

# Chương 20: Tổng Quan Về Chất Lượng Phần Mềm

Nguồn: cc2e.com/2036

## Nội dung

- 20.1 Đặc tính của Chất lượng Phần mềm: trang 463
- 20.2 Các kỹ thuật Cải thiện Chất lượng Phần mềm: trang 466
- 20.3 Hiệu quả Tương đối của các Kỹ thuật Chất lượng: trang 469
- 20.4 Thời điểm Thực hiện Đảm bảo Chất lượng: trang 473
- 20.5 Nguyên lý Chung về Chất lượng Phần mềm: trang 474

## Chủ đề liên quan

- Xây dựng phần mềm hợp tác: Chương 21
- Testing của lập trình viên (Developer testing): Chương 22
- Gỡ lỗi (Debugging): Chương 23
- Điều kiện tiên quyết để xây dựng: Chương 3 và 4
- Điều kiện tiên quyết có áp dụng cho các dự án phần mềm hiện đại không?: Phần 3.1

Chương này khảo sát các kỹ thuật chất lượng phần mềm từ quan điểm của giai đoạn xây dựng phần mềm. Toàn bộ cuốn sách tập trung vào việc nâng cao chất lượng phần mềm, tuy nhiên chương này nhấn mạnh vào hai khía cạnh "chất lượng" và "đảm bảo chất lượng" (quality assurance) tự thân. Cụ thể, chương này đề cập nhiều hơn đến các vấn đề tổng quát (big-picture issues) hơn là các kỹ thuật thực tiễn. Nếu bạn đang tìm kiếm lời khuyên thực tế về phát triển hợp tác, testing hay debugging, hãy tiếp tục với ba chương tiếp theo.

## 20.1 Đặc Tính của Chất Lượng Phần Mềm

Phần mềm bao gồm cả các đặc tính chất lượng bên ngoài (external) và bên trong (internal).

### Đặc tính bên ngoài

Đây là các đặc tính mà người dùng sản phẩm phần mềm nhận biết được, bao gồm:

- **Correctness** (Tính đúng đắn): Mức độ mà hệ thống không có lỗi trong đặc tả, thiết kế và triển khai.
- **Usability** (Tính dễ sử dụng): Mức độ dễ dàng cho người dùng học và sử dụng hệ thống.
- **Efficiency** (Hiệu suất): Sử dụng tối thiểu tài nguyên hệ thống, bao gồm bộ nhớ và thời gian thực thi.
- **Reliability** (Độ tin cậy): Khả năng của hệ thống thực hiện đúng chức năng trong các điều kiện quy định bất cứ khi nào cần – tức là có thời gian trung bình giữa các lỗi dài.
- **Integrity** (Tính toàn vẹn): Mức độ mà hệ thống ngăn chặn được truy cập trái phép hoặc không đúng đắn vào chương trình và dữ liệu của nó; bao gồm cả việc ràng buộc truy cập và đảm bảo dữ liệu chỉ được truy cập hoặc chỉnh sửa một cách hợp lệ.
- **Adaptability** (Khả năng thích nghi): Mức độ mà hệ thống có thể được sử dụng (không cần chỉnh sửa) trong các ứng dụng hoặc môi trường khác ngoài môi trường được thiết kế ban đầu.
- **Accuracy** (Độ chính xác): Hệ thống, như được xây dựng, không có lỗi, đặc biệt trong các đầu ra định lượng – khác với correctness ở chỗ accuracy đánh giá mức độ hệ thống hoàn thành nhiệm vụ so với mục tiêu thay vì chỉ đúng theo đặc tả.
- **Robustness** (Tính vững bền): Khả năng hệ thống tiếp tục hoạt động dưới điều kiện đầu vào không hợp lệ hoặc môi trường áp lực.

Một số đặc tính trên có thể trùng lặp một phần về ý nghĩa, tuy nhiên mỗi đặc tính lại đóng vai trò riêng biệt trong từng bối cảnh nhất định.

**Lưu ý:** Đối với người dùng, chỉ các đặc tính bên ngoài là quan trọng. Người dùng quan tâm đến việc phần mềm dễ sử dụng hay hoạt động đúng, không phải việc phần mềm có dễ bảo trì hay mã nguồn dễ đọc.

### Đặc tính bên trong

Lập trình viên quan tâm đến các đặc tính bên trong bên cạnh các đặc tính bên ngoài. Cuốn sách này tập trung vào các đặc tính bên trong sau:

- **Maintainability** (Khả năng bảo trì): Dễ dàng sửa đổi hệ thống để thay đổi hoặc bổ sung chức năng, cải thiện hiệu năng hoặc sửa lỗi.

- **Flexibility** (Tính linh hoạt): Mức độ dễ dàng sửa đổi hệ thống cho các mục đích hoặc môi trường khác với mục đích thiết kế ban đầu.
- **Portability** (Khả năng di động): Mức độ dễ dàng sửa đổi hệ thống để chạy trên môi trường khác với môi trường thiết kế.
- **Reusability** (Khả năng tái sử dụng): Mức độ và độ dễ dàng sử dụng các phần của hệ thống trong các hệ thống khác.
- **Readability** (Dễ đọc): Mức độ dễ đọc và hiểu mã nguồn, đặc biệt ở cấp độ từng câu lệnh chi tiết.
- **Testability** (Dễ kiểm thử): Mức độ dễ dàng cho việc kiểm thử từng đơn vị (unit-test) hoặc kiểm thử toàn hệ thống (system-test); cũng như mức độ xác minh hệ thống đáp ứng yêu cầu đề ra.
- **Understandability** (Dễ hiểu): Mức độ dễ hiểu hệ thống ở cả cấp độ tổng thể lẫn chi tiết, tập trung vào sự mạch lạc của hệ thống chung thay vì chỉ chú trọng tới độ dễ đọc từng câu lệnh.

Tương tự như các đặc tính bên ngoài, các đặc tính bên trong này cũng có thể trùng lặp chức năng ở một số điểm, nhưng đều quan trọng và có ý nghĩa riêng biệt.

Chủ đề chính của cuốn sách là các khía cạnh chất lượng nội tại của hệ thống, nên sẽ không bàn thêm chi tiết về chúng trong chương này.

## Mối quan hệ giữa các đặc tính

Sự phân biệt giữa các đặc tính bên trong và bên ngoài không hẳn hoàn toàn rõ ràng, vì ở một mức độ nào đó, chúng ảnh hưởng lẫn nhau. Phần mềm không dễ hiểu (understandable) hoặc bảo trì (maintainable) sẽ cản trở khả năng sửa lỗi, điều này tác động tiêu cực đến correctness và reliability. Hoặc phần mềm không linh hoạt (flexible) sẽ không thể được nâng cấp theo yêu cầu người dùng, từ đó ảnh hưởng đến usability.

Nỗ lực tối đa hóa một vài đặc tính này thường gây xung đột với việc tối đa hóa đặc tính khác. Tìm kiếm giải pháp tối ưu giữa các mục tiêu cạnh tranh là một hoạt động quan trọng, khiến phát triển phần mềm trở thành một ngành kỹ thuật thực thụ.

Hình 20-1 minh họa cách tập trung vào một đặc tính nào đó có thể tác động tích cực, tiêu cực hoặc trung lập đến các đặc tính khác.

Điều quan trọng là cần nhận thức rõ các mục tiêu chất lượng cụ thể của dự án cũng như xác định mối quan hệ giữa từng cặp mục tiêu, xem chúng mang tính hỗ trợ lẫn nhau hay đối lập.

## 20.2 Các Kỹ Thuật Cải Thiện Chất Lượng Phần Mềm

### Đảm bảo chất lượng phần mềm

**Software quality assurance (đảm bảo chất lượng phần mềm)** là một chương trình có kế hoạch và hệ thống các hoạt động nhằm đảm bảo một hệ thống có các đặc tính mong muốn. Ngoài việc tập trung nâng cao chất lượng sản phẩm, quá trình xây dựng phần mềm (software-development process) cũng cần được chú trọng.

Các thành phần của một chương trình chất lượng phần mềm bao gồm:

#### Mục tiêu chất lượng phần mềm

Một kỹ thuật mạnh mẽ giúp cải thiện chất lượng phần mềm là đặt ra **mục tiêu chất lượng rõ ràng** (quality objectives) dựa trên các đặc tính bên ngoài và bên trong đã liệt kê ở trên. Nếu không có mục tiêu cụ thể, lập trình viên sẽ tối ưu các thuộc tính khác với mong mỏi của quản lý.

#### Các hoạt động đảm bảo chất lượng rõ ràng

Một vấn đề phổ biến là chất lượng thường bị xem như mục tiêu phụ. Ở một số tổ chức, lập trình kiểu "làm nhanh, bất chấp" được khuyến khích và được thưởng nhiều hơn cho những ai hoàn thành nhanh, bất kể chất lượng. Tổ chức cần thể hiện chất lượng là ưu tiên hàng đầu bằng cách **chính thức hóa các hoạt động đảm bảo chất lượng** (explicit quality-assurance activity) để lập trình viên nhận thức rõ tầm quan trọng này.

#### Chiến lược kiểm thử (Test strategy)

Đối với kiểm tra chất lượng, các nhóm phát triển thường dựa vào việc xây dựng chiến lược kiểm thử kết hợp với yêu cầu sản phẩm, kiến trúc và thiết kế. Nhiều dự án dựa quá nhiều vào kiểm thử như cách thức chính để đánh giá và cải thiện chất lượng, tuy nhiên như phân sau chỉ ra, điều này là gánh nặng quá lớn nếu chỉ dựa vào testing.

#### Hướng dẫn kỹ thuật phần mềm (Software-engineering guidelines)

Các guideline này kiểm soát các khía cạnh kỹ thuật của phần mềm xuyên suốt mọi hoạt động phát triển: định nghĩa vấn đề, xây dựng yêu cầu, lên kiến trúc, coding và kiểm thử hệ thống.

## Đánh giá kỹ thuật không chính thức (Informal technical reviews)

Thường bao gồm việc tự kiểm tra thiết kế hoặc mã nguồn (desk-checking), hoặc thảo luận code với một vài đồng nghiệp (code walk-through).

## Đánh giá kỹ thuật chính thức (Formal technical reviews)

Hoạt động này nhằm phát hiện vấn đề ở giai đoạn thấp nhất trong vòng đời dự án, tức là khi chi phí sửa lỗi còn thấp nhất. Để đạt được điều này, cần áp dụng **quality gates** – các điểm kiểm tra chất lượng định kỳ để xác định chất lượng đã đủ tốt để chuyển sang giai đoạn tiếp theo chưa. "Cổng" này có thể là rà soát, inspection, review hoặc audit.

**Lưu ý:** Một "gate" không có nghĩa là kiến trúc hoặc yêu cầu phải hoàn thành 100% hoặc bất biến, mà là để xác định mức độ hoàn chỉnh đã đủ tốt đáp ứng nhu cầu của giai đoạn phát triển tiếp theo chưa.

## Đánh giá từ bên ngoài (External audits)

Là một loại đánh giá kỹ thuật cụ thể được thực hiện bởi nhóm độc lập từ bên ngoài tổ chức để xác định trạng thái của dự án hoặc chất lượng sản phẩm.

## Quy trình phát triển (Development process)

Các chiến lược và quy trình chất lượng lồng ghép vào quy trình phát triển giúp nâng cao chất lượng phần mềm. Ngay cả những quy trình không mang tính chất đảm bảo chất lượng cũng ảnh hưởng đến chất lượng sản phẩm cuối cùng.

## Quy trình kiểm soát thay đổi (Change-control procedures)

Các thay đổi không kiểm soát, nhất là với các yêu cầu, có thể làm gián đoạn thiết kế và coding, gây mất nhất quán hoặc tăng chi phí sửa đổi. Kiểm soát tốt các thay đổi giúp bảo đảm chất lượng ổn định.

## Đo lường kết quả (Measurement of results)

Việc đo lường là cần thiết để đánh giá hiệu quả của kế hoạch đảm bảo chất lượng. Nếu không đo lường, bạn sẽ không biết kế hoạch có thành công hay không, và cũng không thể điều chỉnh tốt hơn nữa.

## Prototyping (Nguyên mẫu)

Phát triển các mô hình thực tế cho các chức năng quan trọng giúp kiểm thử tính sử dụng (usability), tính toán hiệu suất hoặc kiểm tra yêu cầu bộ nhớ. Nghiên cứu chỉ ra prototyping dẫn đến thiết kế tốt hơn, đáp ứng nhu cầu người dùng tốt hơn và khả năng bảo trì cao hơn so với phương pháp phát triển truyền thống.

---

## Đặt mục tiêu rõ ràng

Việc đặt ra mục tiêu chất lượng rõ ràng là bước quan trọng nhưng dễ bị bỏ qua. Các nghiên cứu kinh điển cho thấy rằng lập trình viên sẽ thực sự nỗ lực đạt được các mục tiêu nếu chúng được giao phó một cách rõ ràng và hợp lý.

Thí nghiệm bởi Gerald Weinberg và Edward Schulman chỉ ra rằng các nhóm được giao tối ưu từng mục tiêu cụ thể (bộ nhớ tối thiểu, mã dễ đọc, thời gian lập trình ngắn nhất,...) đều đạt thành tích tốt nhất ở mục tiêu họ được giao, nhưng không thể làm tốt đồng thời mọi mục tiêu.

Mục tiêu	Tối ưu hóa	Ít bộ nhớ nhất	Đầu ra dễ đọc nhất	Code dễ đọc nhất	Ít dòng code nhất	Thời gian ngắn nhất
Nhóm tối ưu bộ nhớ	1	4	4	2	5	
Nhóm đầu ra dễ đọc	5	1	1	5	3	
Nhóm code dễ đọc	3	2	2	3	4	
Nhóm ít dòng code nhất	2	5	3	1	3	
Nhóm thời gian ít nhất	4	3	5	4	1	

Kết quả này hàm ý rằng mục tiêu rõ ràng giúp tối ưu hiệu quả, song các mục tiêu này thường mâu thuẫn nhau nên không thể tối ưu cùng lúc tất cả.

---

## 20.3 Hiệu quả Tương đối của các Kỹ thuật Chất lượng

Các kỹ thuật đảm bảo chất lượng phần mềm không mang lại hiệu quả như nhau và mỗi kỹ thuật có mức độ phát hiện lỗi khác nhau. Một cách đánh giá là xác định **tỉ lệ phát hiện lỗi (defect-detection rate)** của từng kỹ thuật:

Giai đoạn/Phương pháp	Tỷ lệ thấp nhất	Tỷ lệ trung bình (modal)	Tỷ lệ cao nhất
Đánh giá thiết kế không chính thức	25%	35%	40%
Inspection thiết kế chính thức	45%	55%	65%
Rà soát code không chính thức	20%	25%	35%
Inspection code chính thức	45%	60%	70%
Xây dựng mô hình/nguyên mẫu	35%	65%	80%
Tự desk-check code	20%	40%	60%
Unit test	15%	30%	50%
New function (component) test	20%	30%	35%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
System test	25%	40%	55%
Beta test quy mô nhỏ (<10 site)	25%	35%	40%
Beta test quy mô lớn (>1000 site)	60%	75%	85%

**Nhận xét:** Không có kỹ thuật riêng lẻ nào có thể phát hiện trên 75% lỗi, mức trung bình chỉ khoảng 40%. Với mô hình phát hiện lỗi nặng về kiểm thử như đơn vị hay tích hợp (unit/integration testing), tỉ lệ thường chỉ ở mức 30-35%. Chỉ những tổ chức dẫn đầu mới kết hợp đa dạng kỹ thuật mới đạt hiệu quả trên 95% trong việc loại bỏ lỗi.

Một nghiên cứu cổ điển của Glenford Myers cho thấy: kết hợp từ hai phương pháp trở lên (ngay cả khi lặp lại cùng một phương pháp) sẽ tăng gấp đôi số lượng lỗi phát hiện so với sử dụng riêng lẻ từng phương pháp.

**Tóm lại:** Để đạt hiệu quả tối ưu trong đảm bảo chất lượng phần mềm, cần kết hợp nhiều kỹ thuật kiểm soát, kiểm thử, đánh giá và các thực hành quy trình – không nên chỉ phụ thuộc vào một hoặc một vài khâu. Các mục tiêu chất lượng cần được xác định rõ ràng, đo lường thường xuyên và tổ chức nên nuôi dưỡng tư duy chất lượng như một ưu tiên hàng đầu xuyên suốt mọi quá trình phát triển.

## Khoảng 20% lỗi được nhiều kiểm tra viên phát hiện

Chỉ khoảng 20% các lỗi được tìm thấy qua các hoạt động kiểm tra (inspections) là do nhiều hơn một kiểm tra viên phát hiện (Kouchakdjian, Green, và Basili 1989; Tripp, Struck, và Pflug 1991; Schneider, Martin, và Tsai 1992).

Glenford Myers chỉ ra rằng các quy trình do con người thực hiện (ví dụ inspections và walk-throughs) thường tốt hơn các phương pháp kiểm thử dựa trên máy tính trong việc phát hiện một số loại lỗi nhất định, và ngược lại đối với các loại lỗi khác (1979). Kết quả này được xác nhận trong một nghiên cứu sau đó, trong đó cho thấy việc đọc mã nguồn (code reading) phát hiện được nhiều lỗi giao diện (interface defect) hơn, còn kiểm thử chức năng (functional testing) lại phát hiện nhiều lỗi điều khiển (control defect) hơn (Basili, Selby, và Hutchens 1986).

Chuyên gia kiểm thử Boris Beizer báo cáo rằng các phương pháp kiểm thử phi chính thức chỉ đạt được mức độ bao phủ kiểm thử (test coverage) ở mức 50–60% trừ khi sử dụng công cụ phân tích mức độ bao phủ (coverage analyzer) (Johnson 1994).

**Kết luận:** Các phương pháp phát hiện lỗi hoạt động hiệu quả hơn khi được kết hợp với nhau thay vì sử dụng đơn lẻ. Jones cũng đưa ra nhận xét tương tự khi quan sát rằng hiệu quả tích lũy của các kỹ thuật phát hiện lỗi cao hơn đáng kể so với từng kỹ thuật riêng lẻ. Thực tế, hiệu quả của việc chỉ sử dụng kiểm thử là rất hạn chế.

Jones chỉ ra rằng sự kết hợp giữa kiểm thử đơn vị (unit testing), kiểm thử chức năng (functional testing), và kiểm thử hệ thống (system testing) thường chỉ mang lại tỷ lệ phát hiện lỗi tích lũy dưới 60%, mức này thường không đủ đáp ứng yêu cầu phần mềm sản xuất.

Dữ liệu này cũng giúp lý giải tại sao các lập trình viên áp dụng phương pháp loại bỏ lỗi có kỷ luật như Extreme Programming lại đạt được mức độ loại bỏ lỗi cao hơn mức họ từng trải nghiệm. Như Bảng 20-3 minh họa, tập hợp các phương pháp loại bỏ lỗi được sử dụng trong Extreme Programming dự kiến sẽ đạt hiệu quả loại bỏ lỗi khoảng 90% trong trường hợp trung bình và 97% trong trường hợp tốt nhất, vượt xa mức trung bình ngành là 85%.

Mặc dù một số người cho rằng hiệu quả này đến từ sự cộng hưởng giữa các thực tiễn trong Extreme Programming, thực chất đây là kết quả tất yếu của việc sử dụng những phương pháp loại bỏ lỗi cụ thể này. Các sự kết hợp khác cũng có thể mang lại hiệu quả tương đương hoặc tốt hơn, và việc xác định các phương pháp loại bỏ lỗi cụ thể để đạt được chất lượng mong muốn là một phần quan trọng của lập kế hoạch dự án hiệu quả.

### Bảng 20-3: Ước lượng tỷ lệ phát hiện lỗi của Extreme Programming

Phương pháp loại bỏ lỗi	Tỷ lệ thấp nhất	Tỷ lệ trung bình	Tỷ lệ cao nhất
-------------------------	-----------------	------------------	----------------

Phương pháp loại bỏ lỗi	Tỷ lệ thấp nhất	Tỷ lệ trung bình	Tỷ lệ cao nhất
Đánh giá thiết kế phi chính thức (pair programming)	25%	35%	40%
Đánh giá code phi chính thức (pair programming)	20%	25%	35%
Kiểm tra code cá nhân tại bàn làm việc	20%	40%	60%
Kiểm thử đơn vị (Unit test)	15%	30%	50%
Kiểm thử tích hợp (Integration test)	25%	35%	40%
Kiểm thử hồi quy (Regression test)	15%	25%	30%
Hiệu quả loại bỏ lỗi tích lũy kỳ vọng	~74%	~90%	~97%

## Chi phí phát hiện lỗi

Một số phương pháp phát hiện lỗi tốn kém hơn những phương pháp khác. Các phương pháp hiệu quả nhất thường có chi phí phát hiện lỗi trên mỗi lỗi thấp nhất, với điều kiện các yếu tố khác giữ nguyên. Điều này quan trọng, vì chi phí trên mỗi lỗi còn bị ảnh hưởng bởi: tổng số lỗi tìm ra, giai đoạn phát hiện lỗi, và các yếu tố khác ngoài bản chất kinh tế của kỹ thuật phát hiện lỗi cụ thể.

Hầu hết các nghiên cứu đều phát hiện rằng inspection rẻ hơn testing. Một nghiên cứu tại Software Engineering Laboratory cho thấy việc đọc mã code (code reading) tìm được nhiều lỗi hơn so với testing khoảng 80% theo đơn vị thời gian (Basili và Selby 1987). Một tổ chức khác phát hiện rằng chi phí phát hiện lỗi thiết kế bằng testing cao gấp 6 lần so với inspections (Ackerman, Buchwald, và Lewski 1989). Một nghiên cứu tại IBM cho thấy chỉ cần 3,5 giờ công để tìm mỗi lỗi qua inspection, trong khi phải mất 15–25 giờ qua kiểm thử (Kaplan 1995).

### Chi phí sửa lỗi

Chi phí phát hiện lỗi chỉ là một phần của "phương trình chi phí" – phần còn lại là chi phí sửa lỗi. Có thể nghĩ rằng phương pháp tìm ra lỗi không ảnh hưởng đến chi phí sửa chữa – tuy nhiên đó là nhận định sai lầm. Càng để lỗi tồn tại lâu trong hệ thống, chi phí cho việc khắc phục càng tăng cao. Phương pháp phát hiện lỗi càng sớm càng giúp giảm chi phí sửa chữa.

*Thêm vào đó*, những phương pháp như inspections phát hiện đồng thời cả triệu chứng và nguyên nhân gốc của lỗi trong một bước, còn những phương pháp như testing chỉ tìm triệu chứng và cần bổ sung công đoạn xác định nguyên nhân gốc – khiến tổng chi phí đội lên nhiều lần.

Ví dụ: Bộ phận phần mềm của Microsoft xác định để tìm và sửa một lỗi qua inspection (một bước), mất trung bình 3 tiếng; còn thông qua testing (hai bước), mất khoảng 12 tiếng (Moore 1992). Collofello và Woodfield báo cáo về một chương trình 700.000 dòng do hơn 400 lập trình viên phát triển, cho thấy code reviews hiệu quả hơn testing nhiều lần – tỉ lệ hoàn vốn (ROI) đạt 1.38 so với 0.17.

**Tóm lại:** Một chương trình đảm bảo chất lượng phần mềm hiệu quả phải kết hợp nhiều phương pháp ở tất cả các giai đoạn phát triển. Đề xuất:

- Tiến hành inspections chính thức đối với mọi yêu cầu, kiến trúc, và thiết kế phần quan trọng
- Xây dựng mô hình hoặc dựng prototype
- Đọc code hoặc inspection code
- Kiểm thử thực thi

## 20.4 Khi nào thực hiện Đảm bảo Chất lượng (Quality Assurance)

Như đã trình bày ở Chương 3 ("Đo hai lần, cắt một lần: Điều kiện tiên quyết ở giai đoạn đầu"), càng phát hiện lỗi sớm càng hạn chế việc lỗi "len lỏi" và làm tăng chi phí sửa chữa về sau. Một lỗi ở giai đoạn yêu cầu (requirement) có thể kéo theo nhiều lỗi ở thiết kế (design), sau đó gây thêm nhiều lỗi mã nguồn. Lỗi yêu cầu có thể tạo ra cấu trúc thừa trong kiến trúc, dẫn đến code, test case, và tài liệu thừa; hoặc dẫn đến quyết định kiến trúc tồi, cần hủy bỏ sau này.

Giống như việc cần phát hiện lỗi trong bản vẽ thiết kế nhà trước khi đổ bê tông móng, việc phát hiện lỗi yêu cầu hoặc kiến trúc trước khi triển khai các hoạt động tiếp theo sẽ giúp tiết kiệm chi phí đáng kể.

Ngoài ra, lỗi ở giai đoạn yêu cầu hoặc kiến trúc mang tính lan rộng hơn lỗi phát sinh trong giai đoạn lập trình – một lỗi kiến trúc có thể ảnh hưởng cả chục lớp/phương thức, trong khi lỗi xây dựng thường chỉ ảnh hưởng nhỏ. Do đó, việc bắt lỗi sớm cũng là biện pháp tiết kiệm chi phí hiệu quả.

### Lời khuyên:

Nên tập trung công việc đảm bảo chất lượng ở các giai đoạn đầu và duy trì xuyên suốt toàn bộ dự án – từ khi bắt đầu kế hoạch tới khi kết thúc.

## 20.5 Nguyên tắc chung về Chất lượng phần mềm

Không có cái gọi là “bữa trưa miễn phí” (no free lunch) – và ngay cả khi có thì cũng không có gì đảm bảo đó là bữa trưa ngon. Song chất lượng phần mềm sở hữu một đặc điểm đặc biệt: **việc nâng cao chất lượng thực ra làm giảm chi phí phát triển**.

Hiểu nguyên tắc này đòi hỏi nhận thức rằng cách nâng cao chất lượng và năng suất tốt nhất chính là **giảm thời gian dành cho việc sửa lại code** (rework), bất kể đó là sửa đổi yêu cầu, thiết kế, hay debug.

Sản lượng trung bình ngành cho sản phẩm phần mềm rơi vào khoảng 10–50 dòng code hoàn chỉnh/ngày/người (bao gồm tất cả các chi phí ngoài lập trình). Trong khi chỉ mất vài phút để gõ 10–50 dòng code, vậy thời gian còn lại được dùng làm gì?

Phần lớn thời gian này dành cho **việc debug, refactor và sửa lại code sai**. Các hoạt động này chiếm khoảng 50% thời gian ở một vòng đời phát triển truyền thống.

**Kết luận:** Giảm thời gian debug bằng cách phòng tránh lỗi ngay từ sớm sẽ nâng cao năng suất dự án.

Các dữ liệu thực nghiệm khẳng định nhận định này: Nghiên cứu tại NASA với 50 dự án (~3 triệu dòng code), tăng cường đảm bảo chất lượng giúp giảm tỷ lệ lỗi mà không làm tăng chi phí phát triển tổng thể (Card 1987). Nghiên cứu tại IBM cũng xác nhận rằng các dự án có ít lỗi nhất lại có lịch trình phát triển ngắn nhất và năng suất cao nhất (Jones 2000).

Ở phạm vi nhỏ hơn, nghiên cứu năm 1985 với 166 lập trình viên: người “hoàn thành chương trình” với thời gian trung bình lại viết nhiều lỗi nhất; những ai nhanh/chậm hơn thời gian trung bình đều viết ít lỗi hơn (DeMarco and Lister 1985).

**Không phải cứ làm nhanh hay chậm là ít lỗi, mà việc lập kế hoạch chất lượng phù hợp sẽ giúp tiết kiệm chi phí, thời gian, và tạo ra phần mềm chất lượng cao hơn.**

## cc2e.com/2043 CHECKLIST: Kế hoạch đảm bảo chất lượng

- Đã xác định các đặc tính chất lượng cụ thể quan trọng cho dự án chưa?
- Các thành viên khác có nhận thức được mục tiêu chất lượng của dự án không?
- Đã phân biệt rõ các đặc tính chất lượng bên ngoài và bên trong chưa?
- Đã đánh giá khả năng xung đột hoặc tương hỗ giữa các đặc tính chất lượng chưa?
- Dự án có sử dụng nhiều kỹ thuật phát hiện lỗi khác nhau, phù hợp với các loại lỗi khác nhau không?
- Dự án đã lên kế hoạch đảm bảo chất lượng ở từng giai đoạn phát triển chưa?
- Đã có hệ thống đo lường chất lượng để xác định xu hướng chất lượng?
- Quản lý có nhận thức rằng đảm bảo chất lượng đòi hỏi đầu tư chi phí upfront để tiết kiệm về sau không?

## Tài liệu tham khảo thêm

- Ginac, Frank P. *Customer Oriented Software Quality Assurance*. Prentice Hall, 1998. Một cuốn sách ngắn gọn, mô tả chi tiết các thuộc tính chất lượng, chỉ số đo chất lượng (quality metrics), chương trình QA (Quality Assurance), vai trò của kiểm thử (testing), và các chương trình cải tiến chất lượng như CMM (Capability Maturity Model) và ISO 9000.
- Lewis, William E. *Software Testing and Continuous Quality Improvement*, 2nd ed. Auerbach Publishing, 2000. Bàn luận toàn diện về vòng đời chất lượng, các kỹ thuật kiểm thử và cung cấp nhiều mẫu biểu, checklist.

## Các tiêu chuẩn liên quan

- **IEEE Std 730-2002:** Tiêu chuẩn kế hoạch đảm bảo chất lượng phần mềm (Software Quality Assurance Plans).
- **IEEE Std 1061-1998:** Tiêu chuẩn phương pháp luận đo lường chất lượng phần mềm (Software Quality Metrics Methodology).
- **IEEE Std 1028-1997:** Tiêu chuẩn đánh giá phần mềm (Software Reviews).
- **IEEE Std 1008-1987 (R1993):** Tiêu chuẩn kiểm thử đơn vị phần mềm (Software Unit Testing).
- **IEEE Std 829-1998:** Tiêu chuẩn tài liệu kiểm thử phần mềm (Software Test Documentation).

## Các điểm mấu chốt (Key Points)

- Chất lượng về bản chất là miễn phí, nhưng cần phân bổ lại nguồn lực để phòng tránh lỗi thay vì khắc phục lỗi sau cùng với chi phí cao.
- Không phải tất cả các mục tiêu đảm bảo chất lượng đều có thể đạt được đồng thời. Hãy xác định mục tiêu nào cần ưu tiên và truyền đạt rõ ràng cho các thành viên.
- Không có một kỹ thuật phát hiện lỗi nào hiệu quả hoàn toàn khi dùng riêng lẻ. Không nên chỉ dựa vào kiểm thử (testing) để loại bỏ lỗi. Chương trình đảm bảo chất lượng thành công sử dụng kết hợp nhiều kỹ thuật cho các loại lỗi khác nhau.
- Có thể áp dụng các kỹ thuật hiệu quả trong và trước giai đoạn xây dựng phần mềm. Càng phát hiện lỗi sớm, mức độ ảnh hưởng và thiệt hại càng ít.
- Đảm bảo chất lượng phần mềm là vấn đề thuộc về quy trình. Khác với sản xuất, không có giai đoạn lặp lại (repetitive phase) ảnh hưởng tới sản phẩm cuối cùng – chất lượng phần mềm được kiểm soát bởi quy trình phát triển.

## Chương 21: Phát triển cộng tác (Collaborative Construction)

### 21.1 Tổng quan về các thực tiễn phát triển cộng tác

*Phát triển cộng tác* (collaborative construction) là thuật ngữ dùng để chỉ các kỹ thuật như pair programming, kiểm tra chính thức (formal inspections), đánh giá kỹ thuật phi chính thức (informal technical reviews), đọc tài liệu mã nguồn (document reading), và các phương pháp khác mà ở đó các lập trình viên cùng chia sẻ trách nhiệm tạo ra mã nguồn và sản phẩm liên quan.

Thuật ngữ “collaborative construction” được Matt Peloquin tại công ty của tác giả đặt ra khoảng năm 2000 – và các nhóm khác cũng độc lập sử dụng.

Dù khác biệt, tất cả các kỹ thuật phát triển cộng tác dựa trên quan niệm: các lập trình viên có những “điểm mù” với công việc của mình, người khác lại không có điểm mù tương tự, và nhận được phản hồi của đồng nghiệp sẽ giúp phát hiện lỗi hiệu quả hơn.

Một nghiên cứu tại Software Engineering Institute cho biết: lập trình viên đưa vào trung bình 1–3 lỗi mỗi giờ khi thiết kế, và 5–8 lỗi mỗi giờ khi viết code (Humphrey 1997) – tấn công các “điểm mù” này là chìa khóa để phát triển phần mềm hiệu quả.

#### Phát triển cộng tác hỗ trợ các biện pháp đảm bảo chất lượng khác

- Mục tiêu chính: tăng chất lượng phần mềm. Như đã phân tích ở Chương 20, kiểm thử phần mềm chỉ phát hiện trung bình khoảng 30% lỗi ở mức unit testing, 35% ở integration testing và beta testing. Ngược lại, kiểm tra thiết kế (design inspection) và kiểm tra code (code inspection) có tỉ lệ phát hiện lỗi trung bình là 55% và 60% (Jones 1996).
- Lợi ích thứ cấp: rút ngắn thời gian phát triển, qua đó giảm chi phí.
- Pair programming có thể đạt mức chất lượng code tương đương inspections chính thức (Shull et al 2002). Dù chi phí pair programming cao hơn phát triển cá nhân 10–25%, nhưng lại giúp giảm đến 45% thời gian phát triển – trong một số trường hợp đây là ưu thế đáng kể so với phát triển cá nhân (Boehm và Turner 2004).

Các nghiên cứu thực tế:

- **IBM:** mỗi giờ dành cho inspection giúp tránh khoảng 100 giờ công việc liên quan (kiểm thử và sửa lỗi) (Holland 1999).
- **Raytheon:** nhờ tập trung vào inspections, chi phí sửa lỗi (rework) giảm từ 40% xuống còn 20% tổng chi phí dự án (Haley 1996).
- **Hewlett-Packard:** chương trình inspections tiết kiệm ~21.5 triệu USD mỗi năm (Grady và Van Slack 1994).

