đầu tiên.
- **Beck, Kent. Extreme Programming Explained: Embrace Change.** Reading, MA: Addison-Wesley, 2000.
  Cuốn sách này trình bày hiện đại, súc tích, và cổ vũ mạnh mẽ nhiều ý tưởng trong sách của Gilb. Tác giả thích cuốn của Gilb vì phân tích sâu sắc, nhưng một số độc giả có thể thấy trình bày của Beck dễ tiếp cận hoặc phù hợp hơn với dự án của mình.

Các điểm chính

- **Trình tự xây dựng (construction sequence) và phương pháp tích hợp (integration approach) ảnh hưởng đến thứ tự thiết kế, lập trình và kiểm thử các class.**
- **Một trình tự tích hợp hợp lý giúp giảm công sức kiểm thử và làm việc gỡ lỗi dễ dàng hơn.**
- **Tích hợp tăng dần có nhiều biến thể; trừ những dự án rất đơn giản, bất kỳ biến thể nào cũng đều tốt hơn tích hợp theo pha (phased integration).**
- **Phương pháp tích hợp tối ưu cho từng dự án thường là sự kết hợp giữa top-down, bottom-up, risk-oriented, và các phương pháp khác. Tích hợp theo hình chữ T (T-shaped integration) và tích hợp theo lát cắt dọc (vertical-slice integration) là hai phương pháp thường mang lại hiệu quả.**
- **Build hằng ngày giúp giảm vấn đề tích hợp, nâng cao tinh thần lập trình viên và cung cấp các thông tin hữu ích cho quản lý dự án.**

Chương 30: Công cụ lập trình (Programming Tools)

cc2e.com/3084 Mục lục

Chủ đề liên quan:

- Công cụ quản lý version (version-control tools): xem Phần 28.2
- Công cụ kiểm thử lỗi (debugging tools): Phần 23.5
- Công cụ hỗ trợ kiểm thử (test-support tools): Phần 22.5

Modern programming tools (công cụ lập trình hiện đại) giúp giảm đáng kể thời gian xây dựng phần mềm. Việc sử dụng bộ công cụ tiên tiến—và thông thạo các công cụ được sử dụng—có thể

tăng năng suất lên 50% hoặc hơn (theo Jones 2000; Boehm và cộng sự 2000). Công cụ lập trình cũng làm giảm đáng kể công việc chi tiết, thủ công mà lập trình viên phải làm.

Một chú chó có thể là người bạn tốt nhất của con người, nhưng một vài công cụ tốt chính là những người bạn thân thiết nhất của lập trình viên. Như Barry Boehm từng phát hiện, 20% công cụ thường chiếm 80% lượng sử dụng (Boehm 1987b). Nếu bạn thiếu một trong những công cụ hữu ích, nghĩa là bạn đã bỏ lỡ điều gì đó mình có thể tận dụng thường xuyên.

Chương này tập trung vào hai hướng:

- Thứ nhất, chỉ đề cập đến các công cụ phục vụ quá trình xây dựng phần mềm. Các công cụ đặc tả yêu cầu (requirements-specification), quản lý và phát triển toàn diện (end-to-end-development tools) nằm ngoài phạm vi của sách này.
- Thứ hai, chương này nói về các loại công cụ thay vì nhấn mạnh thương hiệu cụ thể, bởi các sản phẩm và phiên bản thay đổi rất nhanh trong thực tế.

Một lập trình viên có thể làm việc nhiều năm mà chưa từng phát hiện ra một số công cụ cực kỳ hữu ích. Mục tiêu của chương này là khảo sát những công cụ sẵn có, giúp bạn nhận biết liệu mình có đang bỏ sót công cụ nào có thể giúp ích không. Nếu bạn đã là chuyên gia về công cụ, có thể chỉ cần lướt qua các phần đầu và đọc Phần 30.6 “Đảo công cụ thần kỳ”, rồi chuyển sang chương tiếp theo.

30.1 Công cụ thiết kế (Design Tools)

> **Tham khảo thêm về thiết kế, xem Chương 5 đến Chương 9.**

Công cụ thiết kế hiện nay chủ yếu là các công cụ đồ họa tạo sơ đồ thiết kế. Một số công cụ thiết kế được tích hợp trong CASE (Computer-Aided Software Engineering—kỹ thuật phần mềm hỗ trợ máy tính) với nhiều chức năng rộng hơn; cũng có nhà cung cấp quảng cáo công cụ thiết kế lẻ như CASE tools. Công cụ thiết kế đồ họa thường cho phép bạn diễn đạt thiết kế bằng các ký hiệu đồ họa phổ biến: UML, sơ đồ khối kiến trúc, cây phân cấp, sơ đồ quan hệ thực thể, hoặc sơ đồ class.

Có công cụ chỉ hỗ trợ một ký hiệu, một số khác hỗ trợ nhiều loại ký hiệu song song.