*Các nội dung tiếp theo về các phương pháp phát triển cộng tác sẽ phân tích chi tiết hơn về pair programming, formal inspections, và các kỹ thuật liên quan.*

## Nghiên cứu về Hiệu quả của Inspections (Kiểm tra và phản biện sản phẩm phần mềm)

### Hiệu quả của Inspections so với Kiểm thử và Ảnh hưởng đến Quản lý Bảo trì

- Một nghiên cứu về các chương trình lớn cho thấy rằng mỗi giờ dành cho inspections giúp tránh trung bình 33 giờ công việc bảo trì và inspections hiệu quả gấp tới 20 lần so với kiểm thử (Russell 1991).
- Trong một tổ chức bảo trì phần mềm, trước khi áp dụng code review (xem xét mã nguồn), 55% các thay đổi một dòng đều có lỗi. Sau khi áp dụng reviews, tỷ lệ lỗi chỉ còn 2% (Freedman và Weinberg 1990). Khi xem

xét tất cả thay đổi, 95% được thực hiện đúng từ lần đầu sau review, trong khi trước đó chỉ đạt dưới 20%.

- Một nhóm 11 chương trình do cùng một nhóm phát triển và được phát hành ra môi trường sản xuất: Năm chương trình đầu tiên phát triển không có inspections có trung bình 4,5 lỗi trên 100 dòng mã; sáu chương trình còn lại có inspections, con số này chỉ còn 0,82 lỗi/100 dòng mã. Reviews giúp giảm lỗi tới hơn 80% (Freedman và Weinberg 1990).
- Capers Jones báo cáo rằng trong tất cả các dự án phần mềm mà ông nghiên cứu đạt tỷ lệ loại bỏ lỗi 99% trở lên, tất cả đều sử dụng inspections chính thức; không dự án nào đạt hiệu quả loại bỏ lỗi dưới 75% lại sử dụng inspections chính thức (Jones 2000).

Nhiều ví dụ trên minh họa *Nguyên lý Chung về Chất lượng Phần mềm*: việc giảm số lượng lỗi phần mềm đồng thời cải thiện tiến độ phát triển. Các nghiên cứu đã chứng minh rằng, hợp tác trong reviews không chỉ hiệu quả hơn việc kiểm thử đơn lẻ trong việc phát hiện lỗi, mà còn giúp phát hiện các loại lỗi khác với kiểm thử (Myers 1978; Basili, Selby, Hutchens 1986).

**Karl Wieggers** nhấn mạnh:

"Một reviewer là con người có thể phát hiện thông báo lỗi khó hiểu, chú thích chưa đầy đủ, giá trị biến hard-coded (hard-coded variable values - giá trị biến được gán cố định trong mã), cũng như mẫu code lặp lại cần phải tổng quát hóa – điều mà kiểm thử không làm được." (Wieggers 2002)

Một tác động phụ của reviews là khi biết rằng sản phẩm sẽ được xem xét, lập trình viên có xu hướng kiểm tra, rà soát sản phẩm kỹ hơn. Vì vậy, ngay cả khi kiểm thử được thực hiện hiệu quả, reviews hay các hình thức cộng tác khác vẫn cần thiết trong chương trình đảm bảo chất lượng toàn diện.

---

## Collabrative Construction (Xây dựng phần mềm cộng tác) - Đào tạo, Văn hoá và Chuyên môn

### Truyền đạt quy trình, chuẩn mực qua reviews

- Các quy trình review không chính thức được truyền miệng trong giới lập trình rất lâu trước khi chúng trở thành chuẩn mực.
- Reviews là cơ chế quan trọng để cung cấp phản hồi cho lập trình viên về sản phẩm của họ, về việc tuân thủ chuẩn mực, cũng như giải thích lý do nên làm theo chuẩn.
- Lập trình viên cần phản hồi không chỉ về chuẩn mực mà còn là các khía cạnh chủ quan:
  - Định dạng mã nguồn
  - Chú thích
  - Đặt tên biến
  - Sử dụng biến cục bộ, toàn cục
  - Phương pháp thiết kế
  - Phong cách chung và cách làm việc tại công ty

*Những lập trình viên thiếu kinh nghiệm cần được hướng dẫn; người giàu kinh nghiệm phải khuyến khích chia sẻ, truyền đạt cho đồng nghiệp.*

Reviews tạo ra diễn đàn để các lập trình viên ở các mức kinh nghiệm khác nhau thảo luận các vấn đề kỹ thuật – là cơ hội cải thiện chất lượng sản phẩm lâu dài. Một nhóm sử dụng formal inspections nhận thấy: inspections nhanh chóng đưa năng lực của mọi thành viên lên ngang bằng với những lập trình viên xuất sắc nhất (Tackett và Van Doren 1999).

---

## Tinh thần sở hữu tập thể (Collective Ownership)

Khi áp dụng **collective ownership (tinh thần sở hữu tập thể)**, toàn bộ mã nguồn thuộc nhóm chứ không thuộc về cá nhân, mọi thành viên đều có thể truy cập và sửa đổi.

**Lợi ích cụ thể:**

- Cải thiện chất lượng mã nhờ nhiều người cùng kiểm tra và phát triển.
- Giảm ảnh hưởng khi một người rời nhóm do nhiều người quen thuộc với từng phần mã.
- Chu trình sửa lỗi ngắn hơn vì bất kỳ ai cũng có thể tham gia sửa lỗi.

Một số phương pháp luận như *Extreme Programming* đề xuất ghép đôi lập trình viên chính thức, luân phiên nhiệm vụ – công ty của tác giả áp dụng kiểm tra kỹ thuật chính thức và không chính thức, lập trình đôi khi cần, và luân phiên nhiệm vụ sửa lỗi.

---

## Cộng tác trước, trong và sau quá trình xây dựng



Mặc dù chương này tập trung vào xây dựng chi tiết và code, những nhận xét về collaborative construction (xây dựng cộng tác) cũng áp dụng cho các hoạt động phân tích, ước lượng, lập kế hoạch, kiến trúc và bảo trì.

---

## 21.2 Pair Programming (Lập trình đôi)

**Pair programming:** Một người nhập mã (coding) trên bàn phím, người còn lại quan sát lỗi, suy nghĩ về hướng đi tổng thể, kiểm tra tính đúng của giải pháp.

Khái niệm này bắt nguồn từ *Extreme Programming* (Beck 2000), hiện được ứng dụng rộng rãi (Williams & Kessler 2002).

### Cách triển khai hiệu quả Pair Programming

- **Hỗ trợ bằng coding standards (chuẩn mã hóa):** Giúp tránh tranh cãi không cần thiết về phong cách lập trình.
- **Người không code cần chủ động:** Không chỉ quan sát, phải phân tích, đề xuất cải tiến...
- **Không áp dụng với nhiệm vụ quá đơn giản:** Các nhóm thường chỉ dùng với phần code phức tạp.
- **Luân phiên cặp và nhiệm vụ:** Giúp mọi người hiểu nhiều phần hệ thống, tăng khả năng chia sẻ tri thức, giảm phụ thuộc.
- **Cặp đôi ngang nhịp làm việc:** Người nhanh phải điều chỉnh hoặc hoán đổi cặp phù hợp.
- **Tránh ép buộc những người không hợp tác hoặc có mâu thuẫn cá nhân phải làm đôi với nhau.**
- **Không kết hợp hai người mới cùng một cặp:** Ít nhất một người nên có kinh nghiệm với pairing.
- **Bổ nhiệm team leader (trưởng nhóm):** Để điều phối công việc và là đầu mối với bên ngoài.

### Lợi ích của Pair Programming

- Chịu áp lực tốt hơn làm việc đơn lẻ, đồng đội hỗ trợ duy trì chất lượng.
  - Nâng cao chất lượng mã, sự rõ ràng và dễ hiểu gia tăng đáng kể.
  - Rút ngắn lịch trình; số lỗi giảm, tiết kiệm thời gian sửa cuối dự án.
  - Góp phần truyền đạt văn hóa doanh nghiệp, mentoring (kèm cặp), và thực hành sở hữu tập thể.
- 

### Checklist: Pair Programming Hiệu quả

- ☐ Có coding standard để tránh tranh luận về phong cách?
  - ☐ Cả hai thành viên đều tham gia chủ động?
  - ☐ Lựa chọn đúng nhiệm vụ để lập trình đôi (không áp dụng đại trà)?
  - ☐ Luân chuyển cặp và công việc thường xuyên?
  - ☐ Cặp đôi phù hợp về tốc độ và tính cách?
  - ☐ Có trưởng nhóm điều phối công việc?
- 

## 21.3 Formal Inspections (Kiểm tra chính thức)

**Formal inspection** là một dạng review (xem xét) đặc biệt, chứng minh hiệu quả trong phát hiện lỗi, tiết kiệm kinh phí hơn kiểm thử.

### Đặc điểm chính so với review thông thường:

- Dùng checklist tập trung vào các vấn đề thường gặp.
- Mục tiêu phát hiện (không sửa) lỗi.
- Thành viên phải chuẩn bị trước, mang theo danh sách lỗi đã tìm thấy.
- Vai trò phân công rõ ràng cho mọi thành viên.
- Moderator (chủ tọa) không phải là tác giả sản phẩm.
- Moderator phải được đào tạo về điều phối inspections.
- Chỉ họp khi mọi thành viên chuẩn bị đầy đủ.
- Dữ liệu thu thập tại mỗi inspection dùng để cải thiện các lần sau.
- Quản lý không nên dự (trừ khi review tài liệu quản lý); báo cáo kết quả inspection được gửi cho họ.

### Kết quả đạt được

- Inspections cá nhân thường phát hiện ~60% lỗi, vượt các kỹ thuật khác trừ prototype và beta testing số lượng lớn.
- Kết hợp inspections thiết kế và mã loại bỏ ~70–85% lỗi.
- Số lỗi mỗi 1000 dòng giảm 20–30% (so với reviews không chính thức).
- Inspections giúp tăng năng suất ~20% và tiết kiệm tổng chi phí dự án.

- Inspections giúp đánh giá tiến độ kỹ thuật (không phải tiến độ quản lý).

## Các vai trò trong Inspection

- **Moderator:** Đảm bảo tiến độ inspection, phân phối tài liệu, checklist, tổ chức họp, tổng hợp kết quả và theo dõi nhiệm vụ được giao.
- **Author (tác giả):** Vai trò phụ; trình bày những phần không rõ, giải thích tại sao những điều tưởng là lỗi lại chấp nhận được.
- **Reviewer:** Người có lợi ích liên quan, tìm lỗi trong quá trình chuẩn bị và trong buổi inspection.
- **Scribe (thư ký):** Ghi lại các lỗi và nhiệm vụ đã giao, không nên là author hay moderator.
- **Management:** Không trực tiếp tham dự; nhận báo cáo tổng kết inspection.

**Lưu ý:** Tuyệt đối không dùng kết quả inspection để đánh giá xếp loại nhân sự; đánh giá hiệu suất cần dựa sản phẩm hoàn chỉnh, không dùng bản đang phát triển.

Một buổi inspection nên có ít nhất 3 người (unique role: moderator, author, reviewer). Nhiều hơn 6 người làm giảm hiệu quả. Thường 2–3 reviewer là đủ (kết quả không tăng nhiều khi thêm reviewer - Bush & Kelly 1989; Porter & Votta 1997). Tuy nhiên, nên cân nhắc phù hợp với từng môi trường.

## Quy trình Inspection tổng quan

### 1. Planning (Lập kế hoạch):

- Tác giả gửi tài liệu cho moderator.
- Moderator quyết định ai sẽ review, khi nào và ở đâu họp, phân phối checklist, in tài liệu kèm số dòng.

### 2. Overview (Tổng quan kỹ thuật):

- Tác giả (nếu cần) trình bày các khái niệm, môi trường. Lưu ý: tránh để overview thay cho tính minh bạch của tài liệu.

### 3. Preparation (Chuẩn bị cá nhân):

- Reviewer kiểm tra tài liệu theo checklist, đọc lập.
- Tốc độ: mã ứng dụng ~500 dòng/giờ, mã hệ thống ~125 dòng/giờ – tùy loại hình.

### 4. Perspective-based review (phân vai theo góc nhìn):

- Reviewer có thể tiếp cận tài liệu như người bảo trì, khách hàng hoặc nhà thiết kế.
- Có thể kiểm tra theo kịch bản cụ thể (“Liệu có yêu cầu nào chưa được đáp ứng? ...”).

### 5. Inspection meeting (Họp inspection):

- Moderator chỉ định người khác tác giả trình bày lại/đọc code.
- Toàn bộ logic được giải thích, mọi nhánh lệnh được xem xét.
- Scribe ghi nhận lỗi, thảo luận chỉ dừng lại khi xác nhận có lỗi, chuyển qua lỗi tiếp theo.
- Tốc độ vừa phải để không bỏ sót hoặc gây chán nản.

## Dịch thuật học thuật & tóm tắt: Collaborative Construction & Practices

### Tốc độ kiểm tra mã nguồn (Code Review Rate)

- Một số tổ chức đã phát hiện rằng, đối với **system code** (mã hệ thống), tốc độ kiểm tra tối ưu là khoảng **90 dòng mã/giờ**.
- Đối với **application code** (mã ứng dụng), tốc độ này có thể lên đến **500 dòng mã/giờ** (Humphrey 1989).
- Theo Wieggers (2002), mức trung bình tốt để bắt đầu là **150–200 câu lệnh mã nguồn không trắng, không chú thích/giờ**.

### Quy tắc khi tổ chức kiểm tra

- Không thảo luận giải pháp trong cuộc họp. Nhóm cần tập trung vào việc xác định các **defect** (lỗi).

- Một số nhóm còn không cho phép bàn luận liệu một vấn đề có thực sự là defect không, mà đơn giản giả định rằng nếu có người nhầm lẫn thì tài liệu/thiết kế/mã cần làm rõ.
- Cuộc họp kiểm tra không nên kéo dài quá hai giờ. Nếu kéo dài hơn, năng suất sẽ giảm mạnh do giới hạn về sự tập trung của reviewer, dựa trên kinh nghiệm thực tế của IBM và các công ty khác.
- Tránh tổ chức nhiều hơn một buổi kiểm tra trong cùng một ngày.

## Báo cáo kiểm tra (Inspection Report)

- Moderator (người điều phối) sẽ lập **inspection report** trong vòng một ngày sau cuộc họp, liệt kê từng defect với loại và mức độ nghiêm trọng.
- Báo cáo này giúp đảm bảo rằng mọi defect đều sẽ được sửa và là cơ sở xây dựng checklist phù hợp với đặc thù tổ chức.
- Việc thu thập dữ liệu về thời gian kiểm tra và số lỗi giúp tổ chức có các bằng chứng xác thực về hiệu quả của quá trình kiểm tra, đồng thời điều chỉnh, tối ưu phương pháp.

## Sửa lỗi (Rework) & Theo dõi (Follow-Up)

- Moderator giao các defect cho người có trách nhiệm (thường là tác giả) để sửa.
- Moderator đảm bảo mọi **rework** (sửa lỗi) đều được thực hiện đầy đủ. Tùy mức độ nghiêm trọng và số lượng lỗi, có thể kiểm tra lại toàn bộ, chỉ kiểm tra các sửa đổi, hay cho phép tác giả tự sửa tùy ý.

## Buổi thảo luận ngoài luồng (Third-Hour Meeting)

- Sau khi kiểm tra xong, có thể tổ chức buổi họp không chính thức để thảo luận về các giải pháp nếu các thành viên muốn.

## Điều chỉnh quy trình kiểm tra (Fine-Tuning the Inspection)

- Khi đã thành thạo kiểm tra, tổ chức có thể thử nghiệm cải tiến quy trình nhưng phải tiến hành đo lường ảnh hưởng thay đổi trước khi áp dụng chính thức.
- Lưu ý: Loại bỏ hoặc gộp các giai đoạn có thể tốn kém hơn tiết kiệm được (Fagan 1986).
- Khi phát hiện ra một số loại lỗi xuất hiện thường xuyên, cần bổ sung vào **checklist**, loại trừ các lỗi đã không còn phát sinh, và luôn duy trì checklist không quá một trang giấy để đảm bảo sử dụng hiệu quả.

## Ứng xử trong kiểm tra — "Egoless Inspections"

Mục đích của kiểm tra là phát hiện defect, không phải tranh cãi ai đúng/sai, cũng không phải để phê phán tác giả. Tác giả cần xem đây là một trải nghiệm tích cực, học hỏi, chứ không phải gây áp lực, chỉ trích cá nhân.

Nếu có nhận xét mang tính chế giễu hoặc cá nhân ("Ai biết Java cũng biết nên lập từ 0 đến num-1 chứ không phải từ 1 đến num"), moderator cần can thiệp kịp thời.

Tác giả nên chuẩn bị tâm lý sẽ nghe các phê bình kể cả những điểm không thật sự là lỗi và không nên cố bảo vệ quan điểm tại cuộc họp; có thể cân nhắc, phản hồi sau. Tác giả có quyền quyết định cuối cùng về cách xử lý các defect.

## Kinh nghiệm sử dụng kiểm tra trong biên soạn sách

Tác giả mô tả kinh nghiệm sử dụng kiểm tra chính thức (formal inspections) khi viết sách *Code Complete*. Dù bản thảo đã qua nhiều vòng review và chỉnh sửa, kiểm tra chính thức vẫn phát hiện ra hàng trăm lỗi chưa từng bị phát hiện, chứng minh hiệu quả vượt trội của phương pháp này.

## Tổng kết về kiểm tra chính thức

- **Checklist kiểm tra** giúp reviewer tập trung hơn.
- Quá trình kiểm tra là hệ thống, có chuẩn hóa vai trò và quy trình, để tối ưu nhờ vòng phản hồi liên tục.
- Thu thập dữ liệu về loại lỗi và tốc độ kiểm tra giúp tối ưu hóa liên tục.

## Mô hình CMM (Capability Maturity Model)

SEI (Software Engineering Institute) đã xây dựng mô hình CMM để đo hiệu quả quá trình phát triển phần mềm của tổ chức. Các quy trình kiểm tra đóng vai trò quan trọng trong việc đạt cấp độ cao nhất về chất lượng, năng suất.

## Checklist kiểm tra hiệu quả

- Có checklist tập trung vào các điểm yếu trong quá khứ?
- Tập trung vào phát hiện defect hơn là sửa?
- Reviewer được giao vai trò/scenario cụ thể để chuẩn bị?
- Các thành viên đủ thời gian chuẩn bị?
- Được tập huấn kỹ năng kiểm tra và điều phối?
- Họp kiểm tra không quá hai giờ?
- Thu thập dữ liệu về lỗi để tối ưu checklist?
- Có theo dõi và đánh giá việc thực hiện các action item?
- Quản lý không tham dự kiểm tra?
- Có kế hoạch kiểm tra lại các sửa đổi?

## Các hình thức hợp tác khác ngoài kiểm tra chính thức

### Walk-Throughs

- **Walk-through** là dạng review phổ biến, linh hoạt về hình thức, có thể tổ chức rất không chính thức (tranh luận cạnh whiteboard) hoặc khá bài bản.
- Đặc điểm chung:
  - Chủ trì và điều phối thường là tác giả.
  - Tập trung vào vấn đề kỹ thuật.
  - Mọi người đều phải chuẩn bị trước.
  - Không có sự tham gia của quản lý.
  - Kéo dài 30-60 phút.
  - Tập trung phát hiện lỗi, không giải quyết lỗi.

Tuy nhiên, hiệu quả thường thấp hơn inspections, chỉ phát hiện 20–40% lỗi. Nếu áp dụng không nghiêm túc dễ tốn nhiều thời gian, chi phí (Glass, 1982).

### Code Reading

- Reviewer đọc mã độc lập rồi họp với tác giả thảo luận. Thường kiểm tra với các đoạn 1,000–10,000 dòng. Ưu tiên việc mỗi member tập trung cá nhân phát hiện lỗi. Hiệu quả cao với nhóm làm việc phân tán.
- Theo nghiên cứu tại NASA, **code reading** tìm được nhiều lỗi (~3.3 defect/giờ), cao hơn so với testing (~1.8 defect/giờ) (Card 1987), và tới **20–60% defect** trong vòng đời dự án.

### Dog-and-Pony Shows

- Là các buổi trình diễn/phỏng vấn sản phẩm phần mềm cho khách hàng, thường xuất hiện ở dự án chính phủ. Tập trung trình diễn hơn là đánh giá kỹ thuật, không nên dùng để cải thiện chất lượng kỹ thuật sản phẩm.

## So sánh các hình thức hợp tác (Xem bảng)

Thuộc tính	Pair Programming	Formal Inspection	Walk-Throughs (Kiểm tra không chính thức)
Vai trò định nghĩa	Có	Có	Không
Đào tạo chính thức	Có thể (qua huấn luyện thực tế)	Có	Không
Ai điều phối	Người trực tiếp code	Moderator	Tác giả
Trọng tâm	Thiết kế, viết code, test, sửa lỗi	Phát hiện lỗi	Đa dạng
Tập trung vào loại lỗi phổ biến	Không/chỉ một phần	Có	Không
Theo dõi sửa lỗi	Có	Có	Không
Tăng hiệu quả quy trình nhờ phân tích kết quả	Không	Có	Không
Tỉ lệ lỗi phát hiện (%)	40-60	45-70	20-40

## Lựa chọn kỹ thuật hợp tác

Nếu Pair Programming và Formal Inspection cho kết quả tương tự, quyết định chọn phương pháp nào nên dựa vào phong cách làm việc, sở thích cá nhân của nhóm hoặc từng thành viên trong nhóm, cũng như tính đặc thù của dự án.

## Nguồn tham khảo bổ sung

- Xem chi tiết về Capability Maturity Model (CMM) và các tài liệu về kiểm tra tại: **SEI, Humphrey (1989), Weinberg (1998)**
- [Code Complete website](#)

---

## Ghi chú thuật ngữ:

- **API (Application Programming Interface)**: Giao diện lập trình ứng dụng
- **Defect**: Lỗi
- **Moderator**: Người điều phối kiểm tra
- **Checklist**: Danh sách kiểm tra
- **Inspection report**: Báo cáo kiểm tra
- **CMM (Capability Maturity Model)**: Mô hình năng lực phát triển
- **Rework**: Sửa chữa sau kiểm tra

---

Bản dịch giữ nguyên các đoạn code nếu có (không có trong nguyên bản này), áp dụng định dạng markdown, và chú thích thuật ngữ chuyên ngành như yêu cầu.

# Tài liệu chuyên khảo về Phát triển Phần mềm: Pair Programming và Developer Testing

## Pair Programming

### Sách chuyên ngành

- **Williams, Laurie, and Robert Kessler. *Pair Programming Illuminated*. Boston, MA: Addison Wesley, 2002.**
  - Cuốn sách này trình bày chi tiết các khía cạnh của pair programming (lập trình cặp), bao gồm cách xử lý các dạng kết hợp tính cách khác nhau (ví dụ, chuyên gia và người chưa có kinh nghiệm, người hướng nội và người hướng ngoại) cũng như các vấn đề về triển khai thực tiễn.
- **Beck, Kent. *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison Wesley, 2000.**
  - Cuốn sách này đề cập ngắn gọn đến pair programming và minh họa cách kỹ thuật này được sử dụng kết hợp với các phương pháp hỗ trợ lẫn nhau khác như coding standards (chuẩn mã nguồn), frequent integration (tích hợp thường xuyên), và regression testing (kiểm thử hồi quy).
- **Reifer, Donald. "How to Get the Most Out of Extreme Programming/Agile Methods," *Proceedings, XP/Agile Universe 2002*. New York, NY: Springer; pp.185–196.**
  - Bài báo này tóm tắt kinh nghiệm công nghiệp với Extreme Programming (Lập trình cực đoan) và các phương pháp agile, đồng thời chỉ ra các yếu tố then chốt để thành công với pair programming.

---

## Reviews và Inspections

### Sách và Bài báo chuyên khảo

- **Wiegers, Karl. *Peer Reviews in Software: A Practical Guide*. Boston, MA: Addison Wesley, 2002.**
  - Cuốn sách này mô tả chi tiết các hình thức review khác nhau, bao gồm cả formal inspections (kiểm tra chính thức) và các phương pháp ít chính thức hơn. Tài liệu dựa trên nghiên cứu thực tiễn, tập trung vào ứng dụng thực tế và dễ đọc.
- **Gilb, Tom & Dorothy Graham. *Software Inspection*. Wokingham, England: Addison-Wesley, 1993.**
  - Mang lại thảo luận sâu về inspections khoảng đầu những năm 1990, với trọng tâm thực tiễn và các case study mô tả kinh nghiệm của nhiều tổ chức khi triển khai chương trình inspection.
- **Fagan, Michael E. "Design and Code Inspections to Reduce Errors in Program Development." *IBM Systems Journal* 15, no.3 (1976): 182–211.**

- **Fagan, Michael E.** “Advances in Software Inspections.” *IEEE Transactions on Software Engineering*, SE-12, no.7 (July 1986): 744–51.
  - Hai bài báo này do chính người phát triển inspections biên soạn, cung cấp cốt lõi thông tin cần thiết để vận hành một buổi inspection, bao gồm tất cả các form mẫu inspection tiêu chuẩn.

### Tiêu chuẩn liên quan

- **IEEE Std 1028-1997:** Tiêu chuẩn về Software Reviews
- **IEEE Std 730-2002:** Tiêu chuẩn về Software Quality Assurance Plans (kế hoạch đảm bảo chất lượng phần mềm)

## Những Điểm Chính (Key Points)

- Các phương pháp phát triển hợp tác (collaborative development practices) thường phát hiện tỷ lệ lỗi cao hơn so với kiểm thử (testing) và hiệu quả hơn về chi phí.
- Các phương pháp này cũng phát hiện các loại lỗi khác với kiểm thử, do đó cần phối hợp reviews (xem xét) và testing để đảm bảo chất lượng phần mềm.
- Inspections chính thức sử dụng checklist, chuẩn bị bài, phân vai rõ ràng và cải tiến quy trình liên tục để tối đa hóa hiệu quả phát hiện lỗi; thường phát hiện nhiều lỗi hơn walk-throughs (kiểm tra qua).
- **Pair programming** thường tiêu tốn chi phí tương đương inspections và tạo ra mã code có chất lượng tương tự; đặc biệt hữu ích khi cần rút ngắn tiến độ dự án.
- Inspections chính thức có thể áp dụng với nhiều sản phẩm như requirements (yêu cầu), designs (thiết kế), test cases (tập kiểm thử), cũng như code.
- Walk-through và code reading là các phương án thay thế cho inspections, giúp linh hoạt hơn trong việc sử dụng thời gian của từng thành viên.

## Chương 22: Developer Testing

### Nội dung chính

- **22.1 Vai trò của Developer Testing trong phần mềm:** trang 500
- **22.2 Cách tiếp cận khuyến nghị:** trang 503
- **22.3 Bộ công cụ kiểm thử:** trang 505
- **22.4 Các lỗi thường gặp:** trang 517
- **22.5 Công cụ hỗ trợ kiểm thử:** trang 523
- **22.6 Cải tiến kiểm thử:** trang 528
- **22.7 Ghi chép kết quả kiểm thử:** trang 529

### Chủ đề liên quan

- Bức tranh tổng thể về chất lượng phần mềm: Chương 20
- Thực hành xây dựng phần mềm cộng tác: Chương 21
- Gỡ lỗi: Chương 23
- Tích hợp phần mềm: Chương 29
- Điều kiện tiên quyết cho phát triển: Chương 3

### Tổng quan về Testing

Kiểm thử là hoạt động phổ biến nhất để cải thiện chất lượng, được hậu thuẫn bởi phong phú các nghiên cứu công nghiệp và học thuật cũng như kinh nghiệm thương mại thực tiễn. Phần mềm được kiểm thử theo nhiều cách, một số do developer thực hiện, số khác bởi các chuyên viên kiểm thử:

- **Unit testing:** Kiểm thử một class, routine, hoặc chương trình nhỏ do một nhóm lập trình viên tạo ra, thực hiện riêng biệt với hệ thống lớn hơn.
- **Component testing:** Kiểm thử một class/package/chương trình nhỏ/cấu phần do nhiều nhóm thực hiện, trong môi trường tách biệt hệ thống lớn hơn.
- **Integration testing:** Kiểm thử tổ hợp của hai hoặc nhiều class, package, thành phần, hoặc subsystem (phân hệ) do nhiều nhóm lập trình viên phát triển, bắt đầu khi xuất hiện ít nhất hai class để kiểm thử.
- **Regression testing:** Lặp lại các test case đã từng chạy để phát hiện defect mới xuất hiện trong phần mềm vốn đã vượt qua các thử nghiệm trước đó.
- **System testing:** Kiểm thử phần mềm ở cấu hình cuối cùng, tích hợp với các hệ thống phần mềm và phần cứng khác; đánh giá các khía cạnh như security (bảo mật), performance (hiệu năng), resource loss (mất tài nguyên), timing (vấn đề)

thời gian), v.v.

**Lưu ý:** Trong chương này, "testing" đề cập chủ yếu đến kiểm thử của developer, gồm unit, component, integration tests, và đôi khi gồm regression cũng như system tests. Nhiều loại kiểm thử khác như beta test, acceptance test, performance test, v.v. chủ yếu do bộ phận chuyên trách test thực hiện và không đề cập thêm ở đây.

---

## Phân loại kiểm thử

Testing chia thành hai nhóm lớn:

- **Black-box testing:** Tester không biết nội dung bên trong đối tượng được kiểm thử.
- **White-box testing** (hay glass-box testing): Tester trực tiếp biết rõ nội dung bên trong của đối tượng.

**Lưu ý:** Khi kiểm thử với code do mình viết thì trường hợp black-box không thực sự khả thi.

---

## So sánh giữa Testing và Debugging

Một số lập trình viên dùng lẫn hai khái niệm này nhưng thực tế chúng khác nhau:

- **Testing:** Phát hiện lỗi (error detection)
- **Debugging:** Tìm nguyên nhân và sửa lỗi đã phát hiện (error correction)

Chương này chỉ tập trung vào phát hiện lỗi. Việc sửa lỗi được thảo luận ở chương 23 "Debugging".

---

## Vai trò của Developer Testing liên quan đến Chất lượng Phần mềm

Các hoạt động phát triển cộng tác, ví dụ như **peer review** hoặc **inspection** thường phát hiện tỉ lệ lỗi cao hơn và tiết kiệm hơn so với chỉ testing. Các bước testing riêng lẻ (unit, component, integration) thường chỉ phát hiện dưới 50% số lỗi thực tế. Kết hợp nhiều bước kiểm thử cũng chỉ phát hiện khoảng dưới 60% tổng lỗi (Jones 1998).

**Chú thích thú vị:**

"Programs do not acquire bugs as people acquire germs, by hanging around other buggy programs. Programmers must insert them." — Harlan Mills

---

**Một số đặc điểm của Testing mà lập trình viên cần lưu ý:**

- **Mục tiêu:** Testing nhằm tìm ra lỗi – đi ngược lại với các hoạt động phát triển khác vốn nhằm tránh lỗi.
- **Kiểm thử không thể chứng minh không còn lỗi:** Dù đã phát hiện hàng nghìn lỗi, vẫn không chắc đã hết lỗi hoặc còn vô số lỗi chưa phát hiện.
- **Testing không tự cải thiện chất lượng:** Kết quả test cho thấy chất lượng phần mềm, nhưng chỉ test nhiều hơn thì không giúp tăng chất lượng; điều cốt lõi là phải có quy trình phát triển tốt.
- **Testing yêu cầu thái độ đúng:** Lập trình viên phải thực sự *mong* phát hiện được lỗi khi kiểm thử ("mong tìm lỗi ở code của mình").

Ví dụ nổi tiếng:

Trong một nghiên cứu, lập trình viên kiểm thử một chương trình có 15 lỗi đã biết, trung bình chỉ phát hiện được 5, thậm chí người phát hiện nhiều nhất cũng chỉ là 9 lỗi. Đa số lỗi còn sót do không kiểm tra cẩn thận output của chương trình.

---

## Thời lượng dành cho Developer Testing

Một con số thường được nhắc tới là kiểm thử chiếm 50% tổng thời gian dự án, tuy nhiên số này vừa gộp cả testing lẫn debugging, lẫn các hoạt động kiểm thử độc lập lẫn kiểm thử của developer.

Thực tế, thời gian developer testing nên chiếm khoảng **8–25% tổng thời gian dự án**, tùy thuộc vào quy mô và độ phức tạp dự án.

---

## Sử dụng kết quả kiểm thử

Kết quả kiểm thử dùng để:

1. Đánh giá độ tin cậy của phần mềm.
2. Hướng dẫn hoạt động sửa lỗi tiếp theo.

3. Tổng hợp lâu dài để nhận diện dạng lỗi phổ biến, từ đó định hướng đào tạo/chọn lựa review/test về sau.

---

### Kiểm thử trong giai đoạn xây dựng (construction)

Mặc dù class nên được thiết kế như một black box (người dùng chỉ quan tâm đến interface), khi test thì lợi thế lớn khi “nhìn xuyên hộp kính” để kiểm tra sâu nội dung mã nguồn. Tuy nhiên, người viết code cũng có những “mù điểm” tương tự khi test, nên cần phối hợp cả hai cách.

#### Khuyến cáo:

Nên test kỹ từng routine/class trước khi tích hợp thêm với module khác. Việc này không làm giảm độ khó kiểm thử nhưng giúp dễ dàng xác định và sửa lỗi hơn.

---

## 22.2 Khuyến nghị cách tiếp cận Developer Testing

### Cách tiếp cận hệ thống giúp phát hiện nhiều loại lỗi với nỗ lực tối thiểu:

- Kiểm thử mỗi requirement (yêu cầu) liên quan để đảm bảo requirement đã được hiện thực.  
Lên kế hoạch test case ngay từ giai đoạn lấy requirements hoặc càng sớm càng tốt – lý tưởng là trước khi viết code.
- Kiểm thử mỗi yếu tố thiết kế liên quan để đảm bảo thiết kế đã được hiện thực.  
Lên kế hoạch test ngay từ giai đoạn thiết kế (trước khi code chi tiết).
- Sử dụng “basis testing” để bổ sung chi tiết test case và kết hợp kiểm thử output với luồng dữ liệu (data-flow tests).
- Sử dụng checklist thống kê các loại lỗi thường gặp trong dự án hiện tại hoặc trước đây.

#### Ghi chú:

Lên kế hoạch kiểm thử càng sớm càng tốt sẽ tiết kiệm chi phí sửa lỗi về sau.