Nhìn chung, các công cụ thiết kế này giống như những bộ vẽ hình phức tạp. Bạn có thể dùng phần mềm vẽ cơ bản hoặc bút giấy để vẽ ra mọi thứ mà công cụ có thể vẽ. Tuy nhiên, các công cụ thiết kế mang lại những giá trị không thể có ở gói vẽ thông thường:

- Nếu bạn có sơ đồ bong bóng (bubble chart) và xóa một bong bóng, công cụ sẽ tự động sắp xếp lại các bong bóng khác cũng như các mũi tên và lớp dưới liên quan.
- Khi thêm bong bóng mới, công cụ cũng hỗ trợ các công việc “nhà nội trợ”.
- Bạn có thể dễ dàng chuyển đổi giữa các mức trừu tượng cao và thấp.
- Công cụ còn giúp kiểm tra tính nhất quán thiết kế, thậm chí một số còn có thể sinh mã từ thiết kế.

30.2 Công cụ cho mã nguồn (Source-Code Tools)

Các công cụ hỗ trợ làm việc với mã nguồn hiện đã phát triển phong phú và trưởng thành hơn so với các công cụ thiết kế.

Biên tập (Editing)

Nhóm công cụ này tập trung vào việc chỉnh sửa mã nguồn.

IDE (Integrated Development Environment—Môi trường phát triển tích hợp)

Một số lập trình viên ước tính họ dành tới 40% thời gian để chỉnh sửa mã nguồn (Parikh 1986, Ratliff 1987). Nếu đúng, đầu tư vào một IDE tốt là điều hợp lý.

Dữ liệu thực tế cho thấy ngoài các chức năng soạn thảo văn bản cơ bản, IDE tốt còn có:

- Biên dịch và phát hiện lỗi ngay trong trình soạn thảo
- Tích hợp với công cụ quản lý mã nguồn (source-code control), công cụ build, kiểm thử và debug
- Chế độ hiển thị nén hoặc dạng outline (chỉ hiển thị tên class hoặc cấu trúc logic mà không hiện nội dung, còn gọi là "folding")
- Nhảy đến định nghĩa class, routine, hoặc biến
- Nhảy đến mọi nơi class, routine, hoặc biến đó được sử dụng
- Định dạng riêng cho từng ngôn ngữ
- Trợ giúp tương tác về ngôn ngữ đang chỉnh sửa
- Kiểm tra cặp dấu ngoặc (brace matching)
- Template cho các cấu trúc lệnh phổ biến (ví dụ, IDE tự động hoàn thiện khung vòng lặp for sau khi lập trình viên nhập for)
- Thụt lề thông minh (smart indenting), dễ dàng điều chỉnh thụt lề khi điều kiện logic thay đổi
- Tự động chuyển đổi hoặc refactoring code
- Macro có thể lập trình bằng ngôn ngữ quen thuộc
- Danh sách chuỗi tìm kiếm thường dùng để tiết kiệm thời gian nhập lại
- Hỗ trợ Regular Expression (biểu thức chính quy) cho tìm kiếm-thay thế
- Tìm kiếm-thay thế trên nhiều file cùng lúc
- Chỉnh sửa nhiều file đồng thời
- So sánh từng dòng hai file đặt cạnh nhau (side-by-side diff)
- Đa cấp độ hoàn tác (multilevel undo)

Dù vậy, vẫn có các trình soạn thảo khá sơ khai đang được sử dụng, nhưng nhiều editor hiện nay đã hỗ trợ đầy đủ các tính năng trên.

Tìm kiếm và thay thế chuỗi ở nhiều file (Multiple-File String Searching and Replacing)

Nếu editor của bạn không hỗ trợ tìm-thay thế nhiều file, bạn vẫn có thể tìm các công cụ bổ trợ cho tác vụ này. Chúng hữu ích khi cần tìm tất cả nơi xuất hiện của một class hoặc routine nào đó, hoặc kiểm tra lỗi tương tự ở các file khác.

Bạn có thể tìm kiếm chuỗi chính xác, chuỗi tương tự (bỏ qua viết hoa/thường), hoặc bằng regular expression. Regular expression đặc biệt mạnh khi cần tìm mẫu chuỗi phức tạp, ví dụ: tìm mọi truy cập mảng chứa magic number (chữ số từ "0" tới "9").

Ví dụ với công cụ grep tìm magic number:

```
grep "\[ *[0-9]+ *\]" * cpp
```

Bạn có thể tinh chỉnh tiêu chí tìm kiếm cho phù hợp.

Các công cụ xử lý thay đổi chuỗi hàng loạt phổ biến: Perl, AWK, sed, ...

Công cụ so sánh khác biệt (Diff Tools)

Lập trình viên thường cần so sánh hai file. Khi bạn sửa lỗi và cần loại bỏ các thử nghiệm không thành công, công cụ so sánh file sẽ liệt kê các dòng khác biệt. Khi làm việc nhóm, cần xem đồng nghiệp thay đổi gì kể từ lần cuối mình chỉnh mã, Diff sẽ so sánh phiên bản hiện tại với phiên bản trước đó và hiển thị khác biệt.

Nếu gặp lỗi mới mà không nhớ gặp ở phiên bản trước, thay vì lo lắng về mất trí nhớ, bạn có thể dùng công cụ diff so sánh phiên bản mã nguồn, xác định chính xác thay đổi và tìm nguồn gốc vấn đề.

Chức năng này thường được tích hợp trong công cụ quản lý version (revision-control tools).

Công cụ gộp mã (Merge Tools)

Một số hệ thống quản lý version khóa file nguồn để mỗi lần chỉ một người chỉnh sửa. Phong cách khác cho phép nhiều người cùng lúc làm việc và thực hiện merge tại thời điểm check-in. Lúc này, công cụ gộp thay đổi là cực kỳ cần thiết; chúng thường tự động xử lý các merge đơn giản, còn merge phức tạp sẽ hỏi ý kiến lập trình viên.

Beautifier cho mã nguồn (Source-Code Beautifiers)

> **Tham khảo về bố cục và phong cách mã nguồn, xem Chương 31 "Layout and Style".**

Beautifier làm cho mã nguồn trông đồng đều, nổi bật tên class/routine, chuẩn hóa kiểu thụt lề, định dạng comment nhất quán, ... Một số công cụ đưa mỗi routine lên một trang web hay một trang in riêng, hoặc thậm chí định dạng lại mạnh mẽ hơn. Bạn có thể tùy chỉnh cách beautifier xử lý code.

Có hai loại beautifier: một loại chỉ xuất ra bản hiển thị đẹp hơn mà không đổi code gốc; loại còn lại thay đổi trực tiếp file nguồn (chuẩn hóa cách thụt lề, format parameter list...). Loại sau hữu ích khi xử lý lượng lớn legacy code (mã cũ), tự động định dạng lại cho khớp với chuẩn coding style.

Công cụ sinh tài liệu interface (Interface Documentation Tools)

Một số công cụ trích xuất tài liệu về interface lập trình viên từ mã nguồn, dựa trên các dấu hiệu như trường @tag trong code. Công cụ (như Javadoc) sẽ lấy phần văn bản được đánh dấu và định dạng đẹp để trình bày.

Template

Template giúp bạn xử lý nhanh các thao tác lặp lại và muốn thực hiện nhất quán. Ví dụ, nếu muốn mỗi routine đều có prolog comment tiêu chuẩn, bạn tạo đoạn sườn này dưới dạng template (file

hoặc macro bàn phím). Khi tạo routine mới, bạn chỉ việc chèn template vào. Template cũng được áp dụng cho class, file, hoặc các cấu trúc nhỏ như vòng lặp.

Khi làm việc nhóm, template là cách đơn giản duy trì phong cách coding/documentation nhất quán; chỉ cần chuẩn bị sẵn cho cả nhóm từ đầu, mọi người sẽ sử dụng vì nó tiện lợi.

Công cụ tham chiếu chéo (Cross-Reference Tools)

Công cụ này liệt kê các biến và routine cùng tất cả các vị trí đã sử dụng chúng—thường dưới dạng trang web.

Công cụ sinh cây kế thừa (Class Hierarchy Generators)

Sinh thông tin về cây inheritance, hữu ích trong debug hoặc phân tích cấu trúc, đóng gói chương trình thành các package/subsystem; hiện nay, nhiều IDE cũng tích hợp tính năng này.

Công cụ đánh giá chất lượng code (Analyzing Code Quality)

Kiểm tra cú pháp và ngữ nghĩa 'khó tính' (Picky Syntax and Semantics Checkers)

Các checker này bổ sung cho compiler, kiểm tra code kỹ lưỡng hơn so với compiler thông thường. Ví dụ như:

```
while ( i = 0 )
```

dù cú pháp hợp lệ, thường nên viết là

```
while ( i == 0 )
```

Switch nhầm giữa '=' và '==' là lỗi phổ biến. Công cụ Lint (C/C++) cảnh báo về biến chưa khởi tạo, biến không dùng, parameter truyền vào mà không gán giá trị, phép gán/làm việc nguy hiểm với con trỏ, logic đáng ngờ, code không thể truy cập,... Các ngôn ngữ khác cũng có công cụ tương tự.

Công cụ đo lường metrics (Metrics Reporters)

> **Tham khảo thêm về đo lường trong Phần 28.4 "Measurement"**

Công cụ metrics giúp báo cáo chất lượng code (phức tạp routine, đếm số dòng code, đếm comment, số lần sửa đổi, gắn defect với lập trình viên, ...). Công cụ này giúp xác định routine phức tạp để kiểm thử/kênh review/thiết kế lại kỹ hơn. Theo Jones (2000), metric analysis có thể tăng năng suất bảo trì lên 20%.

Công cụ chuyển đổi cấu trúc mã (Refactoring Source Code)

Refactorer

Xem thêm về refactoring ở Chương 24 "Refactoring"

Refactoring tool hỗ trợ chuyển đổi mã nhanh, dễ dàng và giảm lỗi; ví dụ, đổi tên class trên toàn codebase, trích xuất đoạn routine mới chỉ với thao tác chọn code. Đã có công cụ dành cho Java, Smalltalk và dần xuất hiện cho các ngôn ngữ khác.

Restructurer

Restructuring tools chuyển đổi mã spaghetti đầy goto thành code tốt hơn, có cấu trúc rõ ràng, thường giúp tăng 25–30% năng suất bảo trì (Jones 2000). Công cụ này dự đoán nhiều khi tự động chuyển đổi, đôi khi cần chỉnh sửa thủ công với trường hợp khó. Bạn cũng có thể dùng chúng để tham khảo khi muốn tự chuyển đổi code.

Code Translator

Chuyển mã từ ngôn ngữ này sang ngôn ngữ khác, hữu ích khi di chuyển codebase lớn sang môi trường mới. Tuy nhiên, nếu mã gốc không tốt, kết quả chuyển đổi cũng không khá hơn.