---

### Test First hay Test Last?

Một số lập trình viên phân vân nên viết test case trước hay sau khi code.  
Các nghiên cứu chỉ ra rằng việc viết test case trước sẽ:

- Không tốn thêm công sức mà chỉ thay đổi trình tự công việc.
- Phát hiện và sửa lỗi sớm hơn.
- Buộc phải nghĩ về requirement và design trước khi code, thường tạo ra code tốt hơn.
- Phát hiện requirement kém trước khi code.
- Nếu lưu trữ test case, vẫn có thể kiểm thử lại sau – bên cạnh kiểm thử trước.

**Kết luận:** Programming theo "test first" nên áp dụng tổng quát khi có thể.

---

## Hạn chế của Developer Testing

- **Thiên về clean test:** Lập trình viên thường kiểm thử các trường hợp 'code hoạt động đúng' hơn là kiểm thử các trường hợp 'code gặp trục trặc' (dirty test).
  - Tỷ lệ phổ biến: tổ chức mới phát triển 5 clean test : 1 dirty test; tổ chức trưởng thành là ngược lại (tạo nhiều dirty test hơn rất nhiều).
- **Tự tin quá về độ phủ kiểm thử:** Lập trình viên thường tin là đạt 95% coverage (độ phủ kiểm thử), thực tế chỉ đạt 50–60% trong trường hợp tốt, thậm chí thấp hơn nữa.
- **Ít áp dụng các kỹ thuật coverage nâng cao:** Đa phần chỉ đạt "100% statement coverage" (từng dòng code đều được chạy qua), trong khi "100% branch coverage" (mỗi điều kiện rẽ nhánh đều được kiểm thử cả true lẫn false) mới là tiêu chuẩn tốt hơn.

**Nhận định:** Developer testing cực kỳ quan trọng nhưng không đầy đủ; nên phối hợp với kiểm thử độc lập và các phương pháp xây dựng hợp tác (collaborative construction techniques).

---

## 22.3 Bộ công cụ kiểm thử



**Vì sao không thể chứng minh chương trình đúng hoàn toàn chỉ bằng kiểm thử?** Để kiểm thử triệt để, ta phải thử mọi giá trị input có thể, cũng như mọi kết hợp có thể của các giá trị input – điều này gần như bất khả thi kể cả với chương trình đơn giản.

**Ví dụ:** Một chương trình nhập tên, địa chỉ, số điện thoại – mỗi tên/địa chỉ 20 kí tự, mỗi ký tự có 26 lựa chọn; số điện thoại 10 chữ số, mỗi chữ số có 10 khả năng. Vậy:

Name:  $26^{20}$   
Address:  $26^{20}$   
Phone Number:  $10^{10}$   
Total Possibilities =  $26^{20} * 26^{20} * 10^{10} \approx 10^{66}$

Dẫu mỗi giây chạy được 1 nghìn tỷ ca kiểm thử, thì sau hàng triệu năm vẫn chưa kiểm đủ mọi trường hợp.

*Hoàn tất phần dịch. Nếu cần dịch thêm cụ thể các phần khác, vui lòng yêu cầu chi tiết!*

## Kiểm Thử Không Đầy Đủ (Incomplete Testing)

### Tham Chiếu Chéo (Cross-Reference)

Bởi vì việc kiểm thử toàn diện (exhaustive testing) gần như không khả thi trong thực tiễn, nghệ thuật kiểm thử nằm ở chỗ xác định xem bạn đã bao phủ toàn bộ mã nguồn (code) hay chưa, chủ yếu bằng cách lựa chọn các trường hợp kiểm thử (test cases) có khả năng cao phát hiện lỗi. Trong số 1066 trường hợp kiểm thử có thể có, chỉ một vài trường hợp thực sự giúp phát hiện ra những lỗi mà các trường hợp còn lại không phát hiện được. Do đó, bạn nên tập trung vào việc chọn một số ít trường hợp kiểm thử có khả năng cung cấp thông tin khác nhau thay vì trùng lặp.

Khi lập kế hoạch kiểm thử, hãy loại bỏ những trường hợp không mang lại giá trị bổ sung — đặc biệt là các kiểm thử trên dữ liệu mới mà rất có thể sẽ không làm phát sinh lỗi nếu các dữ liệu tương tự trước đó cũng không gây ra lỗi.

Các phương pháp được đề xuất để tối ưu hóa việc bao phủ được trình bày trong các mục tiếp theo.

### Kiểm Thử Cơ Sở Có Cấu Trúc (Structured Basis Testing)

Mặc dù tên gọi có vẻ phức tạp, kiểm thử cơ sở có cấu trúc thực chất là một khái niệm đơn giản: **Cần kiểm thử mỗi lệnh trong chương trình ít nhất một lần.** Nếu lệnh đó là lệnh điều kiện (như `if`, `while`), kiểm thử phải đa dạng hóa theo đúng các điều kiện bên trong để đảm bảo việc kiểm thử toàn diện.

Cách đơn giản để đảm bảo mọi tình huống được kiểm thử là tính toán số đường đi (path) qua chương trình và xây dựng bộ tối thiểu các trường hợp kiểm thử để kiểm soát mọi đường đi đó. Phương pháp kiểm thử bao phủ mã lệnh (code coverage testing) hoặc kiểm thử bao phủ logic (logic coverage testing) cũng hướng tới mục tiêu này, tuy nhiên chúng không tối ưu số lượng kiểm thử như kiểm thử cơ sở có cấu trúc.

#### Cách Tính Số Lượng Trường Hợp Kiểm Thử Tối Thiểu

- Bắt đầu với **1** cho đường đi cơ bản qua hàm.
- Cộng thêm **1** cho mỗi từ khóa (hoặc tương đương): `if`, `while`, `repeat`, `for`, `and`, `or`.
- Cộng thêm **1** cho mỗi trường hợp (case) trong câu lệnh `case`. Nếu không có trường hợp mặc định (default), cộng thêm 1 nữa.

**Ví dụ đơn giản:**

```
Statement1;  
Statement2;  
if (x < 10) {  
    Statement3;  
}  
Statement4;
```

Ở đây, bắt đầu với một và cộng thêm một cho lệnh `if`, tổng là hai. Như vậy, phải có ít nhất **hai trường hợp kiểm thử**:

- Điều kiện `if` đúng ( $x < 10$ )
- Điều kiện `if` sai ( $x \geq 10$ )

**Ví dụ phức tạp hơn:**

```
// Compute Net Pay
totalWithholdings = 0;

for (id = 0; id < numEmployees; id++) {

    // compute social security withholding, if below the maximum
    if (m_employee[id].governmentRetirementWithheld < MAX_GOVT_RETIREMENT) {
        governmentRetirement = ComputeGovernmentRetirement(m_employee[id]);
    }

    // set default to no retirement contribution
    companyRetirement = 0;

    // determine discretionary employee retirement contribution
    if (m_employee[id].WantsRetirement &&
        EligibleForRetirement(m_employee[id])) {
        companyRetirement = GetRetirement(m_employee[id]);
    }

    grossPay = ComputeGrossPay(m_employee[id]);

    // determine IRA contribution
    personalRetirement = 0;
    if (EligibleForPersonalRetirement(m_employee[id])) {
        personalRetirement = PersonalRetirementContribution(m_employee[id], companyRetirement, grossPay);
    }

    // make weekly paycheck
    withholding = ComputeWithholding(m_employee[id]);
    netPay = grossPay - withholding - companyRetirement - governmentRetirement - personalRetirement;
    PayEmployee(m_employee[id], netPay);

    // add this employee's paycheck to total for accounting
    totalWithholdings = totalWithholdings + withholding;
    totalGovernmentRetirement = totalGovernmentRetirement + governmentRetirement;
    totalRetirement = totalRetirement + companyRetirement;
}

SavePayRecords(totalWithholdings, totalGovernmentRetirement, totalRetirement);
```

Trong ví dụ này, có năm từ khóa quan trọng (`for`, ba `if`, một `&&`), cộng với đường đi cơ sở, tổng cộng là **sáu trường hợp kiểm thử tối thiểu**. Tuy nhiên, sáu trường hợp này cần xây dựng hợp lý để thực sự kiểm soát hết mọi trường hợp.

#### Bảng trường hợp kiểm thử mẫu:

Case	Mô tả	Dữ liệu kiểm thử
1	Trường hợp danh nghĩa	Mọi điều kiện boolean đều đúng
2	Điều kiện <code>for</code> sai	<code>numEmployees &lt; 1</code>
3	Điều kiện <code>if</code> đầu tiên sai	<code>m_employee[id].governmentRetirementWithheld &gt;= MAX_GOVT_RETIREMENT</code>
4	Điều kiện <code>if</code> thứ hai, phần 1 sai	<code>not m_employee[id].WantsRetirement</code>
5	Điều kiện <code>if</code> thứ hai, phần 2 sai	<code>not EligibleForRetirement(m_employee[id])</code>
6	Điều kiện <code>if</code> thứ ba sai	<code>not EligibleForPersonalRetirement(m_employee[id])</code>

Như vậy, số lượng kiểm thử tăng nhanh khi chương trình phức tạp hơn. Việc chia nhỏ hàm và giữ biểu thức boolean đơn giản giúp kiểm thử dễ dàng hơn.

## Kiểm Thử Dòng Dữ Liệu (Data-Flow Testing)

Kiểm thử dòng dữ liệu dựa trên ý tưởng rằng sử dụng dữ liệu cũng dễ gây lỗi như dòng điều khiển (control flow). Dữ liệu có thể ở ba trạng thái:

- **Đã định nghĩa (Defined):** Được khởi tạo nhưng chưa sử dụng.
- **Đã sử dụng (Used):** Được dùng trong tính toán hoặc truyền tham số.
- **Đã bị hủy (Killed):** Đã được "hủy" bằng cách nào đó (ví dụ: pointer đã được giải phóng).

Ngoài ra, còn xuất hiện thuật ngữ liên quan đến phạm vi hàm:

- **Khi vào hàm (Entered):** Dòng điều khiển vào trước khi biến được sử dụng.
- **Khi thoát hàm (Exited):** Sau khi biến được thao tác xong thì thoát khỏi hàm.

### Các tổ hợp trạng thái dòng dữ liệu bất thường

- **Defined-Defined:** Định nghĩa hai lần trước khi sử dụng là lãng phí và dễ lỗi.
- **Defined-Exited:** Biến cục bộ mà không dùng đến thì không hợp lý.
- **Defined-Killed:** Biến được định nghĩa rồi hủy mà không sử dụng chứng tỏ thừa/chưa dùng.
- **Entered-Killed:** Không hợp lý với biến cục bộ (chưa định nghĩa đã giết).
- **Entered-Used:** Dùng biến cục bộ chưa định nghĩa là lỗi logic.
- **Killed-Killed:** Giải phóng hai lần là nguy hiểm (pointer "double free").
- **Killed-Used:** Sử dụng sau khi đã bị hủy là lỗi logic.
- **Used-Defined:** Nếu dùng trước rồi lại định nghĩa thì cần kiểm soát điểm định nghĩa trước đó.

Kiểm tra kỹ các chuỗi trạng thái bất thường trước khi kiểm thử. Sau đó, tạo bộ kiểm thử đủ để bao phủ mọi đường từ điểm định nghĩa đến điểm sử dụng (defined-used path).

### Ví dụ kiểm thử dòng dữ liệu đơn giản:

```
if (Condition1) {  
    x = a;  
} else {  
    x = b;  
}  
  
if (Condition2) {  
    y = x + 1;  
} else {  
    y = x - 1;  
}
```

Để bao phủ tất cả đường kiểm thử của cấu trúc cơ sở đã đủ hai trường hợp, nhưng để bao phủ mọi kết hợp defined-used thì cần thêm:

- x = a, rồi y = x - 1
- x = b, rồi y = x + 1

Tổng cộng, cần **bốn trường hợp kiểm thử** để bao phủ mọi đường dữ liệu.

---

## Phân Vùng Tương Đương (Equivalence Partitioning)

**Equivalence partitioning** là kỹ thuật chia không gian dữ liệu đầu vào thành các lớp tương đương (equivalence class) sao cho kiểm thử một giá trị đại diện cho cả lớp. Điều này giúp giảm số lượng kiểm thử cần thiết mà vẫn đảm bảo kiểm soát được tất cả lỗi.

Ví dụ:

```
if (m_employee[ID].governmentRetirementWithheld < MAX_GOVT_RETIREMENT) { ... }
```

Có hai lớp tương đương: nhỏ hơn và lớn hơn/hoặc bằng MAX\_GOVT\_RETIREMENT.

Kỹ thuật này phát huy hiệu quả lớn khi làm việc từ đặc tả (specification) hoặc với dữ liệu phức tạp chưa thể hiện hết trong chương trình.

---

## Phán Đoán Lỗi (Error Guessing)

Bên cạnh các kỹ thuật hình thức, lập trình viên giỏi còn sử dụng các kinh nghiệm thực tiễn và trực giác để "phán đoán lỗi". Phương pháp này dựa trên việc tạo ra các kiểm thử ở những chỗ nghi ngờ dễ phát sinh lỗi, dựa vào kinh nghiệm trước đó, thống kê lỗi hay gặp...

### Phân Tích Biên (Boundary Analysis)

Kiểm thử biên tập trung vào các lỗi liên quan đến giá trị biên (off-by-one). Ví dụ, nếu kiểm tra giới hạn max, cần ba trường hợp: ngay dưới biên, đúng biên, và ngay trên biên.

Ví dụ trong kiểm thử: | Case | Mô tả | |-----|-----| | 1 | Dưới biên: m\_employee[ID].governmentRetirementWithheld = MAX\_GOVT\_RETIREMENT - 1 | | 3 | Trên biên: m\_employee[ID].governmentRetirementWithheld = MAX\_GOVT\_RETIREMENT + 1 | | 10 | Đúng bằng biên: m\_employee[ID].governmentRetirementWithheld = MAX\_GOVT\_RETIREMENT |

**Lưu ý:** Phép phân tích biên cũng áp dụng với nhóm giá trị lớn hoặc nhỏ bất thường (compound boundaries), ví dụ kiểm tra nhiều nhân viên có mức lương rất cao hoặc cực thấp.

---

## Tóm tắt và khuyến nghị

- Không cần kiểm thử cạn kiệt, tập trung vào các trường hợp có xác suất xuất hiện lỗi cao.
- Áp dụng kết hợp các kỹ thuật: kiểm thử cơ sở có cấu trúc, kiểm thử dòng dữ liệu, phân vùng tương đương, phán đoán lỗi và kiểm thử biên để tối ưu bộ kiểm thử.
- Ghi nhận các lỗi phổ biến và phát triển kỹ năng phán đoán dựa trên chúng.

**Ghi chú:** Các bảng và ví dụ về trường hợp kiểm thử sẽ tiếp tục được mở rộng trong chương này.

## 00. Các Loại Dữ Liệu Không Chất Lượng (Bad Data)

Bên cạnh việc dự đoán lỗi xuất hiện tại các điều kiện biên (boundary conditions), bạn cũng có thể dự đoán và kiểm thử cho một số lớp dữ liệu không chất lượng (bad data) khác. Các trường hợp kiểm thử điển hình đối với dữ liệu không chất lượng bao gồm:

- Quá ít dữ liệu (hoặc không có dữ liệu)
- Quá nhiều dữ liệu
- Dữ liệu sai kiểu (invalid data)
- Dữ liệu sai kích thước
- Dữ liệu chưa được khởi tạo (uninitialized data)

Một số trường hợp kiểm thử mà bạn có thể nghĩ ra khi tuân theo các gợi ý trên đã được đề cập trước đó. Ví dụ, “quá ít dữ liệu” đã được đề cập trong Trường hợp 2 và 12, còn đối với “dữ liệu sai kích thước” thì khá khó để nghĩ ra trường hợp cụ thể. Tuy nhiên, việc phân loại dữ liệu không chất lượng vẫn tạo ra một số trường hợp mới:

Trường hợp	Mô tả kiểm thử
13	Một mảng gồm 100.000.000 nhân viên. Kiểm tra cho trường hợp quá nhiều dữ liệu. Tất nhiên, “quá nhiều” sẽ thay đổi tùy theo hệ thống, nhưng trong ví dụ này, coi như đây là quá mức cho phép.
14	Mức lương âm. Dữ liệu sai kiểu.
15	Số lượng nhân viên âm. Dữ liệu sai kiểu.

## Các Lớp Dữ Liệu Chất Lượng (Good Data)

Khi cố gắng tìm lỗi trong chương trình, bạn có thể dễ dàng bỏ qua khả năng trường hợp danh nghĩa (nominal case) có thể chứa lỗi. Thông thường, các trường hợp danh nghĩa trong phần kiểm thử cơ sở đại diện cho một loại dữ liệu chất lượng. Dưới đây là các loại dữ liệu tốt khác rất đáng kiểm thử. Việc kiểm tra từng loại dữ liệu này có thể tiết lộ lỗi, tùy thuộc vào đối tượng kiểm thử:

- Trường hợp danh nghĩa — giá trị trung bình, như mong đợi
- Cấu hình tối thiểu bình thường (minimum normal configuration)
- Cấu hình tối đa bình thường (maximum normal configuration)
- Tính tương thích với dữ liệu cũ

### Cấu hình tối thiểu bình thường

Cấu hình này hữu ích để kiểm thử không chỉ một, mà là một nhóm đối tượng. Nó giống về mặt tinh thần với điều kiện biên tại các giá trị nhỏ nhất, nhưng khác ở chỗ nó hình thành bộ giá trị tối thiểu từ những giá trị mà hệ thống thường kỳ vọng. Ví dụ, lưu một bảng tính trống khi kiểm thử phần mềm bảng tính; với phần mềm xử lý văn bản, là lưu một tài liệu trống. Trong ví dụ minh họa, kiểm thử cấu hình tối thiểu bình thường sẽ thêm trường hợp sau:

Trường hợp	Mô tả kiểm thử
16	Một nhóm gồm một nhân viên. Kiểm thử cho cấu hình tối thiểu bình thường.

### Cấu hình tối đa bình thường

Ngược lại với cấu hình tối thiểu. Đây là việc hình thành một bộ giá trị tối đa dựa trên các giá trị được mong đợi. Ví dụ, lưu một bảng tính với kích thước lớn nhất được đề cập trên bao bì sản phẩm, hoặc in bảng tính kích thước tối đa đó. Với phần mềm xử lý văn bản, là lưu một tài liệu với dung lượng lớn nhất được khuyến nghị. Giả sử số lượng nhân viên tối đa bình thường là 500, bạn sẽ thêm trường hợp kiểm thử sau:

Trường hợp	Mô tả kiểm thử
------------	----------------

## Trường hợp

## Mô tả kiểm thử

17

Nhóm gồm 500 nhân viên. Kiểm thử cho cấu hình tối đa bình thường.

### Tính tương thích với dữ liệu cũ

Kiểm thử dạng này áp dụng khi chương trình hoặc routine (đoạn mã) mới thay thế cho một chương trình hoặc đoạn mã cũ. Routine mới nên cho kết quả tương tự với dữ liệu cũ như routine cũ (trừ khi routine cũ từng có lỗi). Loại liên tục này chính là cơ sở của kiểm thử hồi quy (regression testing) — nhằm đảm bảo các chỉnh sửa, nâng cấp không làm giảm chất lượng đã đạt. Trong ví dụ đang xét, tiêu chí tương thích không tạo ra trường hợp kiểm thử mới nào.

## Sử dụng Trường hợp Kiểm thử thuận tiện cho Kiểm tra Thủ công

Giả sử bạn viết một trường hợp kiểm thử cho một mức lương danh nghĩa, cách bạn chọn mức lương là gõ một dãy số bất kỳ. Ví dụ:

1239078382346

Đây là một mức lương rất lớn, hơn một ngàn tỷ đô la, nhưng nếu chỉnh lại cho thực tế, chỉ còn \$90,783.82.

Giả dụ trường hợp kiểm thử này phát hiện ra lỗi. Làm sao bạn biết đã tìm thấy lỗi? Lý do là bạn biết đáp án mong muốn do đã tính tay. Tuy nhiên, nếu phải tính toán thủ công với những con số khó như \$90,783.82, thì khả năng mắc lỗi thủ công cũng cao không kém khả năng phát hiện lỗi trong chương trình. Ngược lại, chọn con số tròn như \$20,000 sẽ giúp việc kiểm tra số học dễ dàng, giảm khả năng nhầm lẫn.

Có thể bạn nghĩ những con số khó như \$90,783.82 sẽ dễ phát hiện lỗi hơn, nhưng thực tế, chúng không mang lại xác suất phát hiện lỗi cao hơn so với các số nằm trong cùng một nhóm tương đương (equivalence class).

## 22.4 Các Lỗi Thường Gặp (Typical Errors)

### Kiến Thức về Các Lớp chứa Nhiều Lỗi Nhất

Thông thường, người ta cho rằng lỗi phân bố đều trên toàn bộ mã nguồn. Ví dụ, nếu trung bình có 10 lỗi trên 1000 dòng mã, bạn có thể nghĩ rằng một lớp chứa 100 dòng mã sẽ có một lỗi. Tuy nhiên, **giả định này là sai**.

Theo báo cáo của Capers Jones, một chương trình cải thiện chất lượng tại IBM phát hiện 31 trong số 425 lớp của hệ thống IMS là nhạy cảm với lỗi. Sau khi sửa đổi hoặc phát triển lại 31 lớp đó, chỉ sau chưa đầy một năm, số lượng lỗi khách hàng báo về đã giảm tỷ lệ mười trên một. Chi phí bảo trì tổng thể giảm khoảng 45%. Mức độ hài lòng của khách hàng chuyển từ “không chấp nhận được” lên “tốt”.

**Kết luận:** Hầu hết lỗi tập trung ở một số routine (đoạn mã) thường xuyên gây lỗi.

### Mối quan hệ giữa lỗi và mã nguồn

- 80% lỗi nằm trong 20% các lớp hoặc routine** của dự án (Endres 1975, Gremillion 1984, Boehm 1987b, Shull et al 2002).
- 50% lỗi nằm trong 5% các lớp** của dự án (Jones 2000).

### Hệ quả:

- 20% routine của dự án chiếm tới 80% chi phí phát triển (Boehm 1987b). Không nhất thiết các routine này có nhiều lỗi nhất, nhưng đây là dấu hiệu quan trọng.
- Những routine có nhiều lỗi thường là những routine cực kỳ tốn kém.
- Các hoạt động bảo trì nên tập trung xác định, thiết kế lại và viết lại hoàn toàn các routine đó.

### Phân loại lỗi

Nhiều nhà nghiên cứu đã cố gắng phân loại lỗi theo loại và xác định tần suất xảy ra của từng loại lỗi. Mỗi lập trình viên đều có một danh sách lỗi thường gặp: lỗi chênh lệch 1 đơn vị (off-by-one errors), quên khởi tạo biến vòng lặp, v.v.

Theo phân loại của Boris Beizer:

- 25,18% Lỗi cấu trúc (structural)
- 22,44% Lỗi dữ liệu (data)
- 16,19% Chức năng được thực thi (functionality as implemented)
- 9,88% Lỗi xây dựng (construction)
- 8,98% Tích hợp (integration)
- 8,12% Yêu cầu chức năng (functional requirements)
- 2,76% Định nghĩa hoặc thực thi kiểm thử
- 1,74% Kiến trúc hệ thống, phần mềm
- 4,71% Không xác định

Lưu ý: Các nghiên cứu khác nhau đưa ra kết quả rất khác biệt về tỷ lệ từng loại lỗi. Dù không mang giá trị tuyệt đối, các số liệu này vẫn mang tính gợi ý.

### Một số nhận định từ dữ liệu:

- **Phạm vi sửa lỗi hạn chế:** 85% lỗi có thể khắc phục mà không cần sửa nhiều hơn một routine (Endres 1975).
- **Phần lớn lỗi không thuộc về giai đoạn xây dựng:** Nguồn gốc lớn nhất của lỗi là: kiến thức về lĩnh vực ứng dụng còn hạn chế, yêu cầu không ổn định hoặc mâu thuẫn, lỗi giao tiếp và phối hợp (Curtis, Krasner, Iscoe 1988).
- **Đa phần lỗi lập trình là do lập trình viên:** Khoảng 95% lỗi là do lập trình viên, 2% do hệ điều hành hoặc trình biên dịch, 2% do phần mềm khác, 1% do phần cứng (Brown và Sampson 1973, Ostrand và Weyuker 1984).
- **Lỗi nhập liệu (typos) phổ biến:** 36% lỗi xây dựng là do nhập liệu sai (Weiss 1975). Trong phần mềm động lực học bay gần 3 triệu dòng, 18% lỗi là do nhập liệu (Card 1987). Ba lỗi phần mềm tốn kém nhất lịch sử đều chỉ do một ký tự thay đổi.
- **Hiểu sai thiết kế là nguyên nhân lặp lại:** 16-19% lỗi do hiểu lầm thiết kế (Beizer 1990, Weiss 1975).
- **Hầu hết lỗi dễ sửa:** Khoảng 85% lỗi có thể sửa dưới vài giờ, 15% sửa trong vài giờ tới vài ngày, chỉ 1% lâu hơn (Weiss 1975, Ostrand và Weyuker 1984, Grady 1992).
- **Đo lường lỗi trong tổ chức của bạn:** Kinh nghiệm giữa các tổ chức rất đa dạng, cần tự đo lường quy trình phát triển của mình để xác định vấn đề.

## Tỷ lệ lỗi xuất phát từ giai đoạn xây dựng (construction)

- **Dự án nhỏ:** 75% lỗi do giai đoạn lập trình, 10% do yêu cầu, 15% từ thiết kế (Jones 1986a).
- **Dự án lớn:** Lỗi xây dựng chiếm ít nhất 35% tổng số lỗi, có thể tới 75% trên dự án rất lớn (Beizer 1990, Jones 2000, Grady 1987).

Mặc dù lỗi lập trình rẻ hơn lỗi ở giai đoạn yêu cầu và thiết kế, tổng chi phí sửa lỗi lập trình vẫn rất lớn. Trung bình, chi phí sửa một lỗi lập trình bằng 25–50% chi phí sửa một lỗi thiết kế; tổng chi phí sửa lỗi lập trình bằng 1–2 lần tổng chi phí sửa lỗi thiết kế (Grady 1987).

## Số lượng lỗi nên kỳ vọng tìm được

- **Trung bình ngành:** 1–25 lỗi/1000 dòng mã với phần mềm giao sản phẩm (Boehm 1981, Gremillion 1984, Yourdon 1989a, Jones 1998, Jones 2000, Weber 2003).
- **Microsoft:** 10–20 lỗi/1000 dòng mã khi kiểm thử nội bộ; 0.5 lỗi/1000 dòng mã trên sản phẩm phát hành (Moore 1992).
- **Cleanroom development:** 3 lỗi/1000 dòng mã cho phần mềm nội bộ, 0.1 lỗi/1000 dòng cho phần mềm đã phát hành (Cobb và Mills 1990).
- **Team Software Process (TSP):** 0.06 lỗi/1000 dòng mã (Weber 2003).

**Nguyên lý:** Xây dựng phần mềm chất lượng tốt ban đầu rẻ hơn rất nhiều so với việc làm ra phần mềm kém và phải sửa chữa.

## Lỗi Trong Chính Quá Trình Kiểm Thử

Bạn có thể từng trải qua trường hợp, phần mềm có lỗi, bạn đoán một vài đoạn mã có thể sai nhưng khi kiểm tra đều đúng. Sau nhiều lần kiểm thử, lại không thấy lỗi nữa. Cho đến khi kiểm tra lại dữ liệu kiểm thử — mới phát hiện lỗi nằm ở dữ liệu kiểm thử, không phải trong mã!

**Điểm mấu chốt:** Trường hợp kiểm thử có thể chứa lỗi ngang hoặc nhiều hơn so với mã cần kiểm thử (Weiland 1983, Jones 1986a, Johnson 1994).

**Lý do:** Trường hợp kiểm thử thường được tạo nhanh, đôi khi xem như tạm thời, không qua quy trình thiết kế hoặc kiểm tra cẩn thận như mã nguồn. Để hạn chế lỗi trong kiểm thử, thực hiện các nguyên tắc sau:

- Kiểm tra kỹ lưỡng trường hợp kiểm thử như với mã nguồn sản xuất; bao gồm kiểm tra lại, bước qua từng dòng bằng trình gỡ lỗi, kiểm duyệt dữ liệu kiểm thử.
- Lập kế hoạch kiểm thử song song với phát triển phần mềm từ giai đoạn yêu cầu, tránh trường hợp giả định sai ngay khi tạo trường hợp kiểm thử.
- Lưu lại các trường hợp kiểm thử để tái sử dụng cho kiểm thử hồi quy hoặc phát triển phiên bản mới.
- Tích hợp kiểm thử đơn vị (unit test) vào framework kiểm thử toàn hệ thống (như JUnit), điều này giúp tăng chất lượng và tránh việc bỏ quên hay lãng phí trường hợp kiểm thử.

## 5. Công Cụ Hỗ Trợ Kiểm Thử (Test-Support Tools)

### Tổng Quan Về Công Cụ Kiểm Thử

Phần này khảo sát các loại công cụ kiểm thử mà bạn có thể mua thương mại hoặc tự phát triển. Tài liệu không đề cập đến các sản phẩm cụ thể bởi chúng có thể nhanh chóng lỗi thời. Để cập nhật, hãy tham khảo các tạp chí lập trình yêu thích của bạn.

#### *Xây dựng Scaffolding để Kiểm thử Từng Lớp (Class)*

Thuật ngữ *scaffolding* (giàn giáo) xuất phát từ lĩnh vực xây dựng. Giàn giáo được thiết lập để công nhân tiếp cận những vị trí mà bình thường không thể vươn tới. Trong phần mềm, *scaffolding* được xây dựng nhằm phục vụ duy nhất cho việc kiểm thử mã nguồn.

**Tham khảo sâu hơn:** Xem bài luận “A Small Matter of Programming” của Jon Bentley trong cuốn *Programming Pearls, 2d ed. (2000)* để biết ví dụ về scaffolding.

Một loại scaffolding là một class (lớp) được mô phỏng (dummied up) để phục vụ cho class đang được kiểm thử. Class này được gọi là **mock object** hoặc **stub object** (đối tượng mô phỏng/đối tượng giả lập) (Mackinnon, Freemant, và Craig 2000; Thomas và Hunt 2002). Cách tiếp cận tương tự cũng được áp dụng với các routine cấp thấp, gọi là **stub routines** (thủ tục giả lập).

Mức độ hiện thực hóa của *mock object* hoặc *stub routine* tùy thuộc vào yêu cầu thực tế. Một số chức năng điển hình của scaffolding:

- **Trả quyền điều khiển ngay lập tức** mà không thực thi xử lý.
- **Kiểm tra dữ liệu đầu vào.**
- **In thông báo chẩn đoán** hoặc *echo* (phản hồi) các tham số đầu vào, ghi log vào tệp.
- **Nhận giá trị trả về từ đầu vào tương tác (interactive input).**
- **Trả về một đáp án cố định** bất kể đầu vào là gì.
- **Tiêu tốn số vòng lặp CPU tương đương đối tượng hoặc routine thật.**
- **Hoạt động như một bản sao chậm, công kênh, đơn giản hoặc kém chính xác** của đối tượng hoặc routine thực.

Một loại *scaffolding* khác là **driver** (trình điều khiển) hoặc **test harness** (bộ khung kiểm thử), thực chất là một routine giả mạo gọi tới routine thực tế để kiểm thử. Một số chức năng chính:

- Gọi object với bộ đầu vào cố định.
- Nhận đầu vào từ tương tác người dùng và gọi object.
- Đọc tham số từ dòng lệnh (nếu hệ điều hành hỗ trợ) và gọi object.
- Đọc tham số từ tệp và gọi object.
- Thực thi nhiều lần với bộ dữ liệu đầu vào định nghĩa trước.

**Lưu ý:** Ranh giới giữa công cụ kiểm thử và công cụ gỡ lỗi (debugging tool) khá mờ nhạt. Xem chi tiết trong phần 23.5 “Debugging Tools—Obvious and Not-So-Obvious”.

Một dạng khác là **dummy file** (tệp mô phỏng), phiên bản nhỏ gọn của tệp thật với cấu trúc tương tự. Lợi ích chính:

- Biết chính xác nội dung, đảm bảo tệp không bị lỗi.
- Dễ dàng phát hiện lỗi khi sử dụng tệp này trong kiểm thử.

Mặc dù xây dựng scaffolding tốn thêm công sức, nhưng có thể tái sử dụng khi phát hiện lỗi trong class. Hiện nay có nhiều công cụ hỗ trợ tạo nhanh *mock object* cũng như scaffolding.

Các bộ kiểm thử như JUnit, CppUnit, NUnit, v.v. cung cấp sẵn scaffolding cho chương trình của bạn. Nếu môi trường không được hỗ trợ, bạn có thể viết vài routine trong một class, bao gồm một hàm `main()` làm scaffolding, cho phép kiểm thử các routine riêng lẻ trước khi tích hợp.

Các routine kiểm thử có thể được giữ lại trong file mã nguồn và vô hiệu hoá bằng các lệnh tiền xử lý hoặc chú thích, đảm bảo không ảnh hưởng tới mã thực thi.

## Diff Tools (Công Cụ So Sánh)

**Tham khảo:** Xem “Retesting (Regression Testing)” trong phần 22.6.

Kiểm thử hồi quy (regression testing) trở nên dễ dàng hơn với công cụ tự động so sánh đầu ra thực tế với đầu ra mong đợi. Một cách cơ bản là chuyển hướng đầu ra sang tệp, rồi dùng công cụ so sánh tệp như `diff` để phát hiện sai lệch.

## Test-Data Generators (Trình Sinh Dữ Liệu Kiểm Thử)

Bạn cũng có thể tự viết code để kiểm thử các phân đoạn chương trình một cách hệ thống.

Ví dụ điển hình: Tác giả đã phát triển một thuật toán mã hóa độc quyền, sinh ra các tệp kiểm thử gồm chuỗi ký tự ngẫu nhiên kích thước từ 0K tới 500K, mật khẩu từ 1 tới 255 ký tự, kiểm thử cho nhiều trường hợp biên. Thống kê cho thấy, trung bình kiểm thử trên nhiều trường hợp đã phát hiện ra các lỗi đặc biệt, nâng cao độ tin cậy của chương trình.