Công cụ quản lý phiên bản (Version Control)

Tham khảo về công cụ này và lợi ích ở Phần 28.2 “Software Code Changes”

Có thể xử lý việc phát sinh nhiều phiên bản phần mềm bằng công cụ quản lý version (version-control tools):

- **Source-code control** (quản lý mã nguồn)
- ... (các nội dung chi tiết tiếp theo của phần version control không nằm trong đoạn gốc)

Lưu ý: Nếu bạn cần dịch chi tiết các phần tiếp theo về "Version Control tools" hoặc các chủ đề cụ thể hơn, vui lòng đính kèm nội dung liên quan.

Từ điển Dữ liệu và Vai trò trong Quản lý Dự án Phần mềm

Trong nhiều trường hợp, *data dictionary* (từ điển dữ liệu) chủ yếu tập trung vào *database schema* (lược đồ cơ sở dữ liệu). Đối với các dự án lớn, từ điển dữ liệu cũng rất hữu ích trong việc theo dõi hàng trăm hoặc hàng nghìn định nghĩa *class* (lớp). Trong các dự án nhóm lớn, công cụ này giúp tránh các *naming clash* (xung đột tên gọi). Sự xung đột này có thể là xung đột cú pháp trực tiếp (cùng một tên được sử dụng hai lần) hoặc là xung đột tinh vi hơn, khi các tên khác nhau được dùng cho cùng một đối tượng hoặc cùng một tên được dùng cho các ý nghĩa khác nhau một cách tinh tế.

Đối với mỗi mục dữ liệu (bảng cơ sở dữ liệu hoặc class), từ điển dữ liệu chứa tên và mô tả mục đó. Ngoài ra, từ điển cũng có thể bao gồm ghi chú về cách mục này được sử dụng.

30.3 Công Cụ Xử Lý Mã Thực Thi (*Executable-Code Tools*)

Các công cụ làm việc với mã thực thi phong phú không kém các công cụ làm việc với *source code* (mã nguồn).

Tạo Mã (*Code Creation*)

Các công cụ trong mục này hỗ trợ quá trình tạo mã.

Biên Dịch và Liên Kết (*Compilers and Linkers*)

Compiler (trình biên dịch) chuyển đổi mã nguồn thành mã thực thi. Phần lớn chương trình được thiết kế để được biên dịch, tuy nhiên, vẫn còn một số chương trình được thực thi theo kiểu *interpreted* (diễn dịch).

Linker (trình liên kết) chuẩn sẽ liên kết một hoặc nhiều *object file* (tệp đối tượng), được trình biên dịch tạo ra từ file mã nguồn của bạn, với mã tiêu chuẩn cần thiết để tạo thành một chương trình thực thi. Linker thường có khả năng liên kết các file từ nhiều ngôn ngữ khác nhau, cho phép bạn lựa chọn ngôn ngữ phù hợp nhất cho từng phần của chương trình mà không phải bận tâm đến chi tiết tích hợp.

Overlay linker (trình liên kết phân lớp) giúp bạn phát triển những chương trình có thể thực thi trong bộ nhớ ít hơn tổng dung lượng mà chúng tiêu tốn. Overlay linker tạo ra một file thực thi chỉ nạp một phần của nó vào bộ nhớ trong từng thời điểm, phần còn lại sẽ được nạp từ ổ đĩa khi cần thiết.

Công Cụ Xây Dựng (*Build Tools*)

Mục đích của *build tool* (công cụ xây dựng) là giảm thiểu thời gian cần thiết để xây dựng một chương trình dựa trên các phiên bản mới nhất của file mã nguồn. Đối với mỗi *target file* (tệp mục tiêu) trong dự án, bạn xác định các file mã nguồn mà nó phụ thuộc và cách để tạo ra nó. Build tool còn giúp loại trừ lỗi do trạng thái không đồng nhất giữa các mã nguồn; build tool đảm bảo tất cả tệp đều được đồng bộ. Các công cụ xây dựng phổ biến bao gồm *make* (trên UNIX) và *ant* (cho các chương trình Java).

Ví dụ, giả định bạn có một target file tên là `userface.obj`. Trong file make, bạn chỉ ra rằng để tạo `userface.obj`, bạn cần biên dịch file `userface.cpp`. Bạn cũng chỉ rõ rằng `userface.cpp` phụ thuộc vào `userface.h`, `stdlib.h`, và `project.h`. Việc "phụ thuộc vào" có nghĩa là nếu bất kỳ file header nào thay đổi, `userface.cpp` cần được biên dịch lại.

Khi xây dựng chương trình, công cụ make kiểm tra tất cả các phụ thuộc bạn đã mô tả và xác định các file cần biên dịch lại. Nếu có năm trong số 250 file nguồn phụ thuộc vào định nghĩa dữ liệu trong `userface.h` và file này được cập nhật, make sẽ tự động biên dịch lại 5 file đó, không biên dịch lại 245 file còn lại không liên quan. Sử dụng make hay ant giúp giảm thiểu rủi ro so với việc biên dịch lại toàn bộ 250 file hay biên dịch từng file thủ công và quên mất file nào đó, gây ra lỗi không đồng bộ kỳ lạ. Tổng thể, build tool như make hoặc ant giúp rút ngắn thời gian và tăng độ tin cậy của chu trình biên dịch-liên kết-thực thi.

Một số nhóm phát triển tìm ra các hướng tiếp cận khác thay thế cho công cụ kiểm tra phụ thuộc như make. Ví dụ, nhóm phát triển Microsoft Word nhận thấy việc biên dịch lại toàn bộ các file nguồn nhanh hơn so với việc kiểm tra phụ thuộc phức tạp bằng make, với điều kiện các file

nguồn đã được tối ưu hóa. Nhờ vậy, máy của từng lập trình viên có thể xây dựng lại toàn bộ chương trình Word—với hàng triệu dòng mã—chỉ trong khoảng 13 phút.

Thư Viện Mã (*Code Libraries*)

Cách hiệu quả để viết mã chất lượng cao trong thời gian ngắn là tìm kiếm hoặc mua sẵn mã nguồn mở phù hợp thay vì tự viết toàn bộ. Một số lĩnh vực thư viện đáng chú ý gồm:

- *Container classes* (lớp chứa dữ liệu)
- Dịch vụ giao dịch thẻ tín dụng (dịch vụ thương mại điện tử)
- Công cụ phát triển đa nền tảng
- Công cụ nén dữ liệu
- Kiểu dữ liệu và thuật toán
- Công cụ thao tác cơ sở dữ liệu và tệp dữ liệu
- Công cụ lập biểu đồ, vẽ sơ đồ, và biểu diễn dữ liệu
- Công cụ xử lý ảnh
- Quản lý bản quyền
- Các phép toán toán học
- Công cụ kết nối mạng và giao tiếp Internet
- Trình tạo báo cáo và xây dựng truy vấn báo cáo
- Công cụ bảo mật và mã hóa
- Công cụ bảng tính và lưới dữ liệu
- Công cụ xử lý văn bản, kiểm tra chính tả
- Công cụ thoại, điện thoại và fax

Trình Tạo Mã Tự Động (*Code-Generation Wizards*)

Nếu bạn không tìm được mã như ý, có thể sử dụng các công cụ sinh mã tự động. Các công cụ này —thường tích hợp trong *IDE* (Integrated Development Environment – môi trường phát triển tích hợp)—giúp sinh mã thay cho bạn.

Các công cụ sinh mã chủ yếu hướng đến ứng dụng cơ sở dữ liệu, nhưng phạm vi bao phủ rất rộng. Thông thường, code generator sẽ tạo mã cho database, user interface, hoặc thậm chí là compiler. Mã do các công cụ sinh ra hiếm khi đạt được chất lượng như lập trình viên viết bằng tay, nhưng với nhiều ứng dụng, chất lượng như vậy là đủ chấp nhận được và giá trị thực tế cao hơn khi có nhiều ứng dụng hoạt động được thay vì chỉ có một phần mềm hoàn hảo.

Code generator cũng rất hữu ích để tạo nhanh *prototype* (mẫu thử nghiệm) cho mã sản xuất. Bạn có thể tạo ra bản mẫu trong vài giờ để trình diễn các tính năng giao diện người dùng hoặc thử nghiệm các ý tưởng thiết kế khác nhau. Nếu tự mã hóa bằng tay, có thể mất cả tuần để hoàn thiện tính năng tương tự. Khi chỉ cần thử nghiệm, nên chọn cách nhanh chóng và tiết kiệm chi phí.

Nhược điểm chung của công cụ sinh mã là thường sinh ra mã gần như không thể đọc được. Nếu bạn phải bảo trì loại mã này, có thể sẽ hối tiếc vì không tự viết ngay từ đầu.

Cài Đặt và Thiết Lập (*Setup and Installation*)

Nhiều nhà cung cấp cung cấp công cụ tạo chương trình setup. Các công cụ này hỗ trợ tạo bản cài đặt trên đĩa, CD, DVD hoặc qua mạng, kiểm tra sự tồn tại của các thư viện chung, kiểm tra phiên bản, v.v.

Tiền Xử Lý (*Preprocessors*)

Preprocessor (trình tiền xử lý) và các macro tiền xử lý hữu ích cho việc debug vì chúng giúp chuyển đổi nhanh giữa mã phát triển và mã sản xuất. Trong quá trình phát triển, nếu muốn kiểm tra tình trạng phân mảnh bộ nhớ ở đầu mỗi routine, bạn chỉ cần sử dụng macro ở đầu routine. Khi chuyển sang mã sản xuất, bạn có thể định nghĩa lại macro để không sinh ra mã nào cả. Macro tiền xử lý cũng hữu dụng cho việc viết mã hướng đến nhiều môi trường biên dịch khác nhau (ví dụ Windows và Linux).

Nếu sử dụng ngôn ngữ có cấu trúc điều khiển sơ khai như assembler, bạn có thể viết preprocessor điều khiển luồng nhằm mô phỏng các cấu trúc như if-then-else và while.

Nếu ngôn ngữ không có preprocessor, bạn vẫn có thể thêm vào quy trình build một preprocessor độc lập, như M4 (có tại www.gnu.org/software/m4/).

Gỡ Lỗi (*Debugging*)

Các công cụ sau đây hỗ trợ quá trình gỡ lỗi:

- Cảnh báo từ *compiler* (trình biên dịch)
- *Test scaffolding* (giàn thử nghiệm)
- *Diff tool* (so sánh các phiên bản mã nguồn)
- *Execution profiler* (phân tích thực thi)
- *Trace monitor* (theo dõi lược đồ)
- *Interactive debugger* (gỡ lỗi tương tác)—cả phần mềm và phần cứng

Các công cụ kiểm thử (testing tools) cũng có liên quan chặt chẽ đến công cụ gỡ lỗi.