**Một số bài học rút ra:**

- Bộ sinh dữ liệu ngẫu nhiên thiết kế tốt có thể tạo ra các trường hợp kiểm thử bất ngờ.
- Kiểm thử với dữ liệu ngẫu nhiên có độ bao phủ chương trình tốt hơn tự kiểm thử thủ công.
- Có thể điều chỉnh phân bố kiểm thử để phù hợp thực tế sử dụng.
- Thiết kế module tốt giúp kiểm thử dễ dàng hơn (có thể tách riêng chức năng mã hóa/giải mã khỏi giao diện người dùng).
- Có thể tái sử dụng test driver khi cần cập nhật chương trình.

## Coverage Monitors (Trình Giám Sát Độ Bao Phủ Mã Lệnh)

Theo Karl Wieggers (2002), kiểm thử không đo lường độ bao phủ thường chỉ kiểm tra được khoảng 50–60% mã nguồn. *Coverage monitor* (trình giám sát độ bao phủ) giúp xác định phần mã nào đã/ chưa được kiểm tra, từ đó bổ sung thêm các trường hợp kiểm thử cần thiết.

## Data Recorder / Logging (Ghi Nhật Ký/ Theo Dõi Dữ Liệu)

Các công cụ này theo dõi trạng thái chương trình, hỗ trợ chẩn đoán lỗi tương tự như “hộp đen” trên máy bay. Việc ghi nhận nhật ký sự kiện và trạng thái hệ thống trước khi lỗi xảy ra giúp phân tích nguyên nhân.

- Có thể tích hợp chức năng này trong phiên bản phát triển và loại bỏ khi phát hành chính thức.
- Có thể bỏ qua nếu sử dụng cơ chế lưu trữ tự động dọn dẹp (self-pruning storage) và xác định thông báo lỗi hợp lý.

## Symbolic Debuggers (Bộ Gỡ Lỗi Biểu Tượng)

**Tham khảo:** Xem phần 4.3, “Your Location on the Technology Wave”, về mức độ sẵn có của debugger.

*Symbolic debugger* hỗ trợ cho quy trình kiểm tra mã (walk-through, inspection), cho phép thực thi từng bước, theo dõi giá trị biến, xem song song mã nguồn ngôn ngữ bậc cao và mã máy, rất hữu ích để phòng tránh các “vùng mù” trong quá trình kiểm thử.

*Imaginative use* (sử dụng sáng tạo) *debugger* còn giúp học ngôn ngữ lập trình, quan sát các kỹ thuật tối ưu hóa và cách truyền đối số giữa các hàm.

## System Perturbers (Công Cụ Nhiễu Loạn Hệ Thống)

Đây là nhóm công cụ chuyên gây nhiễu để phát hiện lỗi khó tái hiện, ví dụ:

- **Memory filling:** Nạp bộ nhớ với giá trị bất kỳ để kiểm tra biến chưa khởi tạo.
- **Memory shaking:** Tái bố trí bộ nhớ trong môi trường đa nhiệm.
- **Selective memory failing:** Giả lập thiếu bộ nhớ hoặc thất bại khi cấp phát động.
- **Memory-access checking (kiểm tra truy cập bộ nhớ):** Giám sát thao tác con trỏ và phát hiện các con trỏ bị lỗi (uninitialized, dangling).

## Error Databases (Cơ Sở Dữ Liệu Lỗi)

CSDL lỗi giúp quản lý và kỹ thuật: kiểm tra lỗi lặp lại, theo dõi tốc độ phát hiện/sửa lỗi, thống kê trạng thái và mức độ nghiêm trọng của lỗi. Thông tin cần lưu giữ chi tiết tại phần 22.7, “Keeping Test Records”.

---



## 22.6 Nâng Cao Quy Trình Kiểm Thử

Các bước cải tiến kiểm thử tương tự như cải tiến bất kỳ quy trình nào khác: cần hiểu rõ thực trạng, áp dụng thay đổi nhỏ có kiểm soát, quan sát tác động và điều chỉnh dần cho đến khi quy trình tốt hơn.

### Lập Kế Hoạch Kiểm Thử

**Tham khảo:** Đưa kiểm thử vào tài liệu kế hoạch ngay từ đầu dự án.

Lập kế hoạch kiểm thử từ giai đoạn khởi đầu giúp phân bổ đủ thời gian, coi kiểm thử quan trọng ngang bằng thiết kế và lập trình. Kế hoạch kiểm thử chi tiết còn bảo đảm quá trình kiểm thử có tính lặp lại (repeatable).

### Kiểm Thử Lặp Lại (Regression Testing)

Kiểm thử hồi quy đảm bảo sau mỗi thay đổi sản phẩm vẫn thỏa mãn các kiểm thử trước đó. Phải đảm bảo chạy cùng một bộ kiểm thử qua mỗi lần thay đổi.

### Kiểm Thử Tự Động (Automated Testing)

Cách duy nhất quản lý hiệu quả kiểm thử hồi quy là tự động hóa:

- Kiểm thử thủ công dễ dẫn tới lỗi mất và bỏ sót do lặp lại nhiều lần.
- Công cụ kiểm thử tự động giảm rủi ro mắc lỗi.
- Bộ kiểm thử tự động được sử dụng liên tục với nỗ lực bổ sung thấp nhất.
- Hỗ trợ phát hiện lỗi sớm, giảm thời gian và công sức điều tra.
- Tăng cường độ an toàn cho các thay đổi lớn nhờ khả năng quét lỗi nhanh sau chỉnh sửa.
- Đặc biệt hữu ích trong các môi trường công nghệ mới, chưa ổn định.

Các công cụ chính gồm: scaffolding, generator đầu vào, bắt đầu ra, so sánh đầu ra thực tế với mong đợi,...

---

## 22.7 Ghi Nhận Hồ Sơ Kiểm Thử

Bên cạnh khả năng lặp lại, cần đo lường dự án để xác định tác động của thay đổi. Một số dữ liệu lấy mẫu:

- Thông tin hành chính về lỗi (ngày báo cáo, người báo cáo, tiêu đề, phiên bản build, ngày sửa,...)
- Mô tả đầy đủ vấn đề
- Các bước để tái hiện lỗi
- Cách khắc phục tạm thời
- Các lỗi liên quan
- Mức độ nghiêm trọng (fatal, cosmetic,...)
- Nguồn gốc lỗi (yêu cầu, thiết kế, lập trình, kiểm thử,...)
- Phân loại lỗi (off-by-one, sai gán giá trị, chỉ số mảng sai, gọi routine sai,...)
- Lớp/routine bị ảnh hưởng
- Số lượng dòng mã bị ảnh hưởng
- Số giờ để tìm/sửa lỗi

Sử dụng các dữ liệu trên, bạn có thể tính toán:

- Số lỗi theo từng class/routine, chuẩn hóa theo kích thước
- Trung bình số giờ kiểm thử cho mỗi lỗi phát hiện
- Trung bình số lỗi phát hiện trên mỗi trường hợp kiểm thử
- Tỷ lệ mã nguồn được bao phủ bởi kiểm thử
- Số lỗi tồn đọng theo mức độ nghiêm trọng,...

### Ghi Chép Cá Nhân Về Kiểm Thử

Có thể hữu ích khi giữ bảng kiểm tra lỗi cá nhân và thống kê thời gian dành cho viết code, kiểm thử, sửa lỗi.

---

## Tài Liệu Tham Khảo

Có nhiều sách chuyên sâu về kiểm thử phần mềm, chẳng hạn:

- **Testing Computer Software, 2nd ed.** – Kaner, Cem, Jack Falk và Hung Q. Nguyen. Phù hợp cho kiểm thử các ứng dụng phân phối điện rộng.
- **Lessons Learned in Software Testing** – Kaner, Cem, James Bach, và Bret Pettichord. Bổ trợ cho tựa sách ở trên, đi sâu vào thực tiễn kiểm thử.

---

*Lưu ý: Các đoạn mã hoặc chỉ dẫn lệnh trong tài liệu gốc được giữ nguyên dạng, bao quanh bởi dấu `markdown` nếu cần tích hợp vào nội dung tài liệu dịch.*

# Giới thiệu về Kiểm thử Phần mềm

## Tài liệu tham khảo

**Tamre, Louise. *Introducing Software Testing*. Boston, MA: Addison-Wesley, 2002.**

Đây là một cuốn sách về kiểm thử phần mềm dễ tiếp cận, hướng tới đối tượng là các lập trình viên cần hiểu về kiểm thử. Trái với tiêu đề, cuốn sách đi sâu vào các chi tiết liên quan đến kiểm thử mà thậm chí hữu ích với cả những kiểm thử viên có kinh nghiệm.

---

**Whittaker, James A. *How to Break Software: A Practical Guide to Testing*. Boston, MA: Addison-Wesley, 2002.**

Cuốn sách này liệt kê 23 phương pháp tấn công (attacks) mà kiểm thử viên có thể sử dụng để làm phần mềm gặp lỗi, đồng thời trình bày các ví dụ với những phần mềm phổ biến. Bạn có thể sử dụng cuốn sách này làm nguồn thông tin chính về kiểm thử hoặc bổ sung cho các tài liệu khác do phương pháp tiếp cận đặc biệt của nó.

---

**Whittaker, James A. "What Is Software Testing? And Why Is It So Hard?" *IEEE Software*, tháng 1 năm 2000, tr. 70–79.**

Bài báo này cung cấp một phần giới thiệu tốt về các vấn đề trong kiểm thử phần mềm và giải thích những thách thức liên quan đến kiểm thử hiệu quả.

---

**Myers, Glenford J. *The Art of Software Testing*. New York, NY: John Wiley, 1979.**

Đây là cuốn sách kinh điển về kiểm thử phần mềm và vẫn còn được xuất bản (dù giá tương đối cao). Nội dung của sách rất rõ ràng: Bài kiểm tra tự đánh giá; Tâm lý học và kinh tế học của kiểm thử chương trình; Kiểm tra, duyệt và đánh giá chương trình; Thiết kế bộ kiểm thử (test-case design); Kiểm thử theo lớp; Kiểm thử bậc cao; Debugging; Công cụ và kỹ thuật khác. Sách ngắn (177 trang), dễ đọc. Bài quiz ở phần đầu giúp bạn bắt đầu suy nghĩ như một kiểm thử viên và cho thấy có nhiều cách làm hỏng một đoạn code.

---

## Test Scaffolding (Khung kiểm thử - hỗ trợ kiểm thử tự động và kiểm thử đơn vị)

**Bentley, Jon. "A Small Matter of Programming" trong *Programming Pearls*, 2nd Ed. Boston, MA: Addison-Wesley, 2000.**

Bài luận này bao gồm nhiều ví dụ tốt về test scaffolding.

**Mackinnon, Tim, Steve Freeman, and Philip Craig. "Endo-Testing: Unit Testing with Mock Objects," *eXtreme Programming and Flexible Processes Software Engineering - XP2000 Conference*, 2000.**

Bài viết này là tài liệu gốc thảo luận việc sử dụng mock objects (đối tượng mô phỏng) để hỗ trợ kiểm thử của lập trình viên.

**Thomas, Dave và Andy Hunt. "Mock Objects," *IEEE Software*, tháng 5/6 năm 2002.**

Bài viết này dễ tiếp cận, giới thiệu về việc sử dụng mock objects để hỗ trợ kiểm thử.

Trang web hỗ trợ lập trình viên sử dụng JUnit:  
[www.junit.org](http://www.junit.org)

Các nguồn tương tự:

- [cppunit.sourceforge.net](http://cppunit.sourceforge.net)
- [nunit.sourceforge.net](http://nunit.sourceforge.net)

---

## Test First Development (Phát triển hướng kiểm thử - Test-Driven Development)

Beck, Kent. *Test-Driven Development: By Example*. Boston, MA: Addison-Wesley, 2003.

Beck mô tả chi tiết về “test-driven development” (phát triển hướng kiểm thử) – một phương pháp phát triển nổi bật bằng việc viết test case trước, sau đó mới viết code đáp ứng test case đó. Mặc dù ngôn ngữ có phần “truyền giáo”, lời khuyên trong sách là xác đáng. Sách ngắn, đi thẳng vào vấn đề, có ví dụ thực tế với code thật.

---

## Relevant Standards (Các tiêu chuẩn liên quan)

- IEEE Std 1008-1987 (R1993), *Standard for Software Unit Testing*
- IEEE Std 829-1998, *Standard for Software Test Documentation* (Tiêu chuẩn tài liệu kiểm thử phần mềm)
- IEEE Std 730-2002, *Standard for Software Quality Assurance Plans* (Kế hoạch đảm bảo chất lượng phần mềm)

---

## CHECKLIST: Test Cases (Danh sách kiểm tra cho bộ kiểm thử)

- ☐ Mỗi yêu cầu áp dụng cho class hoặc routine (hàm, phương thức) đều có test case riêng?
- ☐ Mỗi phần tử từ thiết kế liên quan đến class hoặc routine đều có test case riêng?
- ☐ Mỗi dòng code đã được kiểm thử ít nhất với một test case? Việc này đã được xác minh bằng cách tính toán số lượng tối thiểu các bài kiểm thử cần thiết để thực thi từng dòng code chưa?
- ☐ Mọi đường đi luồng dữ liệu (data-flow path) defined-used đã được kiểm thử với ít nhất một test case?
- ☐ Đã kiểm tra code với các dạng luồng dữ liệu có khả năng sai sót, như defined-defined, defined-exited, defined-killed?
- ☐ Đã sử dụng danh sách lỗi phổ biến để viết test case nhằm phát hiện các lỗi hay xảy ra trong quá khứ không?
- ☐ Đã kiểm tra mọi giới hạn đơn giản (boundary): giá trị lớn nhất, nhỏ nhất, và các giá trị sát ranh giới (off-by-one)?
- ☐ Đã kiểm tra mọi giới hạn kết hợp (compound boundaries) – tức là các tổ hợp dữ liệu đầu vào có thể khiến biến tính toán được quá lớn hoặc quá nhỏ?
- ☐ Test cases kiểm tra dữ liệu nhập liệu sai kiểu (wrong kind of data), ví dụ, số nhân viên âm trong chương trình tính lương?
- ☐ Có kiểm thử với các giá trị đại diện/điển hình giữa phạm vi?
- ☐ Đã kiểm thử cấu hình tối thiểu hợp lệ (minimum normal configuration)?
- ☐ Đã kiểm thử cấu hình tối đa hợp lệ (maximum normal configuration)?
- ☐ Đã kiểm thử khả năng tương thích với dữ liệu cũ? Đã kiểm thử phần cứng cũ, phiên bản hệ điều hành cũ, và giao tiếp với các phần mềm phiên bản cũ?
- ☐ Các test case có giúp kiểm tra bằng tay dễ dàng không?

---

## Key Points (Những điểm chính)

- Kiểm thử bởi lập trình viên là thành phần chủ chốt của một chiến lược kiểm thử hoàn chỉnh. Việc kiểm thử độc lập cũng rất quan trọng nhưng không nằm trong phạm vi cuốn sách này.
- Việc viết test case trước hoặc sau khi viết code tiêu tốn cùng một lượng thời gian, nhưng kiểm thử trước giúp rút ngắn chu trình phát hiện và sửa lỗi.
- Mặc dù có rất nhiều loại kiểm thử, kiểm thử chỉ là một phần của chương trình đảm bảo chất lượng phần mềm. Phương pháp phát triển chất lượng, như giảm thiểu lỗi ngay từ yêu cầu và thiết kế, cũng quan trọng không kém. Các thực hành phát triển hợp tác cũng hiệu quả tương tự kiểm thử trong việc phát hiện lỗi, mà các thực hành này còn phát hiện ra các dạng lỗi khác.
- Bạn có thể tạo ra nhiều test case một cách xác định bằng các phương pháp như basis testing, data-flow analysis, boundary analysis, phân lớp dữ liệu sai (classes of bad data) và phân lớp dữ liệu đúng (classes of good data). Ngoài ra, kỹ thuật error guessing cũng hữu ích.
- Lỗi thường tập trung ở một số class hoặc routine dễ bị lỗi. Hãy xác định, thiết kế lại và viết lại code của những phần này.

- Dữ liệu kiểm thử thường chứa nhiều lỗi hơn chính code đang kiểm thử. Săn lùng những lỗi này gây lãng phí thời gian mà không cải thiện code, vì vậy các lỗi trong dữ liệu kiểm thử còn gây khó chịu hơn lỗi lập trình. Hãy đầu tư vào phát triển test cẩn thận như phát triển code.
- Kiểm thử tự động (automated testing) nói chung hữu ích và là yếu tố then chốt cho regression testing (kiểm thử hồi quy).
- Cách tốt nhất để cải tiến việc kiểm thử lâu dài là thực hiện thường xuyên, đo lường, và sử dụng những gì học được để cải thiện quy trình kiểm thử.

## Chương 23: Debugging (Gỡ lỗi)

### Mục lục

- 23.1 Tổng quan về các vấn đề Debugging: trang 535
- 23.2 Tìm lỗi: trang 540
- 23.3 Sửa lỗi: trang 550
- 23.4 Xem xét tâm lý trong Debugging: trang 554
- 23.5 Công cụ Debugging - Rõ ràng và ít rõ ràng: trang 556

### Chủ đề liên quan

- Toàn cảnh về chất lượng phần mềm: Chương 20
- Kiểm thử bởi lập trình viên: Chương 22
- Refactoring (Tái cấu trúc): Chương 24

## Tổng quan về các vấn đề Debugging

**Debugging khó gấp đôi việc lập trình. Do đó, nếu bạn viết code một cách quá "khôn khéo", chính bạn sẽ không đủ thông minh để gỡ lỗi nó.** —Brian W. Kernighan

*Debugging* là quá trình xác định nguyên nhân gốc (root cause) của một "error" (lỗi) và sửa chữa nó. Nó khác biệt với *testing*, là quá trình phát hiện lỗi ban đầu. Ở một số dự án, thời gian dành cho debugging chiếm tới 50% tổng thời gian phát triển.

Đối với nhiều lập trình viên, debugging là phần khó khăn nhất trong lập trình. Tuy nhiên, bạn có thể giảm số lượng lỗi cần gỡ với các lời khuyên phù hợp. Đa phần lỗi bạn gặp chỉ là lỗi nhỏ hoặc lỗi đánh máy, dễ tìm thấy khi xem lại mã nguồn hoặc chạy qua *debugger*. Chương này đề cập đến cách làm debugging trở nên dễ dàng hơn.

### Nguồn gốc từ "bug" trong phần mềm

Thuật ngữ "bug" trong phần mềm thường được cho là xuất phát từ sự kiện một con bướm đêm (moth) bị kẹt vào mạch điện của máy tính Mark I, gây ra sự cố. Ngoài lĩnh vực phần mềm, từ "bug" xuất hiện ít nhất từ thời Thomas Edison (1878).

*Bug* thực ra không phải là các sinh vật "chui" vào code mà là do lập trình viên mắc lỗi. Trong ngữ cảnh kỹ thuật, "bug" nên gọi là "error", "defect" (khuyết tật), hoặc "fault" (lỗi).

## Vai trò của Debugging đối với chất lượng phần mềm

Tương tự như kiểm thử, debugging không giúp nâng cao chất lượng phần mềm; nó chỉ là phương tiện để chẩn đoán lỗi. Chất lượng phần mềm cần được xây dựng ngay từ đầu thông qua yêu cầu, thiết kế và thực hành lập trình chất lượng cao. Debugging là giải pháp cuối cùng.

## Sự khác biệt về hiệu quả Debugging

Nghiên cứu cho thấy, giữa các lập trình viên có kinh nghiệm, vẫn tồn tại sự khác biệt đến 20 lần về thời gian tìm ra cùng một tập lỗi. Người giỏi nhất có thể tìm hết lỗi và sửa chính xác mà không làm phát sinh lỗi mới. Người kém nhất có thể không phát hiện được phần lớn lỗi và thậm chí tạo ra nhiều lỗi mới khi sửa.

Kết quả một nghiên cứu kinh điển:

**3 người nhanh nhất 3 người chậm nhất**

	3 người nhanh nhất	3 người chậm nhất
Thời gian debug trung bình (phút)	5.0	14.1
Số lỗi không phát hiện	0.7	1.7
Số lỗi mới tạo ra khi sửa	3.0	7.7

Nguồn: “Some Psychological Evidence on How People Debug Computer Programs” (Gould 1975)

Nhóm chậm cần tới gần 14 vòng debug, mỗi vòng mất thời gian gấp 3 lần nhóm nhanh, để hoàn thành việc sửa lỗi. Điều này đã được các nghiên cứu khác xác nhận.

**Nguyên tắc chung về chất lượng phần mềm:** Nâng cao chất lượng sẽ làm giảm chi phí phát triển. Bạn không cần phải lựa chọn giữa chất lượng, chi phí, và thời gian – ba yếu tố này nâng đỡ lẫn nhau.

## Xem lỗi như cơ hội cải thiện

Khi có một defect, có nghĩa là bạn chưa hoàn toàn hiểu chương trình của mình, và đó là dịp để học được nhiều điều:

- **Tìm hiểu sâu hơn về chương trình:** Nếu bạn hiểu hoàn toàn, đã không có lỗi.
- **Nhận diện loại lỗi mình thường mắc:** Tại sao bạn mắc lỗi này? Làm sao phát hiện sớm hơn? Có kiểu lỗi tương tự nào ẩn trong code không?
- **Đánh giá khả năng đọc code:** Đôi khi việc đọc lại code để tìm lỗi giúp nhận ra những điểm chưa rõ ràng, từ đó nâng cao chất lượng code trong tương lai.
- **Nhìn lại cách giải quyết vấn đề:** Bạn có tự tin vào quy trình debugging của mình? Bạn hay đoán mò hay có phương pháp rõ ràng? Tối ưu hóa phương pháp debug giúp giảm tổng thời gian phát triển.
- **Đánh giá cách sửa lỗi:** Bạn chỉ sửa "triệu chứng" hay dám sửa tận gốc vấn đề? Sửa tạm bằng "goto" hay giải quyết bài bản?

Tóm lại, debugging là "mảnh đất màu mỡ" để bạn phát triển kỹ năng lập trình.

## Một số cách tiếp cận debug thiếu hiệu quả

Đáng tiếc, các chương trình đào tạo hiếm khi cung cấp hướng dẫn debug hiệu quả. Nhiều lập trình viên tự "phát minh" lại các nguyên tắc debug cơ bản, dẫn đến lãng phí thời gian.

### The Devil’s Guide to Debugging

Một số thói quen debug "đáng tránh" bao gồm:

- **Tìm lỗi bằng cách đoán mò:** Rải các print statement, thay đổi đại code, không lưu bản gốc, không ghi chú thay đổi.
- **Không hiểu bản chất vấn đề:** Không cần biết rõ, chỉ cần sửa cho chạy.
- **Sửa lỗi bằng biện pháp rõ ràng nhất ở chỗ quan sát thấy lỗi**

```
x = Compute( y )
if ( y = 17 )
    x = $25.15
```

- **Debugging theo kiểu mê tín:** Đổ lỗi cho máy, compiler, hệ điều hành thay vì nhận trách nhiệm về lỗi.

**Ghi nhớ:** Lúc đầu, hãy luôn giả định lỗi là do mình. Điều này vừa giúp tìm lỗi hiệu quả hơn, vừa củng cố uy tín, tránh phải giải thích lại khi sau này phát hiện lỗi thực sự là của mình.

## 23.2 Tìm kiếm Defect (Finding a Defect)

Quá trình debug bao gồm việc tìm ra và hiểu rõ defect, thường chiếm đến 90% công việc. Sử dụng phương pháp khoa học (scientific method) giúp quá trình này hiệu quả hơn nhiều so với việc đoán mò.

### Phương pháp Khoa học trong Debugging

Các bước thực hiện:

1. Thu thập dữ liệu thông qua các thử nghiệm lặp lại được (repeatable experiments).

2. Xây dựng giả thuyết phù hợp với dữ liệu thu được.
3. Thiết kế thử nghiệm để kiểm chứng giả thuyết.
4. Kiểm chứng hoặc bác bỏ giả thuyết.
5. Lập lại quy trình khi cần thiết.

Quy trình này có nhiều điểm tương đồng với kỹ thuật *debugging* hiệu quả.

Hết phần dịch.

## Ổn Định Lỗi

### 2. Xác Định Nguồn Gốc Lỗi (the “fault”)

#### a. Thu thập dữ liệu gây ra lỗi

#### b. Phân tích dữ liệu đã thu thập, hình thành giả thuyết về lỗi

#### c. Xác định cách kiểm chứng hoặc bác bỏ giả thuyết, bằng kiểm thử chương trình hoặc xem xét mã nguồn

#### d. Kiểm chứng hoặc bác bỏ giả thuyết bằng phương pháp xác định ở 2(c)

3. Khắc phục lỗi
4. Kiểm thử bản vá
5. Tìm các lỗi tương tự

Bước đầu tiên này tương tự với bước đầu tiên trong phương pháp khoa học, dựa vào **tính lặp lại**. Lỗi sẽ dễ chẩn đoán hơn nếu có thể “ổn định hóa” nó—tức là khiến nó xuất hiện một cách đáng tin cậy.

Bước thứ hai sử dụng quy trình của phương pháp khoa học. Bạn thu thập dữ liệu kiểm thử cho thấy lỗi, phân tích dữ liệu thu được và hình thành giả thuyết về nguồn gốc của lỗi. Sau đó, bạn thiết kế ca kiểm thử hoặc hình thức kiểm tra để đánh giá giả thuyết, và hoặc xác nhận giả thuyết đã đúng, hoặc tiếp tục điều chỉnh. Khi đã kiểm chứng được giả thuyết, bạn sửa lỗi, kiểm thử lại, và tìm kiếm các lỗi tương tự trong mã nguồn.

## Ví dụ Minh Họa

Giả sử bạn có một chương trình cơ sở dữ liệu nhân viên, thỉnh thoảng xảy ra lỗi. Chương trình này in danh sách nhân viên cùng số tiền khấu trừ thuế theo thứ tự chữ cái. Dưới đây là một phần của đầu ra:

```
Formatting, Fred Freeform $5,877
Global, Gary $1,666
Modula, Mildred $10,788
Many-Loop, Mavis $8,889
Statement, Sue Switch $4,000
Whileloop, Wendy $7,860
```

Lỗi ở đây là Many-Loop, Mavis và Modula, Mildred bị sắp xếp sai thứ tự.

## Ổn Định Lỗi

Nếu một **defect** (khuyết) không xuất hiện một cách đáng tin cậy, việc chuẩn đoán gần như là bất khả thi. Làm cho một lỗi gián đoạn trở nên có thể lặp lại là một trong những thách thức khó nhất khi **debugging** (gỡ lỗi). Một lỗi không xuất hiện một cách nhất quán, thường là **initialization error** (lỗi khởi tạo), **timing issue** (lỗi thời gian) hoặc **dangling-pointer problem** (trò trò lơ lửng — xem thêm phần “Pointers”). Chẳng hạn, nếu phép tính tổng đúng lúc này, sai lúc khác, có thể một biến không được khởi tạo đúng — hầu hết thời gian nó chỉ tình cờ là 0.

Nếu chương trình sử dụng **pointer** mà xuất hiện hiện tượng kỳ lạ, rất có thể con trỏ chưa được khởi tạo hoặc đã tham chiếu vùng nhớ đã giải phóng.

Ổn định lỗi không chỉ tìm một ca kiểm thử có thể sinh lỗi, mà còn làm đơn giản ca kiểm thử đến mức nếu đổi bất kỳ yếu tố nào thì lỗi không còn xuất hiện. Mục tiêu là đơn giản hóa ca kiểm thử tới mức độ mà mỗi thay đổi đều ảnh hưởng tới hiện tượng lỗi.

# Đơn Giản Hóa Ca Kiểm Thử Bằng Phép Loại Trừ

Giả sử có 10 yếu tố phối hợp tạo nên lỗi. Đặt giả thuyết yếu tố nào không liên quan, đối những yếu tố này và chạy lại thử nghiệm. Nếu lỗi vẫn xuất hiện, loại bỏ các yếu tố đó khỏi nhóm nghi ngờ và tiếp tục đơn giản hóa thêm.

Ví dụ, khi chương trình chạy lần đầu, Many-Loop, Mavis bị xếp sau Modula, Mildred. Nhưng khi chạy lại, danh sách đã đúng thứ tự. Đến khi Fruit-Loop, Frita được nhập vào (cũng gặp lỗi tương tự), bạn nhớ ra: cả Modula, Mildred trước đó cũng vừa được nhập lẻ.

Bạn đưa ra giả thuyết: “Lỗi liên quan tới việc nhập từng nhân viên lẻ.” Nếu đúng, chạy lại chương trình sẽ đặt Fruit-Loop, Frita về đúng chỗ:

```
Formatting, Fred Freeform $5,877
Fruit-Loop, Frita $5,771
Global, Gary $1,666
Many-Loop, Mavis $8,889
Modula, Mildred $10,788
Statement, Sue Switch $4,000
Whileloop, Wendy $7,860
```

Ca chạy này củng cố giả thuyết. Để kiểm chứng, bạn tiếp tục nhập thêm từng nhân viên và theo dõi kết quả.

---

## Xác Định Nguồn Gốc Lỗi Bằng Phương Pháp Khoa Học

Có thể bạn nghi ngờ một **off-by-one error** (lỗi lệch chỉ số 1). Hãy thử thay đổi tham số gần biên, ở ngay biên và quá biên để kiểm tra. Trong ví dụ, bạn kiểm tra lại với một nhân viên mới. Nhưng kết quả lại đúng, bác bỏ giả thuyết đầu.

Xem lại đầu ra thử nghiệm, bạn nhận thấy chỉ có Fruit-Loop, Frita và Many-Loop, Mavis là tên có dấu gạch nối (hyphen). Cập nhập giả thuyết: “Lỗi do tên có dấu gạch nối.”

Tiếp tục theo dõi, bạn phát hiện ra mã nguồn sử dụng hai thuật toán sắp xếp khác nhau: một khi nhập dữ liệu mới (chỉ sắp xếp sơ bộ) và một khi lưu dữ liệu (sắp xếp hoàn chỉnh). Thuật toán sắp xếp sơ bộ không xử lý tốt các ký tự dấu câu như dấu gạch nối, dẫn đến dữ liệu in ra chưa sắp xếp đúng.

Bạn điều chỉnh giả thuyết cuối: “Tên có dấu câu không được sắp xếp chính xác cho tới khi lưu lại dữ liệu.” Thử nghiệm thêm ca kiểm thử để xác nhận điều này.


---

## Một số Kỹ Thuật Tìm Lỗi

- **Sử dụng tối đa dữ liệu hiện có:** Khi tạo giả thuyết, giải thích càng nhiều dữ liệu càng tốt. Nếu dữ liệu nào không phù hợp với giả thuyết, đừng loại bỏ — hãy đặt câu hỏi và tạo giả thuyết mới.
- **Làm tinh gọn ca kiểm thử:** Nếu không tìm được nguồn lỗi, tiếp tục giản lược ca kiểm thử, thay đổi nhiều tham số hơn nữa.
- **Kiểm thử đơn vị (unit test):** Lỗi dễ tìm ở đoạn mã nhỏ, chạy kiểm thử đơn vị để cô lập mã.
- **Sử dụng công cụ hỗ trợ:** Tận dụng các công cụ như interactive debuggers (gỡ lỗi tương tác), memory checkers (kiểm soát bộ nhớ), picky compilers (trình biên dịch cảnh báo nhiều),...

Một ví dụ: Lỗi ghi đè vùng nhớ khó nhận biết trực tiếp, nhưng có thể dễ dàng tìm ra khi sử dụng memory breakpoint (điểm dừng bộ nhớ).

- **Tái hiện lỗi bằng nhiều cách khác nhau:** Thử các ca kiểm thử tương tự để thu hẹp vùng nghi ngờ, như hình minh họa bên dưới:

 Minh Họa: Tái hiện lỗi các cách khác nhau để xác định nguyên nhân.

- **Sinh thêm dữ liệu để tạo thêm giả thuyết mới:** Thiết kế thêm ca kiểm thử khác biệt với các ca đã biết.
- **Sử dụng kết quả kiểm thử phủ định:** Nếu ca kiểm thử bác bỏ giả thuyết, hãy điều chỉnh phạm vi tìm kiếm lỗi.
- **Động não liệt kê nhiều giả thuyết:** Không dừng lại ở giả thuyết đầu tiên — đưa ra nhiều phương án sau đó kiểm thử từng cái.

---

## Một số Lời khuyên Thực Tế

- **Ghi chú các phương án thử nghiệm:** Đừng sa đà vào một hướng duy nhất, nếu không hiệu quả thì thử chuyển sang hướng khác.

- **Thu hẹp vùng nghi ngờ trong mã:** Thay vì kiểm tra toàn bộ, hãy sử dụng log, trace, hoặc binary search algorithm (thuật toán tìm kiếm nhị phân) để tập trung vào vùng có khả năng chứa lỗi.
- **Đặc biệt kiểm tra các class/routine hay dính lỗi:** Module nhiều tiền sử lỗi thường rất dễ phát sinh lỗi mới, hãy kiểm tra kỹ vùng này.
- **So sánh phiên bản cũ – mới:** Nếu lỗi mới xuất hiện, hãy đối chiếu source code cũ - mới bằng diff tool hoặc kiểm tra version control log.
- **Tăng dần phạm vi nghi ngờ nếu không phát hiện lỗi:** Đừng quá tập trung vào một vùng hẹp nếu không hiệu quả, hãy thử mở rộng ra.
- **Tích hợp hệ thống từng phần để debug hiệu quả:** Tích hợp từng module nhỏ, kiểm thử từng bước, nếu có lỗi thì dễ dàng cách ly.
- **Dùng checklist kiểm tra lỗi phổ biến:** Tham khảo checklist kiểm tra chất lượng code để tránh lỗi thường gặp.
- **Đem vấn đề trao đổi với đồng nghiệp:** Chỉ cần trình bày lại vấn đề với người khác đôi khi cũng phát hiện ra nguyên nhân. (confessional debugging)
- **Thư giãn và nghỉ ngơi:** Khi căng thẳng hoặc bí ý tưởng, hãy nghỉ giải lao ngắn — nhiều ý tưởng hoặc giải pháp sẽ xuất hiện ngoài giờ làm!

## Brute-Force Debugging (Gỡ Lỗi Bằng Sức Mạnh Thô)

Đây là phương pháp thường bị bỏ qua, dù rất hiệu quả với các ca lỗi phức tạp. Tuy có thể tốn công, tỉ mỉ và mất thời gian, nhưng lại **đảm bảo** giải quyết được vấn đề.