Kiểm Thử (*Testing*)

Các công cụ và tính năng sau hỗ trợ kiểm thử hiệu quả:

- Khung kiểm thử tự động như JUnit, NUnit, CppUnit,...
- Công cụ sinh kiểm thử tự động
- Trình ghi và phát lại test case
- Công cụ theo dõi độ bao phủ (logic analyzer, execution profiler)
- Trình gỡ lỗi biểu tượng (*symbolic debugger*)
- Hệ thống gây nhiễu (*system perturber*)—bổ sung, lắc, làm hỏng hoặc kiểm tra truy cập bộ nhớ
- Công cụ so sánh dữ liệu (*diff tool*)
- Bộ giàn thử (scaffolding)
- Công cụ chèn lỗi có chủ đích (*defect-injection tool*)
- Phần mềm theo dõi lỗi (*defect-tracking software*)

Tinh Chỉnh Mã (*Code Tuning*)

Các công cụ sau giúp bạn tối ưu hiệu năng mã nguồn.

Execution Profiler

Execution profiler (bộ phân tích thực thi) giám sát mã của bạn khi chạy và báo cáo số lần mỗi câu lệnh được thực hiện hoặc lượng thời gian chương trình dành cho một câu lệnh hay nhánh thực thi nào đó. Quá trình này giống như bác sĩ dùng ống nghe kiểm tra tim phổi—nó cung cấp góc nhìn sâu giúp bạn xác định các điểm "nóng" và nên tập trung tinh chỉnh ở đâu.

Assembler Listing và Disassembler

Đôi khi bạn cần xem mã assembler do trình biên dịch ngôn ngữ bậc cao sinh ra. Một số trình biên dịch sinh ra file liệt kê assembler, số khác thì không và bạn cần dùng *disassembler* (phần mềm dịch ngược) để khôi phục mã assembler từ mã máy do compiler xuất ra. Xem mã assembler giúp đánh giá độ hiệu quả của trình biên dịch khi chuyển mã cấp cao thành mã máy, đồng thời lý giải tại sao mã cấp cao nhìn có vẻ tối ưu lại thực thi chậm. Trong Chương 26—"Các kỹ thuật tinh chỉnh mã"—nhiều kết quả benchmark khá bất ngờ, và tôi đã thường xuyên phải tra cứu mã assembler để hiểu kỹ nguyên nhân.

Nếu bạn chưa quen với assembler, việc so sánh từng câu lệnh cấp cao với mã assembler do compiler sinh ra là cách tiếp cận tuyệt vời. Lần đầu tiếp xúc thường mang lại cảm giác "vỡ mộng": bạn sẽ thấy compiler sinh ra quá nhiều mã so với nhu cầu, và sẽ không còn nhìn compiler một cách ngây thơ như trước nữa.

Ngược lại, trong một số môi trường, compiler buộc phải sinh ra mã phức tạp, và việc nghiên cứu sản phẩm đầu ra có thể giúp bạn trân trọng công sức phải bỏ ra nếu lập trình ở ngôn ngữ bậc thấp hơn.

30.4 Môi Trường Hướng Công Cụ (*Tool-Oriented Environments*)

Một số môi trường đặc biệt phù hợp với lập trình hướng công cụ. Môi trường UNIX nổi tiếng với bộ công cụ nhỏ, tên lạ nhưng hoạt động rất hiệu quả cùng nhau: grep, diff, sort, make, crypt, tar, lint, ctags, sed, awk, vi, v.v. Ngôn ngữ C và C++ (gắn bó mật thiết với UNIX) cũng mang triết lý tương tự—thư viện tiêu chuẩn C++ được tạo thành từ các hàm nhỏ dễ dàng kết hợp thành những hàm lớn hơn nhờ khả năng tương thích tốt.

Một số lập trình viên quá quen thuộc và làm việc hiệu quả tại UNIX đến mức họ mang thói quen đó sang các môi trường khác, sử dụng các công cụ "nhái" UNIX cho Windows cũng như hệ điều hành khác. Ví dụ, cygwin cung cấp các công cụ tương đương UNIX trên Windows (www.cygwin.com).

Eric Raymond, trong tác phẩm *The Art of Unix Programming* (2004), cung cấp một góc nhìn sâu sắc về văn hóa lập trình UNIX.

30.5 Tự Xây Dựng Công Cụ Lập Trình (*Building Your Own Programming Tools*)

Giả sử bạn được giao một công việc cần 5 tiếng để hoàn thành và có hai lựa chọn:

- Làm trực tiếp trong 5 tiếng, hoặc
- Dành 4 tiếng 45 phút xây dựng công cụ, sau đó chỉ mất 15 phút để hoàn thành công việc bằng công cụ đó

Phần lớn các lập trình viên giỏi sẽ chọn phương án thứ hai trong đa số trường hợp. Việc xây dựng công cụ là phần không thể thiếu trong nghề lập trình. Gần như tất cả các tổ chức lớn (trên 1000 lập trình viên) đều có nhóm phát triển công cụ nội bộ. Nhiều đơn vị còn sở hữu các công cụ đặc thù, vượt trội hơn cả những sản phẩm thương mại (Jones 2000).

Bạn có thể tự xây dựng nhiều công cụ như mô tả ở chương này, dù có thể thiếu tính kinh tế, nhưng không hề có rào cản kỹ thuật lớn nào.

Công Cụ Đặc Thù Cho Dự Án (*Project-Specific Tools*)

Hầu hết dự án vừa và lớn cần các công cụ chuyên biệt riêng cho dự án đó. Ví dụ, bạn có thể cần công cụ sinh dữ liệu kiểm thử đặc biệt, kiểm tra chất lượng tệp dữ liệu, hoặc mô phỏng phần cứng chưa sẵn có. Dưới đây là vài ví dụ:

- Một nhóm hàng không phụ trách phát triển phần mềm điều khiển cảm biến hồng ngoại và phân tích dữ liệu thu nhận. Để xác thực hiệu năng phần mềm, máy ghi dữ liệu trên chuyến bay sẽ lưu lại hành động của phần mềm, sau đó các kỹ sư dùng công cụ phân tích dữ liệu tự viết để đánh giá hệ thống.
- Microsoft từng dự kiến tích hợp công nghệ font mới vào Windows. Do cả dữ liệu và phần mềm đều mới, lỗi có thể phát sinh từ cả hai phía. Các nhà phát triển đã tự xây dựng các công cụ kiểm tra lỗi trong file dữ liệu font, giúp phân biệt lỗi dữ liệu và lỗi phần mềm.
- Một công ty bảo hiểm phát triển hệ thống tính toán mức tăng phí phức tạp, đòi hỏi kiểm tra kỹ hàng trăm mức phí được tính toán. Việc kiểm tra thủ công từng trường hợp mất nhiều phút, do đó công ty đã làm một công cụ phần mềm tính toán riêng biệt, giúp kiểm tra nhanh từng trường hợp và tiết kiệm thời gian so với kiểm tra thủ công từ chương trình chính.

Phần lập kế hoạch dự án cần tính đến các dạng công cụ này và lên kế hoạch thời gian phát triển phù hợp.

Script

Script là công cụ tự động hóa các thao tác lặp lại. Trong một số hệ thống, script còn được gọi là batch file hoặc macro. Script có thể đơn giản hoặc phức tạp, và nhiều script hữu ích nhất lại dễ viết. Ví dụ, để bảo vệ quyền riêng tư, tôi mã hóa nhật ký (journal) cá nhân trừ khi đang viết. Tôi có một script dùng để giải mã, mở trình xử lý văn bản, và sau khi hoàn thành thì mã hóa lại:

```
crypto c:\word\journal * %1 /d /Es /s
word c:\word\journal doc
crypto c:\word\journal * %1 /Es /s
```

`%1` là vị trí điền mật khẩu (không nên lưu trực tiếp trong script). Script này giúp tôi không phải gõ lại các tham số dài dòng, giảm thiểu lỗi nhập sai và đảm bảo luôn thực hiện đúng chuỗi thao tác.

Nếu bạn thấy mình gõ lặp lại một chuỗi dài hơn 5 ký tự nhiều lần mỗi ngày, đó là ứng viên tốt cho một script hoặc batch file—chẳng hạn chuỗi lệnh biên dịch, backup, hay bất kỳ câu lệnh có tham số phức tạp nào.

30.6 Miền Mơ Mộng về Công Cụ (*Tool Fantasyland*)

Trong nhiều thập kỷ, các nhà cung cấp và chuyên gia trong ngành đã hứa hẹn về các công cụ đủ mạnh để loại bỏ hoàn toàn lập trình viên. Công cụ đầu tiên nhận được kỳ vọng này là Fortran—*Formula Translation Language*—với ý tưởng giúp nhà khoa học và kỹ sư chỉ cần gõ công thức là lập trình xong, không cần lập trình viên.

Thực tế là Fortran đã giúp các nhà khoa học, kỹ sư viết chương trình, nhưng từ góc nhìn hiện nay, Fortran chỉ là một ngôn ngữ lập trình bậc thấp hơn so với các kỳ vọng. Nó không loại bỏ lập trình viên, và câu chuyện này lặp lại trong suốt tiến bộ của ngành phần mềm.

Ngành phần mềm liên tục phát triển các công cụ mới nhằm giảm thiểu hoặc loại bỏ những khía cạnh tẻ nhạt của quy trình lập trình—từ việc định dạng mã, chỉnh sửa, biên dịch, liên kết, phát hiện lỗi cú pháp cho đến tạo giao diện và tiện ích tiêu chuẩn. Mỗi khi công cụ mới xuất hiện và chứng minh hiệu năng, các chuyên gia thường kỳ vọng rằng những cải tiến này sẽ "loại bỏ lập trình viên". Nhưng thực tế là mỗi bước tiến mới lại đi kèm những hạn chế nhất định. Sau một thời gian, các hạn chế dần được khắc phục, nhưng sản phẩm cuối cùng chỉ thực sự tối ưu khi mọi khó khăn phụ thêm phát sinh từ sự đổi mới công cụ đều được loại bỏ. Nhìn chung, việc loại bỏ được "khó khăn phụ" không làm tăng năng suất lập trình, mà chỉ bù lại những hạn chế từng mang lại.