Một số kỹ thuật brute-force tiêu biểu:

- Thực hiện **full design/code review** trên phân đoạn bị lỗi
- Viết lại từ đầu một phần mã, thậm chí toàn bộ chương trình
- Biên dịch với đầy đủ thông tin debug
- Biên dịch ở mức cảnh báo khắt khe nhất và sửa tất cả các cảnh báo
- Đặt mã vào **unit test harness** (giàn kiểm thử đơn vị) và kiểm tra độc lập
- Tạo bộ kiểm thử tự động, chạy qua đêm
- Dùng debugger bước qua từng vòng lặp cho đến khi lỗi xuất hiện
- Bổ sung các lệnh print/log/tracing để theo dõi chạy mã
- Biên dịch/chạy bằng trình biên dịch khác, môi trường khác
- Liên kết với các thư viện đặc biệt cảnh báo sử dụng sai
- Mô phỏng đúng môi trường máy người dùng cuối
- Tích hợp mã mới từng phần nhỏ, kiểm thử đầy đủ từng phần trước khi tích hợp tiếp theo

**Lưu ý:** Đặt giới hạn thời gian cho mỗi kỹ thuật brute-force để tránh "sa lầy" quá lâu mà không tiến triển.

**Tóm lại:** Quy trình gỡ lỗi hiệu quả cần ổn định lỗi, thu hẹp và cô lập nguyên nhân bằng phương pháp khoa học, sử dụng công cụ thích hợp, đa dạng hoá ca kiểm thử, tận dụng kinh nghiệm đồng đội cũng như sẵn sàng dùng brute-force khi cần thiết. Không ngừng đặt câu hỏi, điều chỉnh giả thuyết và nghỉ ngơi hợp lý sẽ giúp bạn gỡ rối hiệu quả ngay cả các lỗi khó nhằn nhất!

## Bản Dịch Học Thuật: Kỹ Thuật Gỡ Lỗi và Khía Cạnh Tâm Lý

### 1. Sự Đánh Đổi Giữa Rủi Ro và An Toàn Khi Gỡ Lỗi

Trong mỗi lập trình viên đều có phần "cá cược": bạn thường ưu tiên tiếp cận nhanh, nhiều rủi ro với kỳ vọng sẽ phát hiện lỗi chỉ sau năm phút, thay vì chọn phương pháp chắc chắn nhưng mất đến nửa tiếng mới xác định được lỗi đó. Rủi ro nằm ở chỗ, nếu cách năm phút không hiệu quả, bạn dễ cố chấp — việc tìm lỗi "đề" trở thành vấn đề nguyên tắc, dẫn đến hàng giờ, thậm chí nhiều ngày, tuần, tháng trôi qua không hiệu quả.

Có bao nhiêu lần bạn từng dành hai tiếng để debug đoạn code chỉ mất 30 phút để viết? Sự phân bổ lao động này là không tối ưu; trong trường hợp như vậy, viết lại code còn hiệu quả hơn là tiếp tục sửa code tồi.

Khi quyết định áp dụng giải pháp nhanh, hãy thiết lập giới hạn thời gian tối đa cho bản thân. Nếu quá thời hạn, hãy chấp nhận rằng lỗi này phức tạp hơn tưởng tượng ban đầu và chuyển sang phương pháp xử lý triệt để hơn. Cách tiếp cận này giúp bạn xử lý các lỗi đơn giản một cách nhanh chóng và chỉ dành thêm thời gian cho các lỗi khó.

### Lập Danh Sách Kỹ Thuật Brute-force



Trước khi bắt đầu gỡ một lỗi phức tạp, hãy tự hỏi: "Nếu tôi bị mắc kẹt khi debug vấn đề này, có kỹ thuật brute-force (thử tất cả khả năng) nào đảm bảo sẽ xử lý được không?" Nếu xác định được ít nhất một kỹ thuật brute-force (bao gồm cả việc viết lại code), bạn sẽ ít lãng phí thời gian dù có giải pháp nhanh hơn.

---

## 2. Lỗi Cú Pháp (Syntax Errors)

Các lỗi cú pháp đang ngày càng hiếm nhờ các *compiler* (trình biên dịch) ngày càng "thông minh". Những ngày phải mất hai tiếng chỉ để tìm dấu chấm phẩy trong Pascal đã sắp qua.

Một số hướng dẫn giúp giảm thiểu loại lỗi này:

- **Đừng tin hoàn toàn vào số dòng báo lỗi của compiler:** Khi nhận được thông báo lỗi lạ về cú pháp, hãy kiểm tra ngay trước và sau vị trí báo lỗi, vì *compiler* có thể chẩn đoán sai vấn đề.
- **Đừng tin tuyệt đối vào thông báo của compiler:** Thường bạn cần "đọc giữa các dòng", ví dụ, thông báo "floating exception" trên UNIX C cho lỗi chia cho 0 với số nguyên. Trong C++ với *STL* (*Standard Template Library*), bạn có thể thấy thông báo "Error message too long for printer to print; message truncated."
- **Đừng quan tâm quá nhiều đến lỗi thứ hai trở đi:** Một số *compiler* có thể tạo ra chuỗi thông báo lỗi không liên quan sau khi phát hiện lỗi đầu tiên. Hãy sửa lỗi đầu tiên và biên dịch lại.
- **Chia nhỏ và kiểm thử (Divide and conquer):** Nếu lỗi cú pháp khó phát hiện, hãy xóa bớt từng phần code và biên dịch lại để xác định vị trí cụ thể của lỗi.
- **Chú ý các comment hoặc dấu nháy bị lẫn:** Trong môi trường IDE (Integrated Development Environment) nguyên thủy, sử dụng:

```
/* */
```

để đóng hoặc xác định đâu là comment hoặc string bị chưa đóng.

---

## 3. Sửa Lỗi (Fixing a Defect)

Khó khăn lớn nhất khi gỡ lỗi là tìm được nguyên nhân; việc sửa lỗi thực chất lại chính là phần dễ gây ra sai sót.

Nghiên cứu cho thấy xác suất sửa sai còn cao hơn 50%. Để giảm thiểu:

- **Hiểu thấu đáo vấn đề trước khi sửa:** Đừng vội vàng sửa lỗi khi chưa thật sự hiểu. Hãy dùng các trường hợp kiểm thử để xác nhận nguyên nhân.
- **Hiểu cả chương trình chứ không chỉ vấn đề:** Càng hiểu rõ mối quan hệ giữa các thành phần, bạn càng có khả năng sửa đúng và đủ.
- **Xác nhận chẩn đoán lỗi:** Đừng nóng vội. Hãy dùng kiểm thử để loại trừ các khả năng khác trước khi sửa.

*"Đừng debug khi đang đứng."*

—Gerald Weinberg

- **Giữ nguyên source code ban đầu:** Luôn backup trước khi sửa, để dễ dàng so sánh tìm lỗi khi cần rollback.
- **Sửa tận gốc, không chỉ sửa triệu chứng:** Dưới đây là ví dụ về sửa lỗi sai lầm, chỉ khắc phục triệu chứng mà không khắc phục nguyên nhân:

```
// Cách sửa sai lầm
for ( claimNumber = 0; claimNumber < numClaims[ client ]; claimNumber++ ) {
    sum[ client ] = sum[ client ] + claimAmount[ claimNumber ];
}
if ( client == 45 ) {
    sum[ 45 ] = sum[ 45 ] + 3.45;
}
```

Kỹ thuật đặc biệt hóa như trên ngay lập tức biến code thành một "đằm lầy" về bảo trì.

- **Không thay đổi code ngẫu nhiên:** Việc thử +1/-1 hoặc sửa code mà không hiểu vấn đề là "voodoo programming". Hãy chỉ sửa code khi bạn biết chắc tác động của thay đổi.
- **Chỉ sửa một thứ tại một thời điểm.**

- **Kiểm tra kỹ thuật khắc phục:** Từ kiểm thử thủ công, nhờ người khác kiểm tra đến việc dùng automated regression test (kiểm thử hồi quy tự động) như *JUnit* hoặc *CppUnit*.
- **Bổ sung test case để phòng tái xuất hiện lỗi.**
- **Tìm kiếm các lỗi tương tự:** Khi phát hiện một lỗi, nên kiểm tra phần còn lại của hệ thống để tìm các lỗi cùng loại.

## 4. Yếu Tố Tâm Lý Khi Gỡ Lỗi

Gỡ lỗi (debugging) đòi hỏi tư duy chính xác chặt chẽ — giống như các hoạt động phát triển phần mềm khác. Bạn phải hình thành giả thuyết, thu thập dữ liệu, phân tích, loại trừ một cách có phương pháp — điều không dễ thực hiện do thiên hướng chủ quan.

### Hiện Tượng “Psychological Set” (Thiết Lập Tâm Thức)

Hiện tượng này khiến bạn thấy những gì *bạn kỳ vọng*, để bỏ qua các điểm bất thường. Ví dụ, bạn thấy "Num" và nghĩ rằng nó là viết tắt của "Number", không nhận ra lỗi chính tả.

#### Ví dụ kinh điển:

Paris in the the Spring

Nhiều người chỉ thấy một chữ "the".

#### Hậu quả của "psychological set":

- Kỳ vọng các cấu trúc control mới hoạt động giống cấu trúc cũ.
- Nhận diện biến cũ và mới như nhau do tên gần giống.
- Bỏ qua lỗi chính tả hoặc sai cấu trúc do quá quen thuộc.

#### Giải pháp:

- Rèn luyện thói quen code tốt, dùng tên biến rõ ràng, comment đầy đủ, format nhất quán.

### “Psychological Distance” (Khoảng Cách Tâm Lý) Giữa Tên Biến

Nếu tên biến quá giống nhau, bạn dễ nhầm lẫn khi debug. Dưới đây là bảng minh họa:

Tên Biến 1	Tên Biến 2	Khoảng Cách Tâm Lý
stoppt	stcpt	Gần như không khác biệt
shiftrn	shiftrm	Gần như không khác biệt
dcount	bcoun	Nhỏ
claims1	claims2	Nhỏ
product	sum	Lớn

**Khuyến nghị:** Khi đặt tên biến, hãy chọn những tên thật khác biệt để tránh tối đa các vấn đề nhận diện khi debug.

## Tham Khảo Đọc Thêm

Weinberg, G. (1998). *The Psychology of Computer Programming*.

## Tổng hợp

- Gỡ lỗi hiệu quả là sự kết hợp giữa kỹ năng kỹ thuật và nhận thức tâm lý.
- Đừng cố chấp theo giải pháp nhanh hoặc sửa triệu chứng.
- Luôn kiểm soát lịch sử sửa đổi và bổ sung kiểm thử khi phát hiện lỗi mới.
- Nhận diện các "cạm bẫy tâm lý" để tránh ngộ nhận khi đọc và chỉnh sửa code.

#### Chú Giải Thuật Ngữ:

- *API* (Application Programming Interface): Giao diện lập trình ứng dụng
- *IDE* (Integrated Development Environment): Môi trường phát triển tích hợp
- *STL* (Standard Template Library): Thư viện mẫu tiêu chuẩn trong C++
- *JUnit*, *CppUnit*: Bộ công cụ kiểm thử đơn vị dành cho Java, C++

# Công cụ giúp hoàn thiện quá trình kiểm thử và gỡ lỗi

## Công cụ so sánh mã nguồn (Source-Code Comparators)

Một công cụ so sánh mã nguồn, ví dụ như **Diff**, rất hữu ích khi bạn sửa đổi chương trình để khắc phục lỗi. Nếu bạn thực hiện nhiều thay đổi và cần loại bỏ những thay đổi mà bạn không nhớ rõ, công cụ này giúp xác định sự khác biệt và gọi nhắc lại trí nhớ của bạn. Nếu phát hiện một khiếm khuyết trong phiên bản mới mà bạn không nhớ từng xuất hiện ở phiên bản cũ, bạn có thể so sánh các tập tin để xác định những gì đã thay đổi.

## Thông báo cảnh báo từ trình biên dịch (Compiler Warning Messages)

Một trong những công cụ gỡ lỗi hiệu quả và đơn giản nhất chính là **trình biên dịch** của bạn. Hãy đặt mức độ cảnh báo của trình biên dịch lên cao nhất có thể và sửa tất cả lỗi mà nó báo cáo.

**Lưu ý quan trọng:** Việc bỏ qua các cảnh báo của trình biên dịch là câu trả lời sai. Việc tắt hoàn toàn cảnh báo để khỏi nhìn thấy chúng lại càng nguy hiểm hơn. Việc đó chẳng khác nào “nhắm mắt làm ngơ”, điều đó không làm cho lỗi biến mất mà chỉ khiến bạn không còn nhìn thấy chúng.

Hãy giả định rằng những người viết ra trình biên dịch hiểu biết hơn bạn về ngôn ngữ lập trình. Nếu họ cảnh báo, thường có nghĩa đây là dịp bạn có thể học được điều gì đó mới. Hãy nỗ lực để hiểu thực sự cảnh báo ấy có ý nghĩa gì.

## Đổi xử với các cảnh báo như lỗi (Treat warnings as errors)

Một số trình biên dịch cho phép coi cảnh báo như lỗi. Điều này giúp nâng tầm quan trọng của các cảnh báo, thúc đẩy việc xử lý chúng nghiêm túc hơn. Ngoài ra, nhiều trường hợp cảnh báo có thể ảnh hưởng đến quá trình biên dịch hay liên kết chương trình.

## Tiêu chuẩn hóa cấu hình trình biên dịch trong dự án

Thiết lập một tiêu chuẩn yêu cầu tất cả thành viên trong nhóm sử dụng cùng cấu hình trình biên dịch. Nếu không, việc tích hợp mã nguồn từ nhiều người sẽ phát sinh vô số thông báo lỗi, gây ách tắc trong quá trình tích hợp. Điều này dễ được kiểm soát bằng cách dùng make file hoặc build script thống nhất cho dự án.

## Kiểm tra cú pháp và logic mở rộng (Extended Syntax and Logic Checking)

Có thể sử dụng các công cụ bổ sung để kiểm tra mã nguồn cẩn trọng hơn trình biên dịch, ví dụ: công cụ **lint** trong C giúp phát hiện việc sử dụng biến chưa khởi tạo, hoặc nhầm lẫn khi viết `=` thay vì `==`, cùng nhiều vấn đề tinh vi khác.

## Phân tích thực thi (Execution Profilers)

Bạn có thể không nghĩ rằng **profiler** là công cụ gỡ lỗi, nhưng chỉ cần vài phút nghiên cứu kết quả profiling có thể giúp phát hiện những khiếm khuyết bất ngờ. Ví dụ, một đoạn mã quản lý bộ nhớ tưởng là điểm nghẽn hiệu năng, sau khi thay đổi thuật toán nhưng performance không cải thiện, việc dùng profiler có thể giúp xác định chính chỗ khác mới là nguyên nhân gây lãng phí thời gian. Nhờ đó, bạn tránh tối ưu nhầm chỗ.

Hãy kiểm tra kết quả của execution profiler để đảm bảo chương trình của bạn phân bổ thời gian hợp lý ở từng phần.

## Khung kiểm thử/chống đỡ (Test Frameworks/Scaffolding)

Như đã đề cập, việc tách đoạn mã dễ lỗi ra, viết mã kiểm tra cho phần đó và chạy độc lập thường là cách hiệu quả nhất để truy vết khiếm khuyết.

## Bộ gỡ lỗi (Debuggers)

### Khả năng của bộ gỡ lỗi hiện đại

Các **debugger** thương mại ngày càng phát triển, mang đến những khả năng thay đổi cách bạn lập trình:

- Đặt breakpoint để dừng thực thi ở dòng mã cụ thể, hoặc khi biến toàn cục thay đổi, hoặc biến được gán giá trị xác định.
- Duyệt từng dòng mã, “bước vào” hoặc “bước qua” các hàm.
- Hỗ trợ chạy ngược chương trình, quay về thời điểm khi khiếm khuyết xuất hiện.

- Ghi lại việc thực thi các câu lệnh (giống như in ra thông báo “Tôi đang ở đây!” ở nhiều chỗ khác nhau trong chương trình).
- Đầy đủ chức năng quan sát, chỉnh sửa giá trị dữ liệu, kể cả dữ liệu động, danh sách liên kết,...
- Hiện thị chính xác ngôn ngữ lập trình đang dùng ở từng đoạn (nếu chương trình dùng đa ngôn ngữ).
- Ghi nhớ breakpoint, các biến theo dõi... theo từng chương trình, giúp tiết kiệm thời gian cấu hình khi gỡ lỗi nhiều lần.
- Đối với chương trình nhạy cảm về thời gian hoặc bộ nhớ, các **system debugger** hoạt động ở cấp hệ thống sẽ không tác động đến hoạt động của chương trình ứng dụng.

Một debugger tương tác là ví dụ tiêu biểu cho công cụ không bắt buộc phải dùng—nó có thể làm gia tăng kiểu “hack mò mẫm” thay vì thiết kế có hệ thống, nhưng không thể phủ nhận sức mạnh của nó trong lập trình thực tiễn.

## Tranh cãi quanh việc sử dụng debugger

Một số chuyên gia cho rằng nên sử dụng tư duy thay vì phụ thuộc quá nhiều vào debugger. Dù vậy, công cụ mạnh mẽ nào cũng có thể bị lạm dụng, nhưng lý do đó không đủ để loại bỏ chúng hoàn toàn. Giống như không nên tránh uống aspirin chỉ vì lo sợ quá liều. Debugger không thể thay thế hoàn toàn tư duy logic, nhưng trong một số trường hợp, tư duy không thay thế được debugger. Sự kết hợp cả hai là lựa chọn tối ưu.

# CHECKLIST: Nhắc nhở khi gỡ lỗi

## Kỹ thuật tìm kiếm khiếm khuyết

- Sử dụng toàn bộ dữ liệu hiện có để hình thành giả thuyết.
- Tinh chỉnh các test case để tái hiện lỗi.
- Sử dụng bộ test đơn vị (unit test suite).
- Sử dụng tất cả công cụ có thể.
- Tái hiện lỗi theo nhiều cách khác nhau.
- Sinh thêm dữ liệu để tạo thêm giả thuyết.
- Phân tích kết quả của các trường hợp kiểm thử âm (negative tests).
- Động não để nảy sinh thêm giả thuyết.
- Ghi chú lại các ý tưởng, phương án thử nghiệm.
- Thu hẹp hoặc mở rộng khu vực nghi vấn trong mã.
- Đặc biệt chú ý đến các class/routine từng có lỗi.
- Kiểm tra mã vừa bị thay đổi gần đây.
- Tiến hành tích hợp từng bước nhỏ.
- Kiểm tra các lỗi phổ biến.
- Nhờ đồng nghiệp thảo luận vấn đề.
- Tạm thời dừng lại, nghỉ ngơi.
- Đặt giới hạn thời gian cho các chiến thuật gỡ lỗi thô sơ.
- Lập danh sách các kỹ thuật “brute-force” và thử nghiệm chúng.

## Kỹ thuật xử lý lỗi cú pháp

- Không hoàn toàn tin vào số dòng trong thông báo lỗi của trình biên dịch.
- Không quá tin thông điệp mà trình biên dịch cung cấp.
- Chia để trị - “divide and conquer”.
- Sử dụng editor hỗ trợ kiểm soát sai lệch cú pháp (syntax-directed editor).

## Kỹ thuật sửa chữa khiếm khuyết

- Hiểu rõ bản chất vấn đề trước khi sửa.
- Hiểu rõ chương trình, không chỉ vấn đề.
- Xác nhận chắc chắn chẩn đoán lỗi.
- Bình tĩnh, không hấp tấp.
- Lưu lại mã nguồn gốc ban đầu.
- Sửa tận gốc vấn đề, không chỉ biểu hiện bề ngoài.
- Thay đổi mã chỉ khi có lý do rõ ràng.
- Sửa từng thay đổi một.
- Kiểm tra kỹ lưỡng sau khi sửa.
- Thêm unit test cho lỗi vừa sửa.
- Tìm kiếm các lỗi tương tự.

## Phương pháp tổng quát khi gỡ lỗi

- Xem gỡ lỗi là cơ hội học thêm về chương trình, về lỗi, về chất lượng mã và về cách giải quyết vấn đề.
  - Tránh kiểu gỡ lỗi mò mẫm, thiếu hệ thống.
  - Luôn cho rằng lỗi là do mình trước.
  - Vận dụng phương pháp khoa học khi xử lý các lỗi khó tái hiện.
  - Kết hợp nhiều phương pháp tìm lỗi, không dập khuôn.
  - Xác thực rằng lỗi đã được sửa triệt để.
  - Kết hợp sử dụng mọi nguồn lực: cảnh báo, profiler, framework kiểm thử, scaffolding và debugger tương tác.
- 

## Tài liệu tham khảo thêm

- Agans, David J. *Debugging: The Nine Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems*. Amacom, 2003: Nguyên tắc gỡ lỗi cần bản phù hợp cho mọi ngôn ngữ, môi trường.
  - Myers, Glenford J. *The Art of Software Testing*, John Wiley & Sons, 1979: Chương 7 dành riêng cho chủ đề gỡ lỗi.
  - Allen, Eric. *Bug Patterns In Java*, Apress, 2002: Tiếp cận gỡ lỗi trong Java, nhấn mạnh “Phương pháp Khoa học trong Gỡ lỗi” (Scientific Method of Debugging).
  - Robbins, John. *Debugging Applications for Microsoft .NET and Microsoft Windows*, Microsoft Press, 2003.
  - McKay, Everett N. và Mike Woodring. *Debugging Windows Programs: Strategies, Tools, and Techniques for Visual C++ Programmers*, Addison-Wesley, 2000.
- 

## Những điểm then chốt

- **Debugging** là yếu tố quyết định thành bại trong phát triển phần mềm. Hạn chế lỗi từ đầu là ưu việt, nhưng kỹ năng gỡ lỗi vẫn cần thiết, bởi hiệu suất giữa người gỡ lỗi thuần thục và kém chênh lệch ít nhất 10 lần.
  - Cần có phương pháp hệ thống trong tìm kiếm và sửa lỗi. Hãy sử dụng “Phương pháp Khoa học trong Gỡ lỗi”.
  - Hiểu căn cứ gốc rễ vấn đề trước khi sửa. Việc phỏng đoán hay sửa đại chỉ làm mọi thứ tệ hơn.
  - Luôn đặt cảnh báo của trình biên dịch ở mức cao nhất và sửa các lỗi rõ ràng, để tránh việc lỗi nhỏ tích tụ thành lỗi lớn hơn.
  - Các công cụ gỡ lỗi là trợ thủ mạnh mẽ cho quá trình phát triển phần mềm—hãy kết hợp chúng với tư duy của bạn.
- 

## Refactoring (Tái cấu trúc mã nguồn)

### 24.1 Các loại tiến hóa phần mềm (Kinds of Software Evolution)

Tiến hóa phần mềm (software evolution) giống như tiến hóa sinh học: một số “đột biến” (mutations) có lợi, phần lớn thì không. Tiến hóa phần mềm tốt sẽ giúp mã nguồn ngày càng hoàn thiện, theo lộ trình từ đơn giản đến phức tạp hơn. Nhưng nếu tiến hóa tiêu cực, chương trình sẽ đi vào vòng xoáy thoái hóa.

Điểm khác biệt cốt lõi giữa các loại tiến hóa phần mềm là chất lượng chương trình có tăng lên hay giảm sút sau khi chỉnh sửa. Nếu bạn chỉ vá lỗi tạm bợ với “băng keo logic” và sự mê tín, chất lượng sẽ đi xuống. Nếu mỗi lần chỉnh sửa, bạn xem đó là cơ hội củng cố thiết kế, chất lượng sẽ nâng cao.

Một khác biệt quan trọng khác là thay đổi trong giai đoạn xây dựng (construction) khác với thay đổi ở giai đoạn bảo trì (maintenance): Thay đổi thời xây dựng do chính các lập trình viên ban đầu thực hiện, khi hệ thống còn chưa “lên sóng”, áp lực chỉ là tiến độ chứ không phải do hàng trăm người dùng than phiền.

### Triết lý tiến hóa phần mềm

Một điểm yếu phổ biến là nhiều lập trình viên để quá trình tiến hóa mã diễn ra theo bản năng, không có chiến lược. Nếu bạn nhận ra tiến hóa là điều tất yếu, hãy chủ động tận dụng nó—coi đây là cơ hội hoàn thiện mã nguồn. Mỗi lần sửa đổi là dịp để hệ thống dễ bảo trì hơn về sau. Hãy luôn viết code cũng như các chỉnh sửa với tư duy “sẽ còn phải thay đổi tiếp”.

### Luật tối thượng về tiến hóa phần mềm

**Mọi sự tiến hóa phải làm tăng chất lượng bên trong của chương trình.**

### 24.2 Giới thiệu về Refactoring

**Chiến lược then chốt để tuân thủ Luật Tối Thượng của Tiến hóa phần mềm** chính là Refactoring. Theo Martin Fowler, refactoring là “thay đổi cấu trúc bên trong của phần mềm để dễ hiểu và rõ hơn khi sửa đổi, nhưng không thay đổi hành vi quan

sát được” (Fowler 1999).

Thuật ngữ “refactoring” thời hiện đại xuất phát từ từ “factoring”—việc chia nhỏ một chương trình thành các thành phần nhỏ nhất có thể.

## Lý do thực hiện Refactoring

Đôi khi code bị xuống cấp sau bảo trì, hoặc nó vốn dĩ không tốt từ đầu. Một số dấu hiệu cảnh báo (còn gọi là *smells* theo Fowler) cho thấy cần refactoring:

- **Mã bị lặp lại:** Đây gần như luôn là biểu hiện chưa phân tách thiết kế một cách hoàn chỉnh. Copy và paste là lỗi thiết kế (theo David Parnas). Nguyên tắc DRY (Don't Repeat Yourself) nhấn mạnh tuyệt đối tránh lặp lại mã.
- **Hàm (routine) quá dài:** Trong lập trình hướng đối tượng, một hàm dài hơn một màn hình hiếm khi cần thiết. Đây thường là dấu hiệu bị “ép” thiết kế “structured programming” vào mô hình hướng đối tượng.
- **Vòng lặp quá dài hoặc lồng ghép sâu:** Phần thân vòng lặp (loop body) thường nên tách thành các hàm riêng để giảm độ phức tạp.
- **Class có cohesion kém:** Nếu một class “ôm đồm” quá nhiều trách nhiệm khác biệt, cần tách thành nhiều class nhỏ hơn, mỗi class đảm nhận một nhóm chức năng liên quan.
- **Giao diện class mất sự nhất quán trong mức độ trừu tượng:** Các class ban đầu có thể cohesive, nhưng theo thời gian, nếu chỉnh sửa tùy tiện, giao diện của chúng sẽ bị “bê cong”, làm giảm tính toàn vẹn của interface.

## Các Dấu Hiệu Nhận Biết Cần Refactoring

### Danh sách tham số quá nhiều

Các chương trình được thiết kế tốt (well-factored programs) thường bao gồm nhiều thủ tục nhỏ, được định nghĩa rõ ràng và không cần tới danh sách tham số dài. Một danh sách tham số quá dài là dấu hiệu cho thấy việc trừu tượng hóa (abstraction) giao diện của thủ tục chưa được xem xét kỹ lưỡng.

### Sự thay đổi trong một lớp thường được khoanh vùng

Đôi khi, một lớp đảm nhận hai hoặc nhiều trách nhiệm khác nhau, dẫn đến việc bạn chỉ thay đổi một phần lớp hoặc phần khác; ít khi cả hai phần bị ảnh hưởng đồng thời. Đây là dấu hiệu cho việc lớp nên được tách thành nhiều lớp tương ứng với từng trách nhiệm riêng biệt.

### Thay đổi yêu cầu sửa đổi song song nhiều lớp

Khi bạn phải cập nhật khoảng 15 lớp mỗi khi thêm loại đầu ra (output) mới, đó là dấu hiệu mã nguồn cần được tổ chức lại để thay đổi chỉ ảnh hưởng đến một lớp. Mặc dù lý tưởng này khó đạt được, nhưng nó là mục tiêu cần hướng tới.

### Các hệ kế thừa phải được chỉnh sửa song song

Việc phải tạo subclass cho một lớp mỗi khi tạo subclass cho lớp khác là dấu hiệu cần chú ý. Đây là dạng đặc biệt của sửa đổi song song (parallel modification) và nên giải quyết triệt để.

### Câu lệnh case phải chỉnh sửa song song

Nếu bạn phải thay đổi cùng lúc nhiều câu lệnh `case` tương tự nhau ở nhiều phần khác nhau trong chương trình, hãy cân nhắc liệu kế thừa (inheritance) có thể là giải pháp tốt hơn.

### Dữ liệu liên quan không được tổ chức vào lớp

Nếu bạn thường xuyên xử lý cùng một nhóm dữ liệu, hãy xem xét gộp chúng vào một lớp riêng.

### Thủ tục sử dụng nhiều đặc trưng của lớp khác hơn lớp hiện tại

Nếu một thủ tục (routine) dùng nhiều đặc trưng của lớp khác, hãy cân nhắc chuyển thủ tục đó sang lớp kia và gọi từ lớp cũ.

### Kiểu dữ liệu sơ cấp (primitive data type) bị lạm dụng

Kiểu dữ liệu sơ cấp có thể đại diện cho nhiều thực thể ngoài đời thật. Nếu chương trình dùng `int` để biểu diễn tiền, hãy cân nhắc tạo lớp `Money` để kiểm tra kiểu rõ ràng, thêm kiểm soát giá trị, v.v. Nếu `Money` và `Temperature` đều là `int`, trình biên

dịch sẽ không cảnh báo khi gán sai kiểu, ví dụ `bankBalance = recordLowTemperature;`.

## Lớp không làm được nhiều việc

Sau khi refactoring, có thể một lớp không còn tác dụng rõ rệt. Khi đó, hãy cân nhắc chuyển các trách nhiệm đó cho các lớp khác và loại bỏ lớp đó.

## Dữ liệu chuyển qua chuỗi thủ tục trung gian

Nếu dữ liệu chỉ được chuyển từ thủ tục này qua thủ tục khác mà không xử lý gì, gọi là "tramp data". Hãy kiểm tra xem việc chuyển dữ liệu đó có thực sự phù hợp với trừu tượng (abstraction) của các thủ tục hay không.

## Đối tượng trung gian không làm việc gì đáng kể

Nếu một lớp chủ yếu chỉ chuyển tiếp lời gọi tới lớp khác, hãy cân nhắc loại bỏ đối tượng trung gian này và gọi trực tiếp tới lớp đích.

## Hai lớp quá thân mật với nhau

**Encapsulation** (che giấu thông tin) là công cụ mạnh nhất để kiểm soát độ phức tạp và giảm hiệu ứng dây chuyền khi thay đổi mã. Nếu một lớp biết quá nhiều về lớp khác, hãy tăng cường sự tách biệt (encapsulation).

## Tên thủ tục không rõ ràng

Nếu tên thủ tục gây nhầm lẫn, hãy đổi tên tại nơi định nghĩa, tất cả nơi sử dụng, sau đó biên dịch lại.

## Thành viên dữ liệu là public

Việc để dữ liệu public là ý tưởng tồi vì làm mờ ranh giới giữa interface và implementation, vi phạm encapsulation và giảm tính linh hoạt. Nên ẩn chúng qua các routine truy cập.

## Subclass chỉ sử dụng phần nhỏ các phương thức của superclass

Điều này thường xuất phát từ việc subclass được tạo ra không phải vì lý do kế thừa hợp lý mà chỉ để tái sử dụng hàm. Hãy chuyển mối quan hệ "is-a" thành "has-a" - superclass thành biến thành viên của subclass và chỉ công khai những hàm thật sự cần thiết trong subclass.

## Bình luận dùng để giải thích code khó hiểu

Bình luận quan trọng nhưng không nên dùng để "bù đắp" cho mã xấu. Theo lời khuyên: “Đừng giải thích mã xấu – hãy sửa lại nó.”

## Sử dụng biến toàn cục (global variable)

Hãy kiểm tra lại bất cứ khi nào gặp biến toàn cục. Có thể giờ bạn đã biết cách tái cấu trúc để tránh chúng, hoặc đưa chúng vào access routines để kiểm soát tốt hơn.

## Dùng mã thiết lập/takedown trước/sau lệnh gọi thủ tục

Có dấu hiệu bất ổn nếu quá trình gọi một hàm cần một loạt thiết lập biến trước khi gọi và "tháo dỡ" sau khi gọi. Ví dụ:

```
WithdrawalTransaction withdrawal;  
withdrawal.SetCustomerId( customerId );  
withdrawal.SetBalance( balance );  
withdrawal.SetWithdrawalAmount( withdrawalAmount );  
withdrawal.SetWithdrawalDate( withdrawalDate );  
ProcessWithdrawal( withdrawal );  
customerId = withdrawal.GetCustomerId();  
balance = withdrawal.GetBalance();  
withdrawalAmount = withdrawal.GetWithdrawalAmount();  
withdrawalDate = withdrawal.GetWithdrawalDate();
```

Hãy xem liệu interface của Routine có đang thể hiện đúng abstraction không. Có thể biến đổi hàm thành:

```
ProcessWithdrawal( customerId, balance, withdrawalAmount, withdrawalDate );
```

Hoặc, nếu luôn có sẵn một đối tượng `WithdrawalTransaction`, hãy truyền nguyên object này vào routine:

```
ProcessWithdrawal( withdrawal );
```

Bản chất vấn đề nằm ở abstraction: Routine có thực sự cần 4 dữ liệu riêng lẻ hay chỉ cần object?

## Chương trình có mã nghĩ là "cần thiết trong tương lai"

Lập trình viên thường dự đoán sai yêu cầu tương lai. Kết quả:

- Yêu cầu chưa rõ ràng, mã "design ahead" thường bị loại bỏ.
- Dù dự đoán gần đúng, vẫn khó bao quát hết phức tạp, mã vẫn bị loại.
- Người khác nghĩ mã này ổn mà thực chất chỉ là "dự đoán", dễ gây lỗi và tốn thời gian.
- Giá tăng độ phức tạp, chi phí kiểm thử, sửa lỗi cũng tăng.

Các chuyên gia đều thống nhất: tốt nhất là chỉ viết những gì thực sự cần, rõ ràng; giúp người sau dễ chỉnh sửa đúng đắn.