Trong vài thập kỷ qua, đã xuất hiện các công cụ được kỳ vọng thay thế lập trình viên: từ *third-generation language* (ngôn ngữ thế hệ ba), *fourth-generation language*, *automatic programming*, *CASE tool*, cho đến *visual programming*. Mỗi tiến bộ này đều giúp tăng năng suất từng phần, và xét tổng thể, ngành lập trình hiện đại rất khác so với giai đoạn trước đây. Tuy vậy, không tiến bộ nào loại bỏ được nhu cầu về lập trình viên.

Nguyên nhân là bản chất của lập trình luôn phức tạp, dù có sự hỗ trợ tốt từ công cụ. Lập trình viên mãi phải đối mặt với thế giới thực nhiều rối rắm: phải tư duy logic về chuỗi thao tác, phụ thuộc và ngoại lệ; làm việc với người dùng cuối không nhất quán; giải quyết các giao diện không rõ ràng với phần mềm hoặc phần cứng khác; cũng như phải tính đến quy định, nguyên tắc kinh doanh và những phức tạp xuất phát từ bên ngoài lĩnh vực phần mềm.

Chúng ta luôn cần những người biết kết nối giữa bài toán hiện thực và cỗ máy tính toán. Những người này vẫn sẽ được gọi là "lập trình viên", dù họ thao tác với thanh ghi máy trong assembler hoặc kéo-thả trong Microsoft Visual Basic. Miễn là còn máy tính, chúng ta vẫn cần người cho máy biết phải làm gì, và công việc ấy vẫn gọi là lập trình.

Khi nghe nhà cung cấp quảng bá: "Công cụ mới này sẽ loại bỏ lập trình," hãy cảnh giác—hoặc ít nhất mỉm cười trước sự lạc quan ngây thơ đó.

Tài Nguyên Tham Khảo

Một số nguồn tài liệu hữu ích về công cụ lập trình:

- Software Development Magazine's annual Jolt Productivity award website là nguồn cập nhật các công cụ lập trình tốt nhất hiện tại.
- Hunt, Andrew; David Thomas. *The Pragmatic Programmer*. Boston, MA: Addison-Wesley, 2000.

Bản dịch đã được kiểm tra, đảm bảo thuật ngữ thống nhất, lược bỏ các lỗi định dạng, chú thích rõ ràng các thuật ngữ chuyên ngành và trình bày mạch lạc, ngắn gọn.

Tài liệu Tham khảo về Công cụ Lập trình

Các bài báo và sách đề xuất

- **Vaughn-Nichols, Steven**. “Building Better Software with Better Tools,” *IEEE Computer*, Tháng 9 năm 2003, trang 12–14.

    Bài báo này khảo sát các sáng kiến về công cụ do IBM, Microsoft Research và Sun Research dẫn dắt.

- **Glass, Robert L.** *Software Conflict: Essays on the Art and Science of Software Engineering*. Englewood Cliffs, NJ: Yourdon Press, 1991.

    Chương có tiêu đề "*Recommended: A Minimum Standard Software Toolset*" (Khuyến nghị: Bộ công cụ phần mềm tiêu chuẩn tối thiểu) đưa ra một góc nhìn phản biện so với quan điểm "càng nhiều công cụ càng tốt". Glass lập luận việc xác định một bộ công cụ tối thiểu cần thiết cho tất cả các nhà phát triển, đồng thời đề xuất một bộ công cụ khởi đầu.

- **Jones, Capers**. *Estimating Software Costs*. New York, NY: McGraw-Hill, 1998.

- **Boehm, Barry, et al.** *Software Cost Estimation with Cocomo II*. Reading, MA: Addison-Wesley, 2000.

    Cả hai cuốn sách của Jones và Boehm đều dành các phần để thảo luận về tác động của việc sử dụng công cụ đối với năng suất phát triển phần mềm.

---

cc2e com/3019 Danh mục kiểm tra: Công cụ Lập trình

- ❑ Bạn có đang sử dụng một IDE (Integrated Development Environment - Môi trường Phát triển Tích hợp) hiệu quả không?
- ❑ IDE của bạn có hỗ trợ tích hợp với quản lý mã nguồn (source-code control); các công cụ xây dựng (build), kiểm thử (test) và gỡ lỗi (debugging), cũng như các chức năng hữu ích khác không?
- ❑ Bạn có sử dụng các công cụ tự động hóa các thao tác *refactoring* (tái cấu trúc mã lệnh) phổ biến không?
- ❑ Bạn có sử dụng công cụ kiểm soát phiên bản (version control) để quản lý mã nguồn, nội dung, yêu cầu, thiết kế, kế hoạch dự án và các *artifact* (tạo phẩm) khác của dự án không?
- ❑ Nếu bạn đang làm việc trên một dự án quy mô rất lớn, bạn có sử dụng từ điển dữ liệu (data dictionary) hoặc kho lưu trữ trung tâm (central repository) nào chứa mô tả chuẩn xác cho từng lớp (class) trong hệ thống không?
- ❑ Bạn đã cân nhắc sử dụng thư viện mã nguồn (code libraries) để thay thế việc viết mã tùy chỉnh ở những nơi thích hợp chưa?

Lưu ý: Các thuật ngữ chuyên ngành được giữ nguyên kèm chú thích để đảm bảo tính nhất quán và dễ tra cứu.