---

## Checklist: Những lý do cần Refactor

- ☐ Mã bị lặp lại (duplicated).
- ☐ Một thủ tục quá dài.
- ☐ Một vòng lặp quá dài hoặc lồng quá sâu.
- ☐ Lớp cohesion kém.
- ☐ Giao diện lớp không nhất quán về mức độ trừu tượng.
- ☐ Danh sách tham số quá nhiều.
- ☐ Thay đổi trong lớp được khoanh vùng.
- ☐ Thay đổi yêu cầu sửa đổi nhiều lớp song song.
- ☐ Hệ kế thừa phải được cập nhật song song.
- ☐ Câu lệnh case phải chỉnh song song.
- ☐ Các dữ liệu liên quan không được tổ chức thành lớp.
- ☐ Thủ tục dùng nhiều đặc trưng lớp khác hơn lớp hiện tại.
- ☐ Kiểu dữ liệu primitive bị lạm dụng.
- ☐ Lớp không làm gì đáng kể.
- ☐ Chuỗi thủ tục truyền dữ liệu kiểu tramp data.
- ☐ Đối tượng "middleman" không làm gì.
- ☐ Lớp này biết quá nhiều về lớp khác.
- ☐ Tên thủ tục tồi.
- ☐ Dữ liệu public.
- ☐ Subclass chỉ dùng rất ít phương thức của superclass.
- ☐ Dùng bình luận để giải thích code khó.
- ☐ Dùng biến toàn cục.
- ☐ Có mã thiết lập hoặc tháo dỡ trước/sau gọi thủ tục.
- ☐ Có đoạn mã giả định sẽ cần dùng tương lai.

---

## Lý do KHÔNG nên refactor

Trong giao tiếp hàng ngày, "refactoring" thường bị hiểu sai thành bất kỳ sự thay đổi nào trong code, từ sửa lỗi, thêm chức năng đến sửa đổi thiết kế. Thực ra, thay đổi tự thân không có giá trị, chỉ khi thay đổi có chủ đích, có kỷ luật mới là chiến lược nhằm cải thiện chất lượng phần mềm và ngăn cản chu trình "thoái hóa" (software-entropy death spiral).

---

## 24.3 Một số refactoring cụ thể

### Refactoring cấp dữ liệu (Data-Level Refactorings)

- Thay giá trị ma thuật (magic number) bằng hằng số có tên



Nếu bạn dùng literal số/chuỗi như `3.14`, hãy thay thế bằng hằng số có tên như `PI`.

- **Đổi tên biến trở nên rõ ràng hơn**

Nếu tên biến không rõ ý, đổi tên cho dễ hiểu. Áp dụng tương tự với hằng số, lớp, routine.

- **Thay biểu thức bằng routine**

Nếu có biểu thức lặp lại nhiều chỗ, đóng gói thành routine để tiện bảo trì.

- **Giới thiệu biến trung gian**

Dùng biến trung gian để lưu tạm giá trị biểu thức giúp làm rõ mục đích.

- **Tách biến đa dụng thành nhiều biến đơn dụng**

Nếu một biến dùng cho nhiều ý nghĩa khác nhau, hãy chia nhỏ thành các biến với tên cụ thể.

- **Dùng biến cục bộ thay vì thay đổi tham số**

Nếu tham số đầu vào được dùng như biến cục bộ, nên tạo và dùng biến cục bộ riêng.

- **Chuyển primitive thành class**

Giúp tăng kiểm soát kiểu, mở rộng hành vi, ví dụ `Money`, `Temperature`, `Color`, `Shape`, `OutputType`.

- **Chuyển mảng thành object**

Nếu một mảng chứa nhiều loại dữ liệu, hãy tạo class với thuộc tính tương ứng.

- **Encapsulate a collection**

Nếu lớp trả về collection, có thể gây khó đồng bộ. Nên trả về collection read-only và thêm phương thức để quản lý thêm/xóa.

- **Thay record truyền thống bằng data class**

Gói các trường của record thành class để tập trung kiểm soát lỗi, kiểm tra, lưu trữ, v.v.

---

## Refactoring cấp lệnh (Statement-Level Refactoring)

- **Phân rã biểu thức boolean phức tạp**

Tạo biến trung gian giúp mã dễ hiểu.

- **Đưa biểu thức boolean phức tạp vào hàm riêng**

Tăng khả năng đọc và giảm sai sót khi dùng lại nhiều lần.

- **Hợp nhất mã bị lặp trong các nhánh điều kiện**

Nếu đoạn lặp nằm cuối cả hai nhánh `if/else`, hãy đưa ra ngoài.

- **Dùng `break/return` thay biến điều khiển vòng lặp**

Thay vì dùng biến như `done`, hãy dùng `break` hoặc `return`.

- **Trả về giá trị ngay khi biết kết quả**

Tránh đặt giá trị trả về trong các nhánh lồng nhau, giúp mã dễ đọc hơn.

- **Thay điều kiện/phép so sánh lặp lại bằng polymorphism**

Dùng đa hình thay vì lặp lại các câu lệnh `case`.

- **Dùng object null thay vì kiểm tra null nhiều lần**

Thay vì kiểm tra null trong nhiều client code, dùng null object với hành vi mặc định.

---

## Refactoring cấp thủ tục (Routine-Level Refactoring)

- **Tách routine/method từ mã inline**

Thu nhỏ routine bằng cách tách đoạn mã con ra thành routine riêng.

- **Đưa mã routine đơn giản về inline**

Nếu routine quá đơn giản, nên inline tại chỗ dùng.

- **Chuyển routine dài thành class**

Routine quá dài có thể đổi thành class rồi phân rã tiếp thành routine nhỏ.

- **Thay thuật toán phức tạp bằng thuật toán đơn giản hơn**

Nếu có thể, thay thế để dễ bảo trì hơn.

- **Thêm hoặc loại bỏ tham số**

Điều chỉnh khi routine cần nhận thêm/bớt thông tin.

- **Tách query và modification**

Nếu routine vừa truy vấn vừa thay đổi, tách ra hai routine riêng biệt.

- **Hợp nhất routine tương tự thành routine dùng tham số**

Nếu hai routine chỉ khác nhau về hằng số, hợp nhất và truyền giá trị khác nhau qua tham số.

- **Tách routine có hành vi tùy tham số**

Nếu routine thực thi khác nhau dựa trên giá trị tham số, hãy tách thành các routine riêng lẻ.

- **Truyền object nguyên vẹn thay vì nhiều trường riêng lẻ**

Nếu thường xuyên truyền nhiều giá trị của object vào routine, hãy đổi lại truyền trực tiếp object.

---

*Chú thích thêm:*

- **Routine:** Hàm/thủ tục trong lập trình.
  - **Encapsulation (che giấu thông tin):** Nguyên lý giấu trạng thái nội bộ của object, chỉ cung cấp interface ra bên ngoài.
  - **Polymorphism (đa hình):** Cơ chế để đối tượng có thể thay đổi hành vi khi bị gọi thông qua interface chung.
  - **Refactoring:** Tái cấu trúc mã nguồn mà không thay đổi hành vi bên ngoài của chương trình.
- 

## Bao đóng quá trình downcasting

Nếu một **routine** (thủ tục/hàm) trả về một đối tượng, thông thường nó nên trả về loại đối tượng cụ thể nhất mà nó biết. Nguyên tắc này đặc biệt áp dụng cho các routine trả về *iterator* (bộ lặp), *collection* (bộ sưu tập), phần tử của collection, v.v.

## Các phương pháp tái cấu trúc cấp lớp (Class Implementation Refactorings)

Dưới đây là các phương pháp tái cấu trúc giúp cải thiện ở cấp lớp:

### Thay đổi value objects thành reference objects

Nếu bạn liên tục tạo và duy trì nhiều bản sao của các đối tượng lớn hoặc phức tạp, hãy thay đổi cách sử dụng các đối tượng này để chỉ duy trì một bản chính (*value object* - *đối tượng giá trị*) và các phần còn lại của mã sử dụng tham chiếu đến đối tượng đó (*reference object* - *đối tượng tham chiếu*).

### Thay đổi reference objects thành value objects

Nếu bạn gặp phải việc phải quản lý rất nhiều tham chiếu cho các đối tượng nhỏ hoặc đơn giản, hãy thay đổi việc sử dụng các đối tượng đó để tất cả đều là *value objects*.

## Thay thế virtual routines bằng khởi tạo dữ liệu

Nếu bạn có một tập các lớp con khác biệt chỉ dựa trên các giá trị hằng số mà chúng trả về, thay vì override các member routine (hàm thành viên) ở các lớp dẫn xuất, hãy để các lớp dẫn xuất khởi tạo lớp với các giá trị hằng số phù hợp, và sau đó để mã tổng quát trong lớp cha làm việc với các giá trị đó.

## Thay đổi vị trí routine hoặc dữ liệu thành viên

Hãy cân nhắc thực hiện một số thay đổi tổng quát trong hệ phân cấp kế thừa nhằm loại bỏ sự trùng lặp ở các lớp dẫn xuất:

- Kéo một routine lên lớp cha (superclass)
- Kéo một biến trường (field) lên lớp cha
- Kéo thân hàm tạo lên lớp cha

Một số thay đổi khác thường hỗ trợ cho việc chuyên biệt hóa tại các lớp dẫn xuất:

- Đẩy một routine xuống các lớp dẫn xuất
- Đẩy một field xuống các lớp dẫn xuất
- Đẩy thân hàm tạo xuống các lớp dẫn xuất

## Tách mã chuyên biệt ra subclass

Nếu một lớp chứa mã chỉ được sử dụng bởi một tập con các thực thể của nó, hãy chuyển đoạn mã đó vào một lớp con riêng biệt.

## Kết hợp mã tương tự vào superclass

Nếu hai lớp con có mã tương tự, hãy kết hợp và chuyển mã đó lên lớp cha.

---

## Các phương pháp tái cấu trúc giao diện lớp (Class Interface Refactorings)

Các phương pháp tái cấu trúc này giúp tạo ra các giao diện lớp tốt hơn:

### Chuyển routine sang lớp khác

Tạo routine mới ở lớp đích, chuyển thân routine từ lớp nguồn sang lớp đích và gọi routine mới này từ routine cũ.

### Chuyển một lớp thành hai lớp

Nếu một lớp đảm nhận hai hay nhiều trách nhiệm riêng biệt, hãy chia lớp thành nhiều lớp, mỗi lớp có một trách nhiệm rõ ràng.

### Loại bỏ một lớp

Nếu một lớp không đảm nhận nhiều chức năng, hãy chuyển mã của nó vào các lớp khác gắn kết hơn và loại bỏ lớp đó.

### Ẩn delegate

Đôi khi, Lớp A gọi Lớp B và Lớp C, trong khi thực ra Lớp A chỉ nên gọi Lớp B và Lớp B sẽ gọi Lớp C. Xác định abstraction (trừu tượng hóa) đúng đắn cho sự tương tác giữa A và B, và để B chịu trách nhiệm gọi C.

### Loại bỏ middleman

Nếu Lớp A gọi Lớp B và Lớp B gọi Lớp C, đôi khi tối ưu hơn khi để Lớp A gọi Lớp C trực tiếp.

## Thay thế inheritance bằng delegation

Nếu một lớp cần sử dụng lớp khác nhưng muốn kiểm soát chặt chẽ hơn giao diện, hãy biến superclass (lớp cha) thành một trường thành viên và cung cấp các routine phù hợp nhằm tạo abstraction liền mạch.

## Thay thế delegation bằng inheritance

Nếu một lớp công khai mọi routine public của một delegate class (lớp thành viên), hãy cho lớp đó kế thừa trực tiếp từ delegate class thay vì chỉ sử dụng.

## Giới thiệu foreign routine

Nếu một lớp cần routine bổ sung mà bạn không thể chỉnh sửa lớp để cung cấp nó, hãy tạo routine mới trong lớp client để cung cấp chức năng đó.

## Giới thiệu extension class

Nếu một lớp cần nhiều routine bổ sung mà không thể sửa trực tiếp, hãy tạo lớp mới kết hợp chức năng của lớp không thể chỉnh sửa với các routine thêm. Có thể thực hiện bằng kế thừa hoặc bằng wrapper (lớp bọc).

## Bao đóng biến thành viên công khai (encapsulate an exposed member variable)

Nếu dữ liệu thành viên là public, hãy đổi thành private và cung cấp giá trị thông qua một routine.

## Loại bỏ routine Set() cho những trường không được thay đổi

Nếu một trường chỉ nên thiết lập khi tạo đối tượng và không thay đổi sau đó, hãy khởi tạo nó trong constructor thay vì cung cấp routine Set() gây hiểu lầm.

## Ẩn routine không dùng ngoài lớp

Nếu interface của lớp sẽ mạch lạc hơn nếu không có một routine, hãy ẩn routine đó.

## Bao đóng routine không sử dụng

Nếu bạn chỉ thường sử dụng một phần giao diện của một lớp, hãy tạo giao diện mới chỉ bao gồm các routine cần thiết.

## Sát nhập superclass và subclass khi triển khai rất giống nhau

Nếu subclass không mở rộng nhiều chức năng riêng, hãy kết hợp nó vào superclass.

---

## Các phương pháp tái cấu trúc cấp hệ thống (System-Level Refactorings)

Các refactoring sau giúp nâng cao mã ở cấp toàn bộ hệ thống:

### Tạo nguồn tham chiếu dữ liệu đáng tin cậy khi bạn không kiểm soát được dữ liệu

Đôi khi hệ thống duy trì dữ liệu mà bạn không thể truy cập thuận tiện, ví dụ dữ liệu trong một kiểm soát GUI (Graphical User Interface). Hãy tạo một lớp phản ánh dữ liệu trong điều khiển GUI, và để cả hai bên cùng sử dụng nó như nguồn dữ liệu chính.

### Chuyển quan hệ lớp một chiều thành hai chiều

Nếu hai lớp cần sử dụng chức năng của nhau nhưng hiện chỉ một lớp biết về lớp còn lại, hãy cho phép cả hai lớp biết về nhau.

### Chuyển quan hệ hai chiều thành một chiều

Nếu hai lớp biết về chức năng nhau nhưng chỉ có một lớp thực sự cần biết về lớp còn lại, hãy đổi về một chiều.

### Cung cấp factory method thay cho constructor đơn giản

Sử dụng factory method (phương thức tạo đối tượng) khi cần tạo đối tượng dựa trên mã loại (type code), hoặc khi muốn làm việc với *reference objects* thay vì *value objects*.

### Thay thế mã lỗi bằng exception hoặc ngược lại

Tùy thuộc vào chiến lược xử lý lỗi, hãy sử dụng phương thức chuẩn (ví dụ: exception) thay cho việc trả về mã lỗi.

---

## CHECKLIST: Tóm tắt các loại refactoring

### Refactoring cấp dữ liệu

- Thay thế *magic number* (số kỳ diệu) bằng hằng số có tên.
- Đổi tên biến cho rõ ràng, dễ hiểu hơn.
- Di chuyển biểu thức vào một vị trí.
- Thay biểu thức bằng routine.
- Giới thiệu biến trung gian.
- Chuyển biến dùng nhiều lần thành nhiều biến dùng một lần.
- Dùng biến cục bộ cho mục đích cục bộ thay vì parameter (tham số).
- Chuyển *data primitive* thành class.
- Chuyển tập hợp mã loại thành class hoặc enumeration (liệt kê).
- Chuyển tập mã loại thành class có các subclass.
- Đổi mảng thành đối tượng.
- Bao đóng một collection.
- Thay record truyền thống bằng data class.

## Refactoring cấp câu lệnh (Statement-Level)

- Phân tách biểu thức boolean.
- Đưa biểu thức boolean phức tạp vào hàm boolean có tên.
- Hợp nhất các đoạn bị lặp trong các nhánh điều kiện.
- Sử dụng break hoặc return thay vì biến điều khiển vòng lặp.
- Return sớm nhất có thể, thay vì gán giá trị trả về trong nhiều tầng if-then-else lồng nhau.
- Thay các điều kiện lặp lại bằng polymorphism (đa hình).
- Tạo và sử dụng *null objects* thay vì kiểm tra giá trị null.

## Refactoring cấp routine (Routine-Level)

- Trích xuất routine.
- Đưa code trong routine vào nội tuyến.
- Chuyển routine dài thành class.
- Thay thế thuật toán phức tạp bằng thuật toán đơn giản.
- Thêm/thay đổi/loại bỏ parameter.
- Tách nhóm câu truy vấn khỏi câu thay đổi dữ liệu.
- Kết hợp các routine tương tự bằng cách tham số hóa.
- Tách các routine tùy theo hành vi parameter.
- Truyền cả object thay vì field cụ thể, hoặc ngược lại.
- Bao đóng quá trình downcasting.

## Refactoring cấp lớp

- Xem chi tiết ở phần trên.

## Refactoring cấp hệ thống

- Xem chi tiết ở phần trên.

---

# Phương pháp Refactoring An Toàn

Refactoring là kỹ thuật mạnh để nâng cao chất lượng mã nguồn. Như mọi công cụ mạnh mẽ khác, refactoring có thể gây tác hại nếu sử dụng sai. Một số nguyên tắc cơ bản giúp ngăn ngừa lỗi khi thực hiện refactoring:

## Lưu lại mã gốc trước khi bắt đầu

Trước khi refactor, đảm bảo bạn có thể khôi phục về trạng thái ban đầu. Hãy lưu version trong hệ thống quản lý phiên bản, hoặc sao chép các tệp tin vào thư mục backup.

## Thực hiện refactoring nhỏ

Giữ cho refactoring đủ nhỏ để dễ hiểu toàn bộ tác động của nó.

## Thực hiện từng refactoring một

Trừ những refactoring đơn giản nhất, sau mỗi thao tác, hãy biên dịch và kiểm thử trước khi làm tiếp.

## Lập danh sách các bước dự định thực hiện

Để giữ được bối cảnh và kiểm soát quá trình thay đổi.

## Tạo "parking lot"

Nếu trong khi refactor phát hiện cần thực hiện refactoring khác, hãy ghi chú vào danh sách "parking lot" để xử lý sau.

## Tạo checkpoint thường xuyên

Lưu lại trạng thái tại các điểm kiểm tra để có thể quay lại trạng thái ổn định.

## Sử dụng cảnh báo của compiler

Thiết lập mức cảnh báo nghiêm ngặt để phát hiện sớm các lỗi nhỏ.

## Retest

Phải test lại mã đã sửa, cả test case cũ lẫn thêm test unit mới cho mã mới.

## Xem xét lại thay đổi

Việc review thay đổi càng quan trọng hơn sau mỗi refactor.

## Điều chỉnh tùy theo mức rủi ro

Với các refactoring có rủi ro cao (thay đổi interface, schema, boolean logic, v.v.), cần cẩn thận hơn: một lần một thay đổi, kiểm tra, review chéo hoặc lập trình đôi.

---

## Thời điểm không nên refactor

Refactoring mạnh mẽ nhưng không phải là vạn năng và có thể bị lạm dụng:

- **Đừng dùng refactoring che đậy việc vá lỗi**  
Refactoring là thay đổi trên code *đang hoạt động ổn định* mà không thay đổi hành vi chương trình. Vá lỗi là chuyện khác.
- **Không refactor khi nên viết lại hoàn toàn**  
Đôi lúc code cần viết lại mới chứ không chỉ refactor từng bước nhỏ.

---

## Chiến lược Refactoring

Quá nhiều refactoring cũng không hiệu quả. Hãy tập trung vào 20% mang lại 80% giá trị.

Các hướng dẫn quan trọng:

1. Refactor khi thêm routine mới: Sắp xếp lại code liên quan khi thêm hàm mới.
2. Refactor khi thêm class mới: Đây cũng là dịp sắp xếp lại các class liên quan.
3. Refactor khi sửa lỗi: Tận dụng hiểu biết vừa có để cải thiện các đoạn code dễ lỗi.
4. Nhắm vào module dễ lỗi: Những module mà mọi người sợ sửa đổi thường là ứng viên tốt để refactor.
5. Nhắm vào module phức tạp cao: Cải thiện các module được đo là phức tạp nhất sẽ giúp tăng chất lượng chung nhiều nhất.
6. Luôn cải thiện phần bạn chính sửa: Code nào *không bao giờ thay đổi* thì không cần refactor. Khi sửa một phần cũ, hãy cải thiện luôn.
7. Định nghĩa interface giữa code xấu và code chuẩn: Đặt code cũ ở "thế giới lộn xộn", code mới ở "thế giới lý tưởng", và định nghĩa interface rõ ràng giữa chúng. Mỗi lần chạm vào code cũ, nâng cấp dần để chuyển code sang phía lý tưởng hơn.

---

### Tóm lại:

Refactoring là quá trình liên tục, có định hướng, an toàn và cần được quản lý kỹ lưỡng để đảm bảo duy trì và

nâng cao chất lượng phần mềm. Tập trung refactor có chiến lược, ưu tiên đoạn code hay bị lỗi, phức tạp hoặc vừa chỉnh sửa, luôn kết hợp với test, review và kiểm soát phiên bản để giảm tối đa rủi ro.

# Quy Trình Refactoring và Mối Liên Hệ với Hiệu Năng Chương Trình

## Refactoring và Quản Lý Rủi Ro

Quy trình **refactoring** (tái cấu trúc mã nguồn mà không làm thay đổi chức năng bên ngoài) có nhiều điểm tương đồng với quy trình sửa lỗi (defect fixing). Để tìm hiểu thêm về sửa lỗi, tham khảo Mục 23.3, “Fixing a Defect.” Những rủi ro liên quan đến refactoring cũng tương tự như khi điều chỉnh hiệu năng mã nguồn (**code tuning**). Tham khảo thêm về quản lý rủi ro trong code tuning tại Mục 25.6, “Summary of the Approach to Code Tuning.”

Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison Wesley, 1999. Đây là tài liệu hướng dẫn chi tiết và tiêu biểu về refactoring. Cuốn sách trình bày cụ thể nhiều kỹ thuật refactoring đã được tóm tắt trong chương này, đồng thời giới thiệu thêm các refactoring khác không nằm trong phạm vi chương. Fowler cung cấp rất nhiều ví dụ mã nguồn minh họa cách thức thực hiện từng bước refactoring.

## Các Điểm Chính (Key Points)

- Việc thay đổi chương trình là tất yếu**, diễn ra cả trong quá trình phát triển ban đầu lẫn sau khi phát hành.
- Phần mềm có thể được cải thiện hoặc suy giảm** mỗi lần thay đổi. Nguyên tắc tối thượng của tiến hóa phần mềm là: chất lượng nội tại (internal quality) cần phải được nâng cao cùng với tiến trình phát triển mã nguồn.
- Chìa khóa thành công của refactoring** là nhận biết các dấu hiệu cảnh báo (smells) cho thấy cần phải refactor.
- Học hỏi nhiều kỹ thuật refactoring cụ thể** cũng rất quan trọng để thành công.
- Chiến lược và cách tiếp cận an toàn khi refactoring** giúp tăng hiệu quả và hạn chế rủi ro.
- Giai đoạn phát triển là thời điểm tốt nhất để refactor**, giúp cải thiện chương trình và thực hiện các thay đổi mà ta mong muốn từ đầu.

## Chương 25: Chiến Lược Điều Chỉnh Hiệu Năng (Code-Tuning Strategies)

### Mục Lục

- 25.1 Tổng Quan về Hiệu Năng (Performance Overview)
- 25.2 Giới Thiệu về Điều Chỉnh Hiệu Năng Mã Nguồn (Introduction to Code Tuning)
- 25.3 Các Dạng Chậm Chạp và Cồng Kênh (Kinds of Fat and Molasses)
- 25.4 Đo lường (Measurement)
- 25.5 Lặp lại (Iteration)
- 25.6 Tổng Kết Cách Tiếp Cận Điều Chỉnh Mã Nguồn (Summary of the Approach to Code Tuning)

### Chủ đề liên quan

- Kỹ thuật code tuning: Chương 26
- Kiến trúc phần mềm (software architecture): Mục 3.5

## Tổng Quan về Hiệu Năng (25.1 Performance Overview)

Điều chỉnh mã nguồn nhằm nâng cao hiệu năng (performance tuning) từ lâu là chủ đề gây tranh cãi. Từ những năm 1960, tài nguyên máy tính bị hạn chế nghiêm trọng, do đó hiệu năng là mối quan tâm hàng đầu. Đến thập niên 1970, khi máy tính mạnh mẽ hơn, các lập trình viên nhận ra sự tập trung quá mức vào hiệu năng khiến mã nguồn khó đọc và duy trì—từ đó code tuning bớt được chú trọng.

Sự ra đời của máy vi tính cá nhân trong thập niên 1980 khiến vấn đề hiệu năng trở lại, đặc biệt trong bối cảnh phần mềm nhúng cho các thiết bị như điện thoại, PDA hoặc mã thông dịch (interpreted code) vào thập niên 2000. Lúc này, **hiệu năng lại thành chủ đề quan trọng**.

Bạn có thể xem xét hiệu năng ở hai cấp độ: **chiến lược** (strategic) và **chiến thuật** (tactical). Chương này tập trung vào yếu tố chiến lược: hiệu năng là gì, tầm quan trọng và cách tiếp cận tổng thể. Nếu bạn đã nắm vững các chiến lược, hãy chuyển sang Chương 26 để nghiên cứu kỹ thuật ở cấp mã nguồn.

**Lưu ý:** Trước khi tiến hành tối ưu, hãy đọc chương này để tránh lãng phí thời gian vào việc “tối ưu hóa” khi chưa cần thiết.

## Hiệu Năng và Tương Quan với Chất Lượng

Nhiều nhà phát triển nhìn thế giới qua “lăng kính mã nguồn”, luôn cho rằng mã càng tốt thì người dùng càng thích. Tuy nhiên, thực tế, người dùng quan tâm đến các đặc điểm phần mềm “hữu hình” hơn là chất lượng mã. Đa phần người dùng quan tâm đến lượng công việc chương trình xử lý được (throughput), giao diện người dùng trực quan, độ tin cậy cao, giảm thời gian chết hơn là tốc độ thô (raw performance) của mã nguồn bên dưới.

Điển hình: Việc chép 50 ảnh từ máy ảnh số về máy tính qua phần mềm mặc định rất mất thời gian. Dùng đầu đọc thẻ nhớ và Windows Explorer, công việc được đơn giản hóa nhiều lần về thao tác. Người dùng *không quan tâm thời gian truyền file nhanh hay chậm* mà chỉ quan tâm công việc hoàn thành nhanh, ít thao tác hơn.

**Kết luận:** Hiệu năng chỉ liên quan lỏng lẻo tới tốc độ code thực thi. Đôi khi tập trung vào tối ưu tốc độ mã nguồn lại làm giảm chất lượng tổng thể (overall quality) của sản phẩm.

## Cân nhắc Hiệu Năng từ Khi Thiết Kế (Thinking about Performance Early)

Khi đã xác định rằng hiệu năng là ưu tiên (không phân biệt là về tốc độ hay dung lượng bộ nhớ), cần cân nhắc ở nhiều cấp độ:

- Yêu cầu chương trình (Program requirements):** Có thực sự cần hiệu năng hay không? Phân tích thật kỹ để tránh đầu tư lãng phí. Một ví dụ nổi tiếng của Barry Boehm (2000b) cho thấy thay đổi yêu cầu thời gian phản hồi từ “dưới 1 giây” sang “4 giây đủ 90% thời gian” đã giúp giảm 70 triệu đô vào tổng chi phí hệ thống!
- Thiết kế chương trình (Program design):** Việc phân chia chương trình thành các module, class ảnh hưởng quan trọng đến khả năng đạt hiệu năng. Một ví dụ thực tế: việc chuyển từ sử dụng đa thức bậc 13 sang nhiều đa thức bậc 3, kết hợp phân cứng thích hợp, có thể nâng hiệu năng hơn là code tuning.
  - Đặt mục tiêu hiệu năng riêng cho từng module, class.
  - Thiết kế kiến trúc hướng hiệu năng, có khả năng thay thế module dễ dàng.
- Thiết kế lớp (Class) và hàm/method (routine):** Việc chọn kiểu dữ liệu (data type) và thuật toán (algorithm) ảnh hưởng trực tiếp đến chiếm dụng bộ nhớ và tốc độ xử lý.
  - Ví dụ: chọn **quicksort** thay vì **bubblesort**, tìm kiếm nhị phân (binary search) thay vì tuyến tính (linear search).
- Tương tác với hệ điều hành (OS):** Một số vấn đề về hiệu năng có thể do routine của hệ điều hành chậm (OS routines). Cần nhận diện các tác nhân này.
- Trình biên dịch (compiler):** Compiler hiện đại có thể tối ưu hóa mã tốt hơn chính bằng tay trong nhiều trường hợp.
- Phần cứng (hardware):** Nâng cấp phần cứng là giải pháp tối ưu cho phần mềm nội bộ nếu chi phí phù hợp.
- Điều chỉnh mã nguồn (code tuning):** Chỉ tiến hành thay đổi nhỏ ở phạm vi hàm, class hay vài dòng code, không phải thay đổi thiết kế tổng thể.

**Nhận xét:** Những thay đổi ở mỗi lớp (từ thiết kế hệ thống, thuật toán cho đến code tuning) có thể phối hợp với nhau để mang lại mức cải tiến hiệu năng rất lớn, nhưng trường hợp lý tưởng này hiếm khi xảy ra đầy đủ. Tuy nhiên, tiềm năng này vẫn là nguồn động lực tích cực.

## Giới Thiệu về Điều Chỉnh Mã Nguồn (25.2 Introduction to Code Tuning)

Điều chỉnh mã nguồn không phải là cách tối ưu hoặc dễ nhất để cải thiện hiệu năng:

- Cải tiến ở mức thiết kế chương trình, chọn thuật toán, nâng cấp phần cứng hoặc compiler thường hiệu quả và rẻ hơn.
- Việc tuning code đúng là tạo cảm giác “phá vỡ định luật tự nhiên” khi chỉ sửa vài dòng đã giúp routine chạy nhanh hơn 10 lần. Điều này hấp dẫn đối với lập trình viên có kinh nghiệm.
- Việc thành thạo tối ưu hóa code được xem là “nghĩ lễ trưởng thành” trong cộng đồng lập trình viên, giống như những phong cách nhỏ trong thể thao nhưng lại thể hiện bản lĩnh.

**Cảnh báo:** Mã nguồn hiệu quả không phải lúc nào cũng là mã “tốt”!

### Nguyên lý Pareto (Pareto Principle - 80/20 Rule)

Nguyên lý Pareto phát biểu rằng ta thu được 80% kết quả chỉ với 20% nỗ lực – và điều này hoàn toàn áp dụng cho tối ưu hóa chương trình. Cụ thể:



- 20% các routine/ngành nghiệp chiếm 80% thời gian thực thi (Boehm, 1987b).
- Donald Knuth phát hiện dưới 4% mã nguồn chịu trách nhiệm cho hơn 50% thời gian chạy của chương trình (Knuth, 1971).

**Bài học:** Cần đo đạc thực tế để xác định “điểm nóng” hiệu năng (hot spot), sau đó tập trung tối ưu bộ phận nhỏ sử dụng tài nguyên nhiều nhất.

- Một số trường hợp thực tế cho thấy code điều chỉnh nhỏ, đúng chỗ, giúp nâng hiệu năng toàn chương trình lên gấp đôi.

“Cái tốt nhất là kẻ thù của cái đủ tốt.” (The best is the enemy of the good) – đừng mãi mê tối ưu hóa mà không hoàn thành nhiệm vụ chính. Hãy hoàn thành, rồi sau đó hãy tối ưu phần nhỏ thực sự cần thiết.

---

## Những điều tưởng đúng về tối ưu hóa mã (Old Wives’ Tales)

Một số định kiến sai lầm thường gặp:

- *Giảm số dòng lệnh high-level sẽ tăng tốc chương trình:* Không đúng! Ví dụ, đoạn code khởi tạo mảng 10 phần tử dưới dạng for-loop có thể chậm hơn giải pháp gán từng giá trị một cách thủ công.

```
for i = 1 to 10
    a[ i ] = i
end for
```

so với

```
a[ 1 ] = 1
a[ 2 ] = 2
...
a[ 10 ] = 10
```

- Thử nghiệm trên Visual Basic và Java cho thấy cách viết thủ công nhanh hơn ít nhất 60%. Tham khảo bảng số liệu sau:

Ngôn ngữ	for-Loop (giây)	Gán trực tiếp (giây)	Tiết kiệm (%)	Tỉ lệ hiệu năng
Visual Basic	8.47	3.16	63%	2.5:1
Java	12.6	3.23	74%	4:1

- **Không có mối tương quan cố định nào giữa số dòng lệnh high-level với tốc độ/chỉ số sử dụng bộ nhớ của chương trình.** Một số thao tác high-level có thể chậm hơn so với thủ công, hãy luôn đo đạc.
- *Một số thao tác “có lẽ” nhanh hơn thao tác khác:* Không nên phỏng đoán hiệu năng! Các yếu tố như thay đổi ngôn ngữ, compiler, processor, thư viện hay memory, v.v. cũng làm thay đổi đặc tính hiệu năng. Điều quan trọng: **phải đo thực tế.**

Nếu bạn muốn chương trình có khả năng chạy đa nền tảng (portable), hãy hạn chế sử dụng tối ưu hóa thủ công vì nó có thể phản tác dụng khi thay đổi môi trường (compiler, phiên bản, phần cứng...).

---

## Kết luận

Hiệu năng phần mềm phụ thuộc vào rất nhiều yếu tố: yêu cầu thiết kế, lựa chọn thuật toán và cấu trúc dữ liệu, tương tác hệ điều hành, compiler, phần cứng và, cuối cùng mới đến tinh chỉnh ở cấp độ mã lệnh (code tuning). Khi cân nhắc tối ưu hóa, hãy luôn xác định rõ đâu là “điểm nóng” cần tối ưu, không chạy theo các định kiến mơ hồ về tốc độ hay hiệu quả. Các thay đổi nhỏ nhưng đúng chỗ, dựa trên đo đạc thực tế, mới là chiến lược đúng đắn để nâng cao hiệu năng mà không hy sinh các đặc tính phần mềm quan trọng khác.

## 25.2 Giới thiệu về Tinh chỉnh Mã nguồn (Code Tuning)

Khi bạn tiến hành tinh chỉnh mã nguồn, bạn đang **cam kết ngầm** rằng bạn sẽ phải **profiling (phân tích hiệu năng)** lại từng tối ưu hóa mỗi khi bạn thay đổi thương hiệu compiler (bộ biên dịch), phiên bản compiler, phiên bản thư viện, v.v. Nếu không phân tích lại, một tối ưu hóa nâng cao hiệu năng trên một phiên bản compiler hoặc thư viện có thể khiến hiệu năng giảm sút khi bạn thay đổi môi trường build.

### 1. Hiệu năng: Đừng quá tập trung vào tối ưu hóa sớm

“Phần lớn mọi tối ưu hóa sớm đều là nguồn gốc của mọi xấu xa.”  
— Donald Knuth

Một quan điểm cho rằng: nếu bạn cố gắng viết code nhanh nhất và ngắn nhất trong từng routine, thì chương trình của bạn sẽ nhanh. Thực tế, cách tiếp cận này dễ tạo ra tình huống "**không thấy rừng vì mãi nhìn cây**", khiến lập trình viên bỏ qua những tối ưu hóa toàn cục đáng kể vì bận "vi mô hóa" các đoạn nhỏ.

**Những vấn đề lớn với tối ưu hóa trong quá trình phát triển:**

- **Không thể xác định chính xác bottleneck (nút cổ chai) hiệu năng trước khi chương trình hoàn chỉnh.**
  - Lập trình viên thường không đoán đúng được 4% đoạn code tiêu tốn 50% thời gian thực thi, và vì vậy họ thường dành 96% thời gian tối ưu cho những phần không cần thiết.
- **Ngay cả khi xác định đúng bottleneck, lại quá tập trung vào chúng và bỏ qua những phần khác có thể trở thành bottleneck.**
  - Hậu quả, hiệu năng tổng thể lại bị giảm.
- **Tối ưu hóa trong giai đoạn đầu làm mất trọng tâm các mục tiêu phần mềm khác (tính đúng đắn, khả năng che giấu thông tin, khả năng đọc mã nguồn).**
  - Trong khi **performance** có thể cải thiện sau này, những yếu tố như **correctness (tính đúng đắn)** và **readability (tính dễ đọc)** rất khó sửa về sau.

Tóm lại, nhược điểm lớn nhất của tối ưu hóa sớm là thiếu quan điểm tổng thể, ảnh hưởng xấu đến chất lượng cuối cùng của phần mềm và trải nghiệm của người dùng.

Một chương trình phát triển với cấu trúc đơn giản rồi mới dành thời gian tối ưu từng điểm yếu hiệu năng chắc chắn sẽ nhanh hơn chương trình được tối ưu một cách vô tội vạ ngay từ đầu (Stevens 1981).

**Trong một số trường hợp, tối ưu hóa sau vẫn không đủ — vấn đề khi đó là kỹ thuật kiến trúc phần mềm (software architecture) chưa đạt yêu cầu, chứ không phải code chưa đủ tối ưu hóa vi mô.**

Nếu thực sự cần tối ưu sớm, cần giảm thiểu rủi ro bằng cách đặt ra các mục tiêu về size và speed cho tính năng, tối ưu hóa hướng tới mục tiêu này thay vì tối ưu vô định.

**2. Đúng-Thời-Điểm Tối Ưu Hóa (When to Tune)**

"Quy tắc tối ưu hóa của Jackson:

Quy tắc 1: Đừng làm.

Quy tắc 2 (chỉ dành cho chuyên gia): Đừng làm... chưa. Tức là, chưa làm cho tới khi có giải pháp chưa tối ưu hóa hoàn toàn 'trong tầm tay'."

— M.A. Jackson

Một ví dụ thực tế: trong một dự án phân tích dữ liệu tài chính bằng C++, sau khi hoàn thành chức năng xuất đồ thị đầu tiên, quá trình vẽ biểu đồ mất tận 45 phút. Cả nhóm tranh luận dữ dội về việc *viết lại toàn bộ bằng assembly*, tuy nhiên, thay vào đó chỉ cần xác định và xử lý vài *nút cổ chai*, thời gian thực thi giảm từ 45 phút xuống dưới 30 giây. Cuối cùng, chưa đến 1% số dòng code chiếm hơn 90% thời gian thực thi.

**3. Compiler Optimization (Tối ưu hóa bởi bộ biên dịch)**

Các compiler hiện đại tối ưu hóa mạnh mẽ hơn bạn tưởng. Thực tế, compiler đôi khi tối ưu các vòng lặp tốt hơn code được "tối ưu bằng tay". Việc tối ưu hóa code "thông minh" bằng cách sử dụng các thủ thuật phức tạp với chỉ số vòng lặp thực ra lại khiến compiler khó tối ưu hơn.

**Kết luận:** Hãy viết code rõ ràng, để compiler tự tối ưu khi có thể. Một số thử nghiệm cho thấy compiler có thể cải thiện tốc độ từ 40% trở lên so với bản không tối ưu.

**Ví dụ:**

Thời gian thực thi của routine sắp xếp chèn với và không có tối ưu hóa compiler:

Compiler	Không tối ưu hóa (giây)	Có tối ưu hóa (giây)	Tiết kiệm thời gian	Tỷ lệ tăng tốc
C++ compiler 1	2.21	1.05	52%	2:1
C++ compiler 2	2.78	1.15	59%	2.5:1
C++ compiler 3	2.43	1.25	49%	2:1
C# compiler	1.55	1.55	0%	1:1
Visual Basic	1.78	1.78	0%	1:1
Java VM 1	2.77	2.77	0%	1:1
Java VM 2	1.39	1.38	~0%	1:1

Mỗi compiler có điểm mạnh/điểm yếu riêng; nên tự kiểm tra hiệu quả trên chương trình của bạn.

## 4. Những nguồn gốc phổ biến của tính kém hiệu quả

### Các tác vụ Input/Output (I/O)

Input/Output không cần thiết là nguyên nhân lớn cho hiệu năng kém. Nếu có thể, sử dụng data structure trong RAM thay cho file trên disk, database hoặc qua mạng, trừ khi bị giới hạn bộ nhớ.

**So sánh tốc độ truy cập random phần tử giữa mảng trong bộ nhớ và file ngoài:**

Ngôn ngữ	File ngoài (giây)	Mảng trong bộ nhớ (giây)	Tỷ lệ tăng tốc
C++	6.04	0.000	1000:1
C#	12.8	0.010	1000:1

**Truy cập tuần tự (sequential access):**

Ngôn ngữ	File ngoài	Mảng bộ nhớ	Tỷ lệ tăng tốc
C++	3.29	0.021	150:1
C#	2.60	0.030	85:1

Nếu I/O diễn ra qua mạng, hiệu năng thường còn tệ hơn.

### Hiện tượng Paging

**Paging** (hoán chuyển trang bộ nhớ) làm chậm hệ thống mạnh. Chẳng hạn, cách khởi tạo mảng 2 chiều dưới đây sinh ra nhiều page fault:

```
for (column = 0; column < MAX_COLUMNS; column++) {
    for (row = 0; row < MAX_ROWS; row++) {
        table[row][column] = BlankTableElement();
    }
}
```

Khuyến nghị cách thay vòng lặp:

```
for (row = 0; row < MAX_ROWS; row++) {
    for (column = 0; column < MAX_COLUMNS; column++) {
        table[row][column] = BlankTableElement();
    }
}
```

Cách 2 chỉ tạo page fault khi đổi row, giúp cải thiện performance rõ rệt (có thể gấp 1000 lần).

### System calls (Gọi hàm hệ thống)

Các lệnh gọi hàm hệ thống tốn kém do phải context-switch. Nếu cơ chế này quá chậm, xem xét:

- Hiện thực chức năng cần thiết ở mức thấp hơn.
- Tránh gọi lên hệ thống khi không cần.
- Phân hồi cho nhà cung cấp hệ thống để được tối ưu.

**Ví dụ:** Một class `AppTime` khởi tạo thời gian bằng lệnh lấy giờ hệ thống, dù không cần thiết, gây hàng ngàn system call chỉ khi khởi tạo object. Thay bằng khởi tạo giá trị về 0 đã tăng hiệu năng tương đương các thay đổi code-tuning trước cộng lại.

### Interpreted languages (Ngôn ngữ thông dịch)

Ngôn ngữ thông dịch như Python, PHP có hiệu năng thua kém rất xa so với compiled language (ngôn ngữ biên dịch) như C++, C#, Visual Basic.

**Tham khảo: Thời gian thực thi tương đối so với C++**

Ngôn ngữ	Loại	Tỷ lệ tốc độ so với C++
C++	Biên dịch	1:1

Ngôn ngữ	Loại	Tỷ lệ tốc độ so với C++
Visual Basic	Biên dịch	1:1
C#	Biên dịch	1:1
Java	Byte code	1.5:1
PHP	Thông dịch	>100:1
Python	Thông dịch	>100:1

## Errors (Lỗi lập trình)

Các lỗi như quên tắt **debug-mode**, bỏ quên giải phóng bộ nhớ (memory deallocation), thiết kế bảng database sai, polling thiết bị không tồn tại... đều có thể tạo ra bottleneck nghiêm trọng.

**Ví dụ:** Chỉ cần thêm index vào bảng database đã giúp truy vấn tăng tốc gấp 30 lần.

## 5. Chi phí tương đối của các thao tác phổ biến

Bảng ước lượng chi phí thực thi của một số thao tác:

Loại thao tác	Ví dụ	C++	Java
Gán số nguyên	<code>i = j</code>	1	1
Gọi hàm không tham số	<code>foo()</code>	1	n/a
Gán số thực	<code>x = y</code>	1	1
Cộng số nguyên	<code>i = j + k</code>	1	1
Nhân số nguyên	<code>i = j * k</code>	1	1
Chia số nguyên	<code>i = j / k</code>	5	1.5
Lấy căn bậc 2	<code>x = sqrt(y)</code>	15	4
Sin	<code>x = sin(y)</code>	25	20
Log	<code>x = log(y)</code>	25	20
Mũ	<code>x = exp(y)</code>	50	20
Truy cập mảng	<code>i = a[j]</code>	1	1

*Lưu ý:* Các số liệu này chỉ tương đối, tùy thuộc vào môi trường máy, compiler, và không so sánh trực tiếp giữa C++ và Java.

## Tóm tắt

- **Đừng tối ưu hóa sớm** trừ khi thực sự cần thiết.
- **Profiling** chương trình để xác định bottleneck.
- Tập trung vào code rõ ràng, dễ đọc, modular — tối ưu hóa sau khi hệ thống đúng và hoàn thiện.
- Sử dụng các biện pháp tối ưu tổng thể thay vì lạc vào các tối ưu vi mô.
- Lựa chọn môi trường, compiler và ngôn ngữ phù hợp với mục đích và yêu cầu hiệu năng.

## Đọc thêm

Để tham khảo các câu chuyện thực tiễn về hiệu năng và tâm lý lập trình viên, xem:

- Gerald Weinberg, *Psychology of Computer Programming* (1998).

*Hết phần trích dịch.*

## Hầu hết các phép toán thường gặp đều có chi phí gần như nhau

Các phép gọi hàm (routine calls), phép gán (assignments), phép toán số nguyên (integer arithmetic) và phép toán số thực dấu phẩy động (floating-point arithmetic) đều có chi phí thực hiện xấp xỉ bằng nhau. Tuy nhiên, các hàm toán học siêu việt (transcendental math functions) lại cực kỳ tốn kém về mặt chi phí thực thi. Các phép gọi hàm đa hình (polymorphic routine calls) thì đắt hơn một chút so với các loại gọi hàm khác.

Bảng 25-2, hoặc một bảng tương tự bạn tự xây dựng, chính là chìa khóa để mở ra tất cả các cải tiến về tốc độ được mô tả trong Chương 26. Trong mọi trường hợp, việc cải thiện tốc độ đều bắt nguồn từ việc thay thế một phép toán đắt đỏ bằng một phép toán rẻ hơn. Chương 26 sẽ cung cấp các ví dụ về cách thực hiện điều này.

---

## 25.4 Đo lường (Measurement)

Thông thường, các phần nhỏ trong một chương trình lại tiêu tốn một tỷ lệ thời gian chạy không cân xứng. Do đó, hãy đo lường mã nguồn của bạn để xác định các “điểm nóng” (hot spots). Sau khi đã tìm ra và tối ưu hóa các điểm nóng này, hãy đo lại mã nguồn để đánh giá mức độ cải thiện.

Nhiều khía cạnh của hiệu năng có thể trái với直 giác. Trường hợp vừa nêu trong chương này, nơi 10 dòng mã lại chạy nhanh hơn và nhỏ hơn hẳn so với một dòng, là một ví dụ minh chứng cho sự bất ngờ mà mã nguồn có thể mang lại.

Kinh nghiệm cá nhân cũng không giúp ích nhiều cho việc tối ưu hóa. Trải nghiệm của một người có thể bắt nguồn từ máy cũ, ngôn ngữ cũ hoặc trình biên dịch cũ — khi bất kỳ thứ gì trong số này thay đổi, mọi dự đoán đều trở nên vô nghĩa. Bạn không bao giờ chắc chắn về tác động của một lần tối ưu hóa cho đến khi **đo lường** hiệu quả thực sự của nó.

**Lưu ý chính:** Bạn chỉ có thể chắc chắn về kết quả tối ưu hóa nếu có số liệu đo lường.

---

### Ví dụ minh họa

Nhiều năm trước, tôi từng viết một chương trình cộng tổng các phần tử trong một ma trận. Mã gốc như sau:

```
sum = 0;
for (row = 0; row < rowCount; row++) {
    for (column = 0; column < columnCount; column++) {
        sum = sum + matrix[row][column];
    }
}
```

Đoạn code này khá trực quan, nhưng hiệu năng của hàm cộng tổng này lại rất quan trọng, và tôi cho rằng việc truy cập mảng và kiểm tra điều kiện vòng lặp là tốn kém. Dựa vào những gì đã học trong các lớp khoa học máy tính, tôi biết rằng mỗi lần truy cập một mảng hai chiều sẽ thực hiện các phép nhân và cộng tốn kém. Với một ma trận 100x100, tổng cộng sẽ có 10.000 phép nhân và cộng, cộng thêm chi phí vòng lặp.

Bằng cách chuyển sang sử dụng con trỏ (pointer notation), tôi lý luận rằng có thể thay thế 10.000 phép nhân tốn kém bằng 10.000 phép tăng (increment) rẻ hơn. Tôi cẩn thận chuyển đổi code sang dạng sử dụng con trỏ:

```
sum = 0;
elementPointer = matrix;
lastElementPointer = matrix[rowCount - 1][columnCount - 1] + 1;
while (elementPointer < lastElementPointer) {
    sum = sum + *elementPointer++;
}
```

Mặc dù code không còn dễ đọc như trước, đặc biệt đối với người không thành thạo C++, tôi cảm thấy rất hài lòng với bản thân. Tôi tính rằng mình đã tiết kiệm 10.000 phép nhân và rất nhiều chi phí vòng lặp cho ma trận 100x100.

Tuy nhiên, khi đo lường tốc độ cải tiến, tôi hoàn toàn thất vọng: **không có sự cải thiện nào cả**, bất kể kích thước ma trận. Đào sâu vào mã máy, tôi phát hiện ra rằng trình tối ưu hóa của compiler đã tự động chuyển đổi việc truy cập mảng thành sử dụng con trỏ ngay từ đầu. Vì vậy, tác động duy nhất của tối ưu hóa thủ công là làm mã nguồn khó đọc hơn.

Nếu bạn không đo lường để biết code thực sự hiệu quả hơn, thì cũng không nên đánh đổi tính dễ hiểu của code vì một canh bạc hiệu năng.

---

### Độ chính xác của đo lường

Các phép đo hiệu năng (performance measurements) cần phải **chính xác**. Đo giờ bằng đồng hồ bấm giờ hoặc bằng cách đếm "một voi, hai voi, ba voi" là không chính xác. **Công cụ profiling** hoặc sử dụng đồng hồ hệ thống (system clock) với các routine ghi nhận thời gian thực thi là nên dùng.

Dù sử dụng công cụ của người khác hay tự viết mã đo thời gian, hãy đảm bảo rằng bạn chỉ đo **thời gian thực thi** của đoạn mã đang tối ưu. Nên dùng số lượng nhịp đồng hồ CPU (CPU clock ticks) dành cho chương trình của bạn thay vì thời gian thực (time of day), để tránh bị ảnh hưởng bởi tác vụ của các chương trình khác.

Đồng thời, hãy loại trừ thời gian đo và thời gian khởi tạo chương trình khỏi phép đo cuối cùng, để sự so sánh giữa mã gốc và mã tối ưu được công bằng.

---

## 25.5 Lặp lại (Iteration)

Một khi bạn đã xác định được “nút thắt” về hiệu năng (performance bottleneck), bạn sẽ kinh ngạc trước mức độ cải thiện mà tối ưu mã (code tuning) mang lại. Hiếm khi chỉ một kỹ thuật nào đó cho cải thiện gấp 10 lần, nhưng bạn có thể kết hợp các kỹ thuật; hãy tiếp tục thử nghiệm, ngay cả khi đã đạt được một cải tiến.

Ví dụ, tôi từng viết một chương trình mã hóa chuẩn DES (Data Encryption Standard). Thậm chí, tôi phải viết lại đến 30 lần. DES mã hóa dữ liệu số theo cách không thể giải mã nếu không có mật khẩu. Thuật toán mã hóa này cực kỳ phức tạp. Mục tiêu hiệu năng là mã hóa một file 18K trong 37 giây trên một máy tính IBM PC đời đầu. Bản cài đặt đầu tiên mất 21 phút 40 giây — một khoảng cách rất xa so với mục tiêu.

Dù mỗi tối ưu hóa riêng lẻ khá nhỏ, cộng dồn lại rất đáng kể. Nếu chỉ xét từng thay đổi lẻ, thì không bao giờ đạt được mục tiêu, nhưng tổng hợp chúng lại mới đạt hiệu quả cuối cùng.

Mỗi lần tối ưu hóa là một bước tiến, nhưng không phải mọi tối ưu hóa đều có hiệu quả — ít nhất 2/3 số tối ưu hóa tôi thử đều **không đem lại kết quả**, thậm chí có lúc thời gian chạy còn tăng lên gấp đôi.

### Bảng lịch sử tối ưu hóa:

Tối ưu hóa	Thời gian (phút:giây)	Cải thiện (%)
Cài đặt ban đầu - trực quan	21:40	-
Chuyển từ bit field sang mảng	7:30	65
Tháo unroll vòng lặp trong cùng	6:00	20
Loại bỏ bước hoán vị cuối	5:24	10
Kết hợp hai biến	5:06	5
Sử dụng một đồng nhất logic (logical identity)	4:30	12
Dùng chung vùng nhớ giảm giao động dữ liệu	3:36	20
Dùng chung vùng nhớ ở vòng ngoài	3:09	13
Unfold toàn bộ vòng lặp, dùng literal subscripts	1:36	49
Loại bỏ gọi routine, đặt code inline	0:45	53
Viết lại toàn bộ routine bằng assembly	0:22	51
<b>Cuối cùng</b>	<b>0:22</b>	<b>98</b>

## 25.6 Tóm tắt phương pháp tiếp cận tối ưu mã (Code Tuning Approach)

Bạn nên thực hiện các bước sau khi cân nhắc việc tối ưu mã để cải thiện hiệu năng chương trình:

1. Phát triển phần mềm với mã nguồn có cấu trúc tốt, dễ hiểu và dễ sửa đổi.
2. Nếu hiệu năng không đạt yêu cầu,
  - a. Lưu lại một phiên bản hoạt động tốt, để có thể quay lại trạng thái "ổn định lần cuối".
  - b. Đo lường hệ thống để xác định các điểm nóng (hot spots).
  - c. Xác định xem vấn đề hiệu năng có đến từ thiết kế, kiểu dữ liệu hoặc thuật toán không, và liệu tối ưu hóa mã có phải là phương án thích hợp. Nếu không, quay lại bước 1.
  - d. Tối ưu hóa nút thắt xác định ở bước (c).
  - e. Đo lường từng cải tiến một cách riêng biệt.
  - f. Nếu thay đổi không hiệu quả, quay lại mã đã lưu ở bước (a).
3. Lặp lại bước 2.

## Các nguồn tham khảo

- **Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software** (Smith & Williams, 2002) — Trình bày kỹ thuật xây dựng hiệu năng vào phần mềm ngay từ các giai đoạn phát triển sớm, cùng các ví dụ đa dạng.
- **Optimization: Your Worst Enemy** (Joseph M. Newcomer) — Nêu bật những cạm bẫy phổ biến khi tối ưu hóa không hiệu quả.
- **The Art of Computer Programming** (Donald Knuth) — Bộ sách kinh điển về thuật toán, chi tiết về lý thuyết và thực tiễn.

## Checklist: Chiến lược tối ưu mã (Code-Tuning Strategies)

### Hiệu năng chương trình tổng thể

- ☐ Đã cân nhắc thay đổi yêu cầu chương trình để tăng hiệu năng?
- ☐ Cân nhắc sửa đổi thiết kế tổng thể?
- ☐ Cân nhắc thiết kế lại class?
- ☐ Hạn chế tương tác với hệ điều hành?
- ☐ Hạn chế thao tác I/O?
- ☐ Đã thử dùng ngôn ngữ dịch thay vì ngôn ngữ thông dịch?
- ☐ Tận dụng tối ưu hóa của compiler?
- ☐ Cân nhắc chuyển sang phần cứng khác?
- ☐ Chỉ tối ưu hóa mã khi không còn phương án nào khác?

### Phương pháp tối ưu hóa mã

- ☐ Đảm bảo chương trình đã hoàn toàn đúng trước khi tối ưu?
- ☐ Đo lường điểm nghẽn trước khi tối ưu?
- ☐ Đo lường tác động của từng thay đổi?
- ☐ Hoàn nguyên nếu thay đổi không đem lại hiệu quả mong muốn?
- ☐ Đã thử nhiều phương án cho mỗi nút thắt, lặp lại quy trình tối ưu?

## Các điểm mấu chốt (Key Points)

- Hiệu năng chỉ là một khía cạnh của chất lượng phần mềm**, thường không phải là quan trọng nhất. Thiết kế kiến trúc, giải pháp và thuật toán thường ảnh hưởng lớn hơn nhiều đến tốc độ thực thi so với tối ưu mã.
- Số liệu đo lường định lượng** là chìa khóa tối ưu hóa hiệu năng; và cần lặp lại qua từng cải tiến.
- Đa số chương trình dành phần lớn thời gian ở một phần mã rất nhỏ — **hãy xác định chính xác nó qua đo lường**.
- Lặp lại nhiều lần thường cần thiết** để đạt hiệu năng mong muốn.
- Cách tốt nhất chuẩn bị cho công việc tối ưu hóa** là viết mã sạch, dễ hiểu, dễ sửa đổi ngay từ đầu.

## Chương 26: Kỹ thuật tối ưu mã (Code-Tuning Techniques)

- 26.1 Logic**: trang 610
- 26.2 Vòng lặp (Loops)**: trang 616
- 26.3 Biến đổi dữ liệu (Data Transformations)**: trang 624
- 26.4 Biểu thức (Expressions)**: trang 630
- 26.5 Routine**: trang 639
- 26.6 Viết lại ở ngôn ngữ cấp thấp**: trang 640
- 26.7 Bản chất không đổi**: trang 643

Tối ưu mã (code tuning) là một đề tài phổ biến trong lịch sử lập trình máy tính. Một khi bạn quyết định cần cải thiện hiệu năng ở mức độ mã (code level), bạn có rất nhiều kỹ thuật khác nhau. Chương này tập trung vào các kỹ thuật tăng tốc mã, đồng thời đề cập một số mẹo giúp mã nhỏ hơn.

Lưu ý: Việc giảm kích thước (size) thường đến từ việc thiết kế lại class và dữ liệu, không chỉ nhờ tối ưu mã.

Hầu hết các kỹ thuật mô tả ở đây không phải là các công thức có thể cắt dán luôn vào chương trình của bạn. Mục đích là minh họa một vài ý tưởng tối ưu để bạn thích nghi cho tình huống của mình.

Tuy các thay đổi tối ưu mã giống như refactoring (tái cấu trúc, xem lại ở Chương 24), nhưng thực chất đây là “anti-refactorings” — là thay đổi làm **giảm** tính cấu trúc nội bộ để đổi lấy hiệu năng.

Nếu việc thay đổi không làm suy giảm cấu trúc mã, chúng ta đã mặc định dùng rồi, gọi là thực hành lập trình chuẩn.

### 26.1 Logic

Phần lớn lập trình xoay quanh thao tác với logic. Phần này đề cập cách tối ưu hóa các biểu thức logic.

## Dừng kiểm tra khi đã biết đáp án

Giả sử bạn có một câu điều kiện như:

```
if (5 < x) and (x < 10) then
```

Một số ngôn ngữ hỗ trợ **short-circuit evaluation** (đánh giá ngắn mạch), nghĩa là dừng đánh giá khi đã có kết quả (như C++ và Java). Nếu ngôn ngữ bạn dùng không hỗ trợ, bạn cần tự điều chỉnh logic, chẳng hạn:

```
if (5 < x) then
    if (x < 10) then
        ...
```

Nguyên tắc dừng kiểm tra khi đã biết đáp án áp dụng cho nhiều trường hợp, ví dụ: tìm một giá trị âm trong mảng — chỉ cần phát hiện là dừng vòng lặp.

Ví dụ code chưa tối ưu:

```
negativeInputFound = false;
for (i = 0; i < count; i++) {
    if (input[i] < 0) {
        negativeInputFound = true;
    }
}
```

Code tốt hơn là dừng lại khi đã tìm thấy:

- Thêm `break` sau khi gán `negativeInputFound = true;`
- Nếu không có `break`, dùng `goto` để nhảy ra ngoài vòng lặp
- Đổi `for` sang `while` và kiểm tra cả biến cờ lẫn biến đếm
- Dùng sentinel (giá trị đặc biệt), kiểm tra giá trị ở điều kiện vòng lặp

## Kết quả thực tế:

Ngôn ngữ Thời gian gốc Thời gian tối ưu Tiết kiệm thời gian

C++	4.27	3.68	14%
Java	4.85	3.46	29%

*Thời gian trong bảng chỉ mang tính so sánh tương đối trong từng dòng bảng; kết quả thực tế tùy vào compiler và môi trường.*

Hiệu quả phụ thuộc vào số lượng giá trị và xác suất tìm thấy giá trị, thử nghiệm này giả định trung bình 100 phần tử, tỉ lệ tìm thấy là 50%.

---

Tiếp theo sẽ trình bày các kỹ thuật tối ưu logic, vòng lặp,... theo hướng dẫn trên.

# Ưu Tiên Trường Hợp Thông Thường, Tối Ưu Phép Lặp và Kiểm Tra Hiệu Năng Cấu Trúc Logic

## 1. Sắp xếp Xử lý Các Trường hợp Logic

Nguyên tắc lập trình hiệu quả là trường hợp phổ biến nhất nên dễ dàng được kiểm tra và xử lý ngay lập tức; nếu tồn tại sự không tối ưu, chúng nên xuất hiện ở các trường hợp ít gặp. Nguyên tắc này áp dụng cho cả cấu trúc `case` (ví dụ: Select-Case trong Visual Basic) và xâu chuỗi các câu lệnh `if-then-else`.

Ví dụ sau minh họa một đoạn code chưa được tối ưu trong xử lý ký tự nhập liệu của trình xử lý văn bản:

### Visual Basic: Ví dụ về Kiểm tra Logic Kém Hiệu Quả

```
Select inputCharacter
Case "+", "="
    ProcessMathSymbol( inputCharacter )
Case "0" To "9"
    ProcessDigit( inputCharacter )
Case ",", ".", ":", ";", "!", "?"
```



```

        ProcessPunctuation( inputCharacter )
Case " "
    ProcessSpace( inputCharacter )
Case "A" To "Z", "a" To "z"
    ProcessAlpha( inputCharacter )
Case Else
    ProcessError( inputCharacter )
End Select

```

Trong ví dụ này, thứ tự `case` gần với thứ tự sắp xếp ASCII. Hậu quả là chương trình phải kiểm tra qua nhiều trường hợp ít khả năng xảy ra trước khi gặp trường hợp phổ biến nhất (chữ cái), làm giảm hiệu năng.

Nếu bạn biết tần suất xuất hiện của các ký tự, hãy đặt các trường hợp phổ biến lên trước:

## Visual Basic: Ví dụ về Kiểm tra Logic Hiệu Quả

```

Select inputCharacter
Case "A" To "Z", "a" To "z"
    ProcessAlpha( inputCharacter )
Case " "
    ProcessSpace( inputCharacter )
Case ",", ".", ":", ";", "!", "?"
    ProcessPunctuation( inputCharacter )
Case "0" To "9"
    ProcessDigit( inputCharacter )
Case "+", "="
    ProcessMathSymbol( inputCharacter )
Case Else
    ProcessError( inputCharacter )
End Select

```

Do các trường hợp phổ biến được xử lý trước, số lần kiểm tra trung bình giảm xuống và hiệu suất tăng lên.

**Kết quả đánh giá hiệu năng với tỉ lệ nhập liệu: 78% ký tự chữ, 17% khoảng trắng, 5% dấu câu:**

Ngôn ngữ	Thời gian bình thường	Thời gian sau tối ưu	Tiết kiệm thời gian
C#	0.220	0.260	-18%
Java	2.56	2.56	0%
Visual Basic	0.280	0.260	7%

**Lưu ý:** Cách cấu trúc switch-case trong C# và Java (phải liệt kê từng giá trị riêng lẻ thay vì theo dải như Visual Basic) làm hạn chế khả năng tối ưu này. Điều này nhấn mạnh rằng không nên áp dụng tùy tiện mọi mẹo tối ưu hóa — kết quả còn tùy vào cách trình biên dịch (compiler) triển khai.

Sơ sánh code sử dụng cấu trúc if-then-else tương tự:

Ngôn ngữ	Thời gian bình thường	Thời gian sau tối ưu	Tiết kiệm thời gian
C#	0.630	0.330	48%
Java	0.922	0.460	50%
Visual Basic	1.36	1.00	26%

Rõ ràng, hiệu quả giữa các cấu trúc và trình biên dịch rất khác nhau. Kết luận quan trọng là: **Luôn đo lường để biết chính xác tác động của từng phương pháp tối ưu, không nên áp dụng máy móc.**

## 2. So sánh Logic Cấu trúc Tương tự

Sơ sánh hiệu suất giữa `case` và `if-then-else` với cùng một khối logic cho ra các kết quả rất khác biệt giữa các ngôn ngữ:

Ngôn ngữ	case	if-then-else	Tiết kiệm thời gian	Tỉ lệ performance
C#	0.260	0.330	-27%	1:1
Java	2.56	0.460	82%	6:1
Visual Basic	0.260	1.00	258%	1:4

Đây là ví dụ về việc quy tắc “thumb rule” trong tối ưu hóa là không đáng tin cậy — thử nghiệm và đo lường thực tế là cách duy nhất.

## 3. Thay thế Biểu thức Phức tạp bằng Tra cứu Bảng (table lookup)

Trong nhiều trường hợp, **tra cứu bảng** (table lookup) sẽ nhanh hơn chuỗi điều kiện logic phức tạp. Giả sử ta muốn phân loại thành nhóm A, B, hoặc C:

```
if ( ( a && !c ) || ( a && b && c ) ) {
    category = 1;
}
else if ( ( b && !a ) || ( a && c && !b ) ) {
    category = 2;
}
else if ( c && !a && !b ) {
    category = 3;
}
else {
    category = 0;
}
```

Có thể thay thế bằng bảng tra cứu:

```
// Định nghĩa bảng tra cứu categoryTable
static int categoryTable[ 2 ][ 2 ][ 2 ] = {
    // !b!c, !bc, b!c, bc
    { 0, 3, 2, 2 }, // !a
    { 1, 2, 1, 1 }  // a
};
category = categoryTable[ a ][ b ][ c ];
```

**Chú ý:** Dù bảng tra cứu ban đầu khó hiểu, nếu được chú thích tốt thì bảo trì về sau sẽ dễ hơn, và hiệu suất cải thiện đáng kể.

Ngôn ngữ	Thời gian bình thường	Thời gian sau tối ưu	Tiết kiệm thời gian	Tỉ lệ
C++	5.04	3.39	33%	1.5:1
Visual Basic 5.21		2.60	50%	2:1

## 4. Sử dụng Lazy Evaluation (Đánh giá lười biếng)

*Lazy evaluation* (đánh giá lười biếng) là kỹ thuật trì hoãn việc thực hiện cho đến khi thực sự cần thiết. Ví dụ, khi chương trình chứa một bảng gồm 5000 giá trị nhưng chỉ dùng vài phần trăm, hãy tính toán từng giá trị khi cần thiết thay vì “đổ” toàn bộ ngay từ đầu. Khi đã tính xong một phần tử, có thể lưu lại cho lần dùng sau (tức là *caching*).

---

## 26.2 Các Kỹ thuật Tối Ưu Vòng Lặp (Loop)

Loops thường là “điểm nóng” (hot spot) vì được thực thi nhiều lần trong chương trình. Sau đây là các kỹ thuật tăng tốc độ thực thi vòng lặp.

### Unswitching (Tách ngoại điều kiện)

Thông thường, việc so sánh điều kiện trong mỗi vòng lặp là dư thừa nếu giá trị điều kiện không thay đổi. Có thể tách điều kiện ra ngoài như sau:

#### Ví dụ C++: Vòng lặp chưa tách điều kiện

```
for ( i = 0; i < count; i++ ) {
    if ( sumType == SUMTYPE_NET ) {
        netSum = netSum + amount[ i ];
    } else {
        grossSum = grossSum + amount[ i ];
    }
}
```

#### Sau khi tách điều kiện (unswitching):

```
if ( sumType == SUMTYPE_NET ) {
    for ( i = 0; i < count; i++ ) {
        netSum = netSum + amount[ i ];
    }
} else {
    for ( i = 0; i < count; i++ ) {
        grossSum = grossSum + amount[ i ];
    }
}
```

```
}  
}
```

**Lưu ý:** Giải pháp này vi phạm nguyên tắc “DRY” (Don't Repeat Yourself) và giảm khả năng bảo trì. Tuy nhiên, trong tối ưu hóa hiệu suất, đây là sự cân nhắc hợp lý.

## Kết quả:

Ngôn ngữ	Thời gian thường	Thời gian tối ưu	Tiết kiệm
C++	2.81	2.27	19%
Java	3.97	3.12	21%
Visual Basic	2.78	2.77	<1%
Python	8.14	5.87	28%

**Cảnh báo:** Nếu sửa đổi chỉ số vòng lặp (counter), phải thay đổi ở cả hai nơi — gây khó khăn trong bảo trì và tăng khả năng lỗi.

## Jamming (Hợp nhất vòng lặp)

Jamming hay *fusion* là kỹ thuật hợp nhất hai hoặc nhiều vòng lặp có phạm vi lặp giống nhau thành một vòng lặp duy nhất để giảm overhead.

### Visual Basic: Tách hai vòng lặp riêng biệt

```
For i = 0 to employeeCount - 1  
    employeeName( i ) = ""  
Next  
For i = 0 to employeeCount - 1  
    employeeEarnings( i ) = 0  
Next
```

### Hợp nhất (jammed):

```
For i = 0 to employeeCount - 1  
    employeeName( i ) = ""  
    employeeEarnings( i ) = 0  
Next
```

### Hiệu quả đạt được:

Ngôn ngữ	Thời gian thường	Thời gian tối ưu	Tiết kiệm
C++	3.68	2.65	28%
PHP	3.97	2.42	32%
Visual Basic	3.75	3.56	4%

## Unrolling (Gỡ cuộn vòng lặp)

*Loop unrolling* là kỹ thuật làm giảm overhead vòng lặp bằng cách xử lý nhiều phần tử trong mỗi vòng lặp.

### Java: Vòng lặp thông thường

```
i = 0;  
while ( i < count ) {  
    a[ i ] = i;  
    i = i + 1;  
}
```

### Unrolling một lần (xử lý hai phần tử mỗi vòng):

```
i = 0;  
while ( i < count - 1 ) {  
    a[ i ] = i;  
    a[ i + 1 ] = i + 1;  
    i = i + 2;  
}  
if ( i == count ) {
```

```
    a[ count - 1 ] = count - 1;
}
```

### Unrolling hai lần (xử lý ba phần tử mỗi vòng):

```
i = 0;
while ( i < count - 2 ) {
    a[ i ] = i;
    a[ i + 1 ] = i+1;
    a[ i + 2 ] = i+2;
    i = i + 3;
}
if ( i <= count - 1 ) {
    a[ count - 1 ] = count - 1;
}
if ( i == count - 2 ) {
    a[ count - 2 ] = count - 2;
}
```

### Kết quả kiểm thử:

#### Ngôn ngữ Gỡ cuộn 1 lần Gỡ cuộn 2 lần Thường

C++	1.15	1.01	1.75
Java	0.581	0.581	1.01
PHP	4.49	3.70	5.33
Python	3.21 (-27%)	2.79 (-12%)	2.51

*Lưu ý:* Unrolling hữu ích khi số phần tử nhỏ, hoặc chuỗi xử lý theo nhóm đơn giản. Khi số phần tử lớn hoặc không xác định trước, dùng kỹ thuật này vẫn phải đặc biệt cẩn thận với lỗi “off-by-one”.

## Tối thiểu hóa công việc bên trong vòng lặp

Hãy thực hiện các phép tính ngoài vòng lặp nếu có thể, thay vì lặp lại nhiều lần trong mỗi vòng lặp.

### C++: Câu lệnh phức tạp trong vòng lặp

```
for ( i = 0; i < rateCount; i++ ) {
    netRate[ i ] = baseRate[ i ] * rates->discounts->factors->net;
}
```

### Tối ưu bằng biến đệm:

```
quantityDiscount = rates->discounts->factors->net;
for ( i = 0; i < rateCount; i++ ) {
    netRate[ i ] = baseRate[ i ] * quantityDiscount;
}
```

Hiệu quả:

#### Ngôn ngữ Thường Đã tối ưu Tiết kiệm

C++	3.69	2.97	19%
C#	2.27	1.97	13%
Java	4.13	2.35	43%

## Giá trị Sentinel trong Vòng lặp (Sentinel Value)

Khi vòng lặp kiểm tra điều kiện kết thúc phức tạp, hãy cân nhắc sử dụng **sentinel value** (giá trị chốt), một giá trị đặc biệt đánh dấu kết thúc để đơn giản hóa điều kiện:

### Ví dụ tìm kiếm tuyến tính với điều kiện kết hợp:

```
found = FALSE;
i = 0;
while ( ( !found ) && ( i < count ) ) {
    if ( item[ i ] == testValue ) {
        found = TRUE;
    } else {
```

```
        i++;
    }
}
if ( found ) { ... }
```

## Tối ưu bằng Sentinel value:

```
initialValue = item[ count ];
item[ count ] = testValue;
i = 0;
while ( item[ i ] != testValue ) {
    i++;
}
if ( i < count ) { ... }
```

Hiệu quả:

Loại mảng	Ngôn ngữ	Thường	Tối ưu	Tiết kiệm
Số nguyên	C#	0.771	0.590	23%
	Java	1.63	0.912	44%
	Visual Basic	1.34	0.470	65%
Số thực 4 byte	C#	1.351	1.021	24%
	Java	1.923	1.282	33%
	Visual Basic	1.752	1.011	42%

**Chú ý:** Phải chọn sentinel value phù hợp để không làm sai dữ liệu và phải đảm bảo cấp đủ không gian bộ nhớ để lưu giá trị này.

## Kết luận chung

- Không có quy tắc tối ưu nào đúng cho mọi trường hợp.
- Cần đo lường thực tế từng kỹ thuật trên chính môi trường, ngôn ngữ và trình biên dịch cụ thể bạn đang sử dụng.
- Mỗi kỹ thuật đều có *trade-off* (đánh đổi) giữa hiệu suất, khả năng đọc/mở rộng và bảo trì.
- Giá trị lớn nhất của bài học tối ưu hóa là sự linh hoạt và phải sử dụng *các chỉ số định lượng thực tế* làm cơ sở quyết định.

## Kỹ thuật cải thiện hiệu năng: Dịch thuật học thuật

### 1. Tối ưu hóa vòng lặp

Mỗi lần vòng lặp được thực thi, nó phải khởi tạo chỉ số vòng lặp (*loop index*), tăng chỉ số này sau mỗi lần lặp và kiểm tra chỉ số sau mỗi lần lặp. Tổng số lượt thực thi vòng lặp là 100 đối với vòng lặp ngoài và  $100 * 5 = 500$  đối với vòng lặp trong, tổng cộng 600 lần lặp. Chỉ bằng cách hoán đổi vị trí giữa vòng lặp trong và vòng lặp ngoài, bạn có thể thay đổi tổng số lần lặp thành 5 đối với vòng lặp ngoài và  $5 * 100 = 500$  đối với vòng lặp trong, tổng cộng 505 lần lặp.

Một cách phân tích, bạn sẽ kỳ vọng tiết kiệm được khoảng  $(600 - 505) / 600 = 16\%$  khi hoán đổi hai vòng lặp. Dưới đây là sự khác biệt về hiệu năng được đo đạc:

Ngôn ngữ	Thời gian ban đầu	Thời gian tối ưu	Tiết kiệm (%)
C++	4.75	3.19	33%
Java	5.39	3.56	34%
PHP	4.16	3.65	12%
Python	3.48	3.33	4%

Kết quả biến động đáng kể, một lần nữa cho thấy bạn phải **đo lường hiệu ứng tối ưu hóa trong môi trường cụ thể** trước khi áp dụng rộng rãi.

### 2. Giảm độ mạnh của phép toán (Strength Reduction)

Giảm độ mạnh của phép toán nghĩa là thay thế phép toán tốn kém như nhân (multiplication) bằng phép toán rẻ hơn như cộng (addition). Đôi khi, bạn sẽ gặp biểu thức bên trong vòng lặp phụ thuộc vào phép nhân giữa chỉ số vòng lặp và một hệ số.

Thông thường, cộng nhanh hơn nhân và nếu bạn có thể tính toán cùng một giá trị bằng phép cộng thay cho phép nhân trong mỗi lần lặp, đoạn mã sẽ thực thi nhanh hơn.

#### Ví dụ mã Visual Basic sử dụng phép nhân:

```
For i = 0 to saleCount - 1
    commission(i) = (i + 1) * revenue * baseCommission * discount
Next
```

Mã này đơn giản nhưng tốn kém về hiệu năng. Bạn có thể viết lại vòng lặp để tích lũy bội số thay vì tính toán mỗi lần, như sau:

#### Ví dụ mã Visual Basic sử dụng phép cộng thay vì nhân:

```
incrementalCommission = revenue * baseCommission * discount
cumulativeCommission = incrementalCommission
For i = 0 to saleCount - 1
    commission(i) = cumulativeCommission
    cumulativeCommission = cumulativeCommission + incrementalCommission
Next
```

Việc thay đổi này giống như có “phiếu giảm giá” cho chi phí chạy vòng lặp: thay vì nhân mỗi lần, bạn chỉ cộng thêm giá trị. Dưới đây là hiệu quả tiết kiệm thời gian:

Ngôn ngữ	Thời gian gốc	Thời gian tối ưu	Tiết kiệm (%)
C++	4.33	3.80	12%
Visual Basic	3.54	1.80	49%

**Lưu ý:** Các biến đều có kiểu số thực dấu phẩy động (floating point), và saleCount = 20 trong phép đo.

## 3. Biến đổi kiểu dữ liệu (Data Transformations)

### a. Sử dụng số nguyên thay cho số thực dấu phẩy động

Phép cộng và nhân với kiểu số nguyên (integer) thường nhanh hơn so với số thực dấu phẩy động (floating-point). Thay đổi chỉ số vòng lặp từ số thực sang số nguyên có thể tiết kiệm thời gian:

#### Ví dụ sử dụng số thực trong vòng lặp Visual Basic:

```
Dim x As Single
For x = 0 to 99
    a(x) = 0
Next
```

#### Ví dụ sử dụng số nguyên tối ưu hơn:

```
Dim i As Integer
For i = 0 to 99
    a(i) = 0
Next
```

#### Hiệu quả:

Ngôn ngữ	Thời gian gốc	Thời gian tối ưu	Tiết kiệm (%)	Tỷ lệ hiệu năng
C++	2.80	0.801	71%	3,5:1
PHP	5.01	4.65	7%	1:1
Visual Basic	6.84	0.280	96%	25:1

### b. Hạn chế số chiều của mảng

Nhiều chiều của mảng (array) thường đắt đỏ về chi phí truy cập. Nếu có thể, hãy thiết kế dữ liệu dạng mảng một chiều để tiết kiệm thời gian.

#### Ví dụ Java khởi tạo mảng hai chiều truyền thống:

```
for (row = 0; row < numRows; row++) {
    for (column = 0; column < numColumns; column++) {
        matrix[row][column] = 0;
    }
}
```

**Phiên bản dùng mảng một chiều:**

```
for (entry = 0; entry < numRows * numColumns; entry++) {
    matrix[entry] = 0;
}
```

**Tóm tắt hiệu quả tối ưu hóa:**

Ngôn ngữ	Thời gian gốc	Tối ưu	Tiết kiệm (%)	Tỷ lệ hiệu năng
C++	8.75	7.82	11%	1:1
C#	3.28	2.99	9%	1:1
Java	7.78	4.14	47%	2:1
PHP	6.24	4.10	34%	1,5:1
Python	3.31	2.23	32%	1,5:1
Visual Basic	9.43	3.22	66%	3:1

Nhận xét: Việc tối ưu này mang lại hiệu quả tốt ở Visual Basic và Java, vừa phải ở PHP, Python và trung bình với C++, C#. Đừng áp dụng máy móc mà cần kiểm thử thực tế từng trường hợp cụ thể.

**c. Giảm số lần tham chiếu mảng**

Ngoài việc giảm chiều cho mảng, tối thiểu số lần truy cập vào mảng cũng tăng hiệu năng. Nếu phần tử mảng được sử dụng nhiều lần trong một vòng lặp lồng, hãy gán nó ra biến tạm bên ngoài vòng lặp trong cùng.

**Ví dụ C++ truy cập mảng không cần thiết:**

```
for (discountType = 0; discountType < typeCount; discountType++) {
    for (discountLevel = 0; discountLevel < levelCount; discountLevel++) {
        rate[discountLevel] = rate[discountLevel] * discount[discountType];
    }
}
```

**Phiên bản tối ưu hóa:**

```
for (discountType = 0; discountType < typeCount; discountType++) {
    thisDiscount = discount[discountType];
    for (discountLevel = 0; discountLevel < levelCount; discountLevel++) {
        rate[discountLevel] = rate[discountLevel] * thisDiscount;
    }
}
```

**Kết quả:**

Ngôn ngữ	Thời gian gốc	Thời gian tối ưu	Tiết kiệm (%)
C++	32.1	34.5	-7%
C#	18.3	17.0	7%
Visual Basic	23.2	18.4	20%

Lưu ý hiệu quả khác nhau theo ngôn ngữ/biến dịch.

**d. Sử dụng chỉ mục bổ trợ (Supplementary Indexes)**

Chỉ mục bổ trợ là dữ liệu liên quan giúp truy cập một kiểu dữ liệu hiệu quả hơn (ví dụ: thêm trường lưu độ dài thực tế vào chuỗi - string - như Visual Basic).

- **Ví dụ: biến thêm độ dài của chuỗi** giúp kiểm tra độ dài nhanh hơn so với việc phải duyệt từng ký tự đến dấu kết thúc (như trong C).
- **Chỉ mục song song (parallel index):** Khi phần tử dữ liệu lớn, hãy tạo cấu trúc phụ lưu trữ giá trị khóa (key) và con trỏ. Lúc này, việc tìm kiếm/sắp xếp sẽ diễn ra trên cấu trúc chỉ mục, truy xuất đến dữ liệu chính chỉ khi cần.

**e. Sử dụng bộ nhớ đệm (caching)**

Bộ nhớ đệm (cache) là kỹ thuật lưu trữ tạm thời một số giá trị được truy cập thường xuyên để truy xuất nhanh chóng hơn. Ví dụ điển hình là lưu trữ bản ghi (record) hay kết quả tính toán tốn thời gian.

**Ví dụ Java tính cạnh huyền tam giác vuông kết hợp cache:**

### Không sử dụng cache:

```
double Hypotenuse(
    double sideA,
    double sideB
) {
    return Math.sqrt((sideA * sideA) + (sideB * sideB));
}
```

### Sử dụng cache:

```
private double cachedHypotenuse = 0;
private double cachedSideA = 0;
private double cachedSideB = 0;
public double Hypotenuse(
    double sideA,
    double sideB
) {
    if ((sideA == cachedSideA) && (sideB == cachedSideB)) {
        return cachedHypotenuse;
    }
    cachedHypotenuse = Math.sqrt((sideA * sideA) + (sideB * sideB));
    cachedSideA = sideA;
    cachedSideB = sideB;
    return cachedHypotenuse;
}
```

### Hiệu quả tiết kiệm:

Ngôn ngữ	Thời gian gốc	Thời gian tối ưu	Tiết kiệm (%)	Tỷ lệ hiệu năng
C++	4.06	1.05	74%	4:1
Java	2.54	1.40	45%	2:1
Python	8.16	4.17	49%	2:1
Visual Basic	24.0	12.9	47%	2:1

**Thành công của cache phụ thuộc** vào chi phí truy cập phần tử đã lưu, so với tạo phần tử mới, và tần suất truy vấn dữ liệu cũ.

## 4. Tối ưu hóa biểu thức (Expressions)

Phần lớn công việc trong chương trình diễn ra trong các biểu thức toán học hoặc logic. Biểu thức phức tạp thường tốn kém về chi phí tính toán.

### a. Khai thác đồng nhất đại số (algebraic identities)

Dùng các đồng nhất thức để thay thế phép toán đắt giá bằng phép toán rẻ hơn. Ví dụ, biểu thức sau tương đương về logic:

- not a and not b
- not (a or b)

Khi chọn biểu thức thứ hai, bạn bỏ qua một phép toán not.

**Ví dụ tối ưu kiểm tra căn bậc hai:** Thay vì so sánh  $\sqrt{x} < \sqrt{y}$ , hãy so sánh trực tiếp  $x < y$ . Kết quả tiết kiệm thời gian rất ấn tượng:

Ngôn ngữ	Thời gian gốc	Thời gian tối ưu	Tiết kiệm (%)	Tỷ lệ hiệu năng
C++	7.43	0.010	99.9%	750:1
Visual Basic	4.59	0.220	95%	20:1
Python	4.21	0.401	90%	10:1

### b. Áp dụng giảm độ mạnh phép toán (strength reduction)

- Thay phép nhân bằng phép cộng.
- Thay phép lũy thừa bằng phép nhân.
- Thay công thức lượng giác (trigonometric identities).
- Dùng số nguyên có độ dài phù hợp (long, int).
- Thay số dấu phẩy động (floating-point) bằng số nguyên.
- Dùng phép dịch bit thay cho phép nhân/div chia cho 2.



### Ví dụ tính đa thức (polynomial):

```
' Cách thường gặp:
value = coefficient(0)
For power = 1 To order
    value = value + coefficient(power) * x^power
Next

' Phương pháp tối ưu hơn:
value = coefficient(0)
powerOfX = x
For power = 1 to order
    value = value + coefficient(power) * powerOfX
    powerOfX = powerOfX * x
Next
```

### Hiệu quả:

Ngôn ngữ	Thời gian gốc	Tối ưu đầu	Tối ưu hơn nữa	Tiết kiệm so với đầu
Python	3.24	2.60	2.53	3%
Visual Basic	6.26	0.16	0.31	-94%

Lưu ý: Thực tiễn không phải lúc nào tối ưu hóa lý thuyết cũng mang lại hiệu quả tối đa, nên cần kiểm thử cụ thể.

### c. Khởi tạo tại thời điểm biên dịch (compile time)

Nếu sử dụng hằng số (constant) hoặc số ma thuật (magic number) trong lệnh gọi hàm, bạn nên tính toán trước và gán vào hằng số, tránh lặp lại tính toán tại runtime.

### Ví dụ tính logarit cơ số 2 trong C++:

```
// Cách ban đầu
unsigned int Log2(unsigned int x) {
    return (unsigned int) (log(x) / log(2));
}

// Sau khi tối ưu hóa:
const double LOG2 = 0.69314718;
unsigned int Log2(unsigned int x) {
    return (unsigned int) (log(x) / LOG2);
}
```

### Kết quả:

Ngôn ngữ	Thời gian gốc	Thời gian tối ưu	Tiết kiệm (%)
C++	9.66	5.97	38%
Java	17.0	12.3	28%
PHP	2.45	1.50	39%

### d. Cẩn trọng khi dùng hàm hệ thống (system routines)

Hàm hệ thống thường rất chính xác nhưng lại chậm, đôi khi vượt quá yêu cầu của bài toán. Nếu bạn không cần đến độ chính xác cao, nên cân nhắc giải pháp thay thế hoặc tối ưu hóa với kiểu dữ liệu phù hợp.

## 5. Tóm tắt và lưu ý

- **Đo lường là yếu tố then chốt** trước khi áp dụng bất kỳ tối ưu hóa nào.
- **Tối ưu hóa có thể làm tăng độ phức tạp mã, nguy cơ lỗi cao hơn.**
- **Không có công thức tuyệt đối:** Hiệu quả tối ưu hóa thay đổi mạnh tùy ngữ cảnh, ngôn ngữ, trình biên dịch và đặc tính phần cứng.

**Lời khuyên:** Luôn kiểm thử trong điều kiện thực tế của dự án trước khi áp dụng các kỹ thuật tối ưu hóa trên diện rộng.

## Kỹ thuật tối ưu hóa code – Dịch thuật học thuật

### Kỹ thuật sử dụng phép toán nguyên (integer operations)

```
if ( x < 2147483648 ) return 30;
return 31 ;
}
```

Chương trình này sử dụng hoàn toàn phép toán số nguyên, không chuyển đổi sang số thực dấu phẩy động (floating point), và vượt trội rõ rệt so với các phiên bản sử dụng số thực:

Ngôn ngữ	Thời gian gốc	Thời gian tối ưu	Tiết kiệm (%)	Tỷ lệ hiệu năng
C++	9.66	0.662	93%	15:1
Java	17.0	0.882	95%	20:1
PHP	2.45	3.45	-41%	2:3

Hầu hết các hàm “siêu việt” (transcendental functions) đều được thiết kế cho trường hợp xấu nhất – tức là chúng chuyển đổi về số thực dấu phẩy động chính xác kép (double-precision floating point) kể cả khi đầu vào là số nguyên. Nếu gặp những hàm này ở vùng code cần hiệu suất cao và không cần độ chính xác tuyệt đối, hãy cân nhắc thay đổi giải pháp.

Một lựa chọn khác là lợi dụng thực tế rằng phép dịch phải (right-shift) tương đương với phép chia cho hai. Số lần có thể chia một số cho hai (mà kết quả vẫn khác không) cũng là logarit cơ số 2 của số đó. Cách cài đặt dựa trên nhận xét này như sau:

### Ví dụ C++: Hàm tính log<sub>2</sub> bằng phép dịch phải

```
unsigned int Log2( unsigned int x ) {
    unsigned int i = 0;
    while ( ( x = ( x >> 1 ) ) != 0 ) {
        i++;
    }
    return i;
}
```

Đối với lập trình viên không dùng C++, đoạn code này khó đọc, đặc biệt là điều kiện vòng lặp vừa gán vừa so sánh – một thực hành nên tránh trừ khi thật cần thiết.

Chương trình này chạy lâu hơn bản trước đó khoảng 350%, nhưng vẫn nhanh hơn phương án ban đầu và dễ thích ứng cho môi trường 32-bit, 64-bit, v.v. Ví dụ này cho thấy tầm quan trọng của việc không dừng lại sau một lần tối ưu thành công: tối ưu hóa đầu tiên tiết kiệm 30-40% nhưng các tối ưu tiếp theo mới đem lại hiệu quả rõ rệt.

---

## Sử dụng hằng số cùng kiểu dữ liệu

Hãy sử dụng các hằng số được khai báo tên (named constants) hoặc literal cùng kiểu dữ liệu với biến nhận giá trị đó. Nếu hằng số và biến khác kiểu, trình biên dịch phải thực hiện chuyển đổi kiểu (type conversion). Một số trình biên dịch tốt sẽ chuyển đổi tại thời điểm biên dịch (compile time), giảm ảnh hưởng tới thời gian chạy. Ngược lại, trình biên dịch kém hoặc trình thông dịch (interpreter) sẽ phải chuyển đổi lúc chạy, có thể giảm hiệu năng.

### Ví dụ:

Nếu khai báo như sau (giả định *x* là số thực và *i* là số nguyên):

```
x = 5
i = 3.14
```

—> Phải chuyển đổi kiểu dữ liệu.

Nếu khai báo:

```
x = 3.14
i = 5
```

—> Không cần chuyển đổi.

Kết quả hiệu suất khi sử dụng hoặc không sử dụng chuyển đổi kiểu dữ liệu cho thấy sự khác biệt rõ rệt giữa các trình biên dịch:

Ngôn ngữ	Thời gian gốc	Thời gian tối ưu	Tiết kiệm (%)	Tỷ lệ hiệu năng
C++	1.11	0.000	100%	Không đo được
C#	1.49	1.48	<1%	1:1
Java	1.66	1.11	33%	1.5:1
Visual Basic	0.721	0.000	100%	Không đo được
PHP	0.872	0.847	3%	1:1

---

## Tính toán trước kết quả (Precompute Results)

Một quyết định quan trọng về tối ưu hóa là nên tính toán kết quả ngay khi cần (on the fly), hay nên tính toán một lần, lưu lại và tra cứu về sau (lookup). Nếu kết quả được sử dụng nhiều lần, tra cứu sẽ rẻ hơn tính toán động.

### Mức đơn giản:

Chỉ cần đưa các phép tính không đổi ra ngoài vòng lặp.

### Mức phức tạp hơn:

Tính trước một bảng tra cứu (lookup table) lúc chương trình khởi động, lưu trên file, hoặc khởi tạo tại biên dịch.

### Ví dụ: Game mô phỏng không gian

Ban đầu, lập trình viên tính toán hệ số lực hấp dẫn (gravity coefficients) theo khoảng cách với mặt trời. Tuy nhiên, số khoảng cách có thể nhận giá trị thực tế không nhiều, nên hệ số này được tính trước và lưu trong một mảng 10 phần tử.

### Java: Hàm tính tiền trả góp phức tạp, có thể tối ưu

```
double ComputePayment (
    long loanAmount,
    int months,
    double interestRate
) {
    return loanAmount /
        (
            ( 1.0 - Math.pow( ( 1.0 + ( interestRate / 12.0 ) ), -months ) ) /
            ( interestRate / 12.0 )
        );
}
```

Hàm này phức tạp và tính toán tốn kém. Có thể thay bằng bảng tra cứu giá trị mẫu cho biến `interestRate` và `months`, chỉ còn phép chia cho `loanAmount`.

### Sơ đồ xây dựng bảng tra cứu (lookup table):

- `interestRate`: từ 5% đến 20%, mỗi 0.25% (61 giá trị)
- `months`: 12 đến 72 (61 giá trị)
- `loanAmount`: có thể rất lớn, nhưng phần này đơn giản, chỉ cần phép chia

```
double ComputePayment (
    long loanAmount,
    int months,
    double interestRate
) {
    int interestIndex = Math.round( ( interestRate - LOWEST_RATE ) * GRANULARITY * 100.00 );
    return loanAmount / loanDivisor[ interestIndex ][ months ];
}
```

Hiệu quả sau tối ưu: | Ngôn ngữ | Thời gian gốc | Thời gian tối ưu | Tiết kiệm (%) | Tỷ lệ hiệu năng | |-----|-----|-----|-----|  
-----|-----|-----|-----| | Java | 2.97 | 0.251 | 92% | 10:1 | | Python | 3.86 | 4.63 | -20% | 1:1 |

---

## Tối ưu hóa nội vòng lặp – precompute trong loop

Dù không xây bảng tra cứu, có thể đưa phần biểu thức phức tạp ra ngoài vòng lặp để tăng tốc. Ví dụ:

### Java: Precompute phần mẫu số (divisor)

```
double ComputePayments (
    int months,
    double interestRate
) {
    double divisor = ( 1.0 - Math.pow( 1.0+(interestRate/12.0), - months ) ) /
        ( interestRate/12.0 );
    for ( long loanAmount = MIN_LOAN_AMOUNT; loanAmount <= MAX_LOAN_AMOUNT; loanAmount++ ) {
        payment = loanAmount / divisor;
    }
}
```

## Loại bỏ các biểu thức con chung (Eliminate Common Subexpressions)

Khi gặp biểu thức xuất hiện nhiều lần, hãy gán vào biến và dùng lại thay vì tính lại.

### Ví dụ Java: Trước tối ưu

```
payment = loanAmount /  
(  
    ( 1.0 - Math.pow( 1.0 + ( interestRate / 12.0 ), -months ) ) /  
    ( interestRate / 12.0 )  
);
```

### Sau tối ưu

```
monthlyInterest = interestRate / 12.0;  
payment = loanAmount /  
(  
    ( 1.0 - Math.pow( 1.0 + monthlyInterest, -months ) ) /  
    monthlyInterest  
);
```

Lợi ích tăng hiệu năng ở đây không lớn – nguyên nhân có thể do chi phí hàm `Math.pow()` quá lớn so với biểu thức con hoặc trình biên dịch đã tự động tối ưu.

---

## Tối ưu hóa Routine

Một chiến lược quan trọng trong tối ưu hóa code là chia nhỏ chương trình thành các routine (hàm/chương trình con) hợp lý, dễ tối ưu và tái sử dụng.

- Routine nhỏ, rõ ràng giúp tiết kiệm bộ nhớ, vì tránh trùng lặp code ở nhiều nơi.
- Dễ refactor, cải thiện mọi nơi gọi routine.
- Routine dài, phức tạp – khó tối ưu, đặc biệt ở ngôn ngữ cấp thấp như assembler.

### “Viết lại inline”

Ngày xưa, gọi routine bị phạt hiệu năng lớn do phải swap bộ nhớ. Ngày nay, hiệu năng giảm không đáng kể. Sử dụng tính năng `inline` (C++), macro, hoặc viết lại code trực tiếp ở nơi cần thiết cũng hiếm đem lại hiệu quả rõ rệt – thậm chí đôi khi làm hiệu suất kém hơn.

---

## Viết lại bằng ngôn ngữ cấp thấp (Recoding in a Low-Level Language)

Đây là giải pháp cuối cùng khi gặp “nút cổ chai” hiệu năng:

1. Viết toàn bộ app bằng ngôn ngữ cấp cao.
2. Kiểm thử kỹ càng.
3. Nếu cần tăng hiệu năng, dùng “profiling” để xác định vùng nóng (hot spot, thường là 5% code chiếm 50% thời gian).
4. Viết lại chỉ những đoạn này bằng assembler hoặc ngôn ngữ cấp thấp.

Ví dụ, chuyển đổi routine xử lý bit từ Delphi sang assembler tiết kiệm 41% thời gian; từ C++ sang assembler tiết kiệm 29%. Lợi ích sẽ thấp hơn nếu ngôn ngữ gốc đã tối ưu tốt cho bit manipulation.

Quy trình đơn giản là:

- Lấy listing assembler do compiler sinh ra,
  - Trích xuất routine,
  - Tối ưu thủ công từng chút một,
  - Kiểm thử và đo hiệu quả từng bước.
- 

## Tổng kết: Checklist Tối ưu hóa code

## Cải thiện đồng thời tốc độ và kích thước:

- Thay thế logic phức tạp bằng bảng tra cứu (table lookup).
- Jam loop.
- Dùng kiểu số nguyên thay cho floating-point.
- Khởi tạo dữ liệu ở compile-time.
- Sử dụng hằng số đúng kiểu.
- Tính trước kết quả (precompute).
- Loại bỏ biểu thức con chung (common subexpression).
- Viết lại routine then chốt bằng ngôn ngữ cấp thấp.

## Chỉ cải thiện tốc độ:

- Ngừng kiểm tra khi đã đủ thông tin.
- Sắp xếp thứ tự kiểm tra theo xác suất xảy ra.
- So sánh hiệu năng của các cấu trúc logic tương tự.
- Sử dụng lazy evaluation.
- Unswitch loop, unroll loop.
- Giảm số phép toán thực hiện trong vòng lặp.
- Sử dụng sentinel cho vòng lặp tìm kiếm.
- Đặt vòng lặp nhiều phép toán nhất vào trong cùng.
- Khai thác các đồng nhất đại số (algebraic identity).
- Cảnh giác với hàm hệ thống (system routines).
- Viết lại routine theo kiểu inline.

---

## Ghi chú về tính quan trọng của tối ưu hóa code

- Máy tính ngày nay nhanh hơn và nhiều bộ nhớ hơn. Nhiều lớp tối ưu hóa mức low-level đã dần ít quan trọng hơn ở ứng dụng thông thường.
- Tuy nhiên, lập trình nhúng (embedded systems), real-time hoặc hệ thống cần tiết kiệm cực đoan về tài nguyên vẫn cần các kỹ thuật này.
- Việc đo đạc mỗi lần tối ưu hóa vẫn là nguyên tắc không đổi từ thời Donald Knuth.
- Ảnh hưởng của mỗi kỹ thuật tối ưu hóa ngày càng khó dự đoán do phụ thuộc ngôn ngữ, phiên bản compiler, thư viện, và cài đặt.

**Tóm lại:** Mỗi lần tối ưu hóa cần được kiểm chứng bằng đo đạc cụ thể trong thực tế sử dụng. Không có giải pháp chung cho mọi trường hợp.

## Giới thiệu về chi phí bảo trì và tối ưu hóa mã nguồn

Nếu áp dụng quá trình tối ưu hóa một cách bừa bãi, **chi phí bảo trì** sẽ tăng đáng kể do tất cả các thao tác tái kiểm tra hiệu suất (reprofiling) cần thiết. *Kinh nghiệm cho thấy*, việc luôn yêu cầu sự cải thiện có thể đo lường được là một phương pháp tốt để chống lại cảm dỗ tối ưu hóa trước khi thực sự cần thiết và giúp đảm bảo xu hướng ưu tiên mã nguồn rõ ràng, đơn giản.

Nếu một **tối ưu hóa** đủ quan trọng đến mức khiến bạn phải sử dụng công cụ profiler (trình phân tích hiệu suất) để đo lường tác động, thì có lẽ nó cũng đủ quan trọng để thông qua—miễn là tối ưu hóa đó hoạt động hiệu quả. Nhưng nếu tối ưu hóa đó không đủ quan trọng để bạn phải sử dụng profiler, thì nó cũng không đủ quan trọng để đánh đổi sự dễ đọc, tính dễ bảo trì, và các đặc tính quan trọng khác của mã nguồn.

Tác động của việc điều chỉnh mã mà không đo đạc là **giả định** (speculative) trong trường hợp tốt nhất, trong khi tác động lên tính dễ đọc là điều chắc chắn cũng như chắc chắn gây hại.

## Tài nguyên tham khảo thêm

- [cc2e.com/2679](https://cc2e.com/2679)
- **Tài liệu tham khảo ưa thích về điều chỉnh mã nguồn** là *Writing Efficient Programs* (Bentley, Englewood Cliffs, NJ: Prentice Hall, 1982). Cuốn sách này hiện đã ngừng xuất bản nhưng rất đáng đọc nếu bạn tìm được. Đây là một nghiên cứu chuyên sâu về tối ưu hóa mã nguồn trên nhiều khía cạnh khác nhau.

Bentley mô tả các kỹ thuật đánh đổi giữa **thời gian** (time) và **bộ nhớ** (space), cũng như các ví dụ về cách thiết kế lại kiểu dữ liệu nhằm giảm thiểu cả bộ nhớ lẫn thời gian xử lý. Phương pháp của ông mang tính kể chuyện (anecdotal) nhiều hơn so với nội dung bản luận ở đây, và các câu chuyện của ông rất hấp dẫn. Ông đưa ra nhiều đoạn mã qua các bước tối ưu hóa khác nhau để minh họa ảnh hưởng của lần tối ưu hóa đầu tiên, thứ hai, và thứ ba lên một bài toán cụ thể. Bentley trình bày nội dung chính của cuốn sách chỉ trong 135 trang.

Cuốn sách có tỷ lệ "thông tin/tranh chấp" (signal-to-noise ratio) rất cao—nó là một trong số ít những viên ngọc quý mà mọi lập trình viên chuyên nghiệp nên sở hữu.

**Phụ lục 4** của *Programming Pearls* (ấn bản lần 2, Boston, MA: Addison-Wesley, 2000) cung cấp tóm tắt các quy tắc tối ưu hóa mã từ cuốn sách trước đó của Bentley.