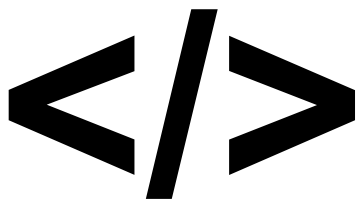


97 điều mọi lập trình viên nên biết

Tác giả: Kevlin Henney

Dịch bởi: CodersX Team

(Bản dịch v1, thông tin ở cuối sách)

Mục lục:

Phần 1: Hành động một cách thận trọng (Act with Prudence)

Nếu bạn đảm nhận bất cứ việc gì, hãy làm một cách thận trọng và cân nhắc những ảnh hưởng sau này. - Anon

Bất kể công việc của bạn có vẻ dễ dàng thế nào trong giai đoạn đầu dự án, bạn không thể tránh khỏi những lúc bị áp lực. Nếu bạn phải chọn giữa “làm đúng” và “làm nhanh”, thường thì bạn sẽ chọn “làm nhanh” và có suy nghĩ trong đầu là sẽ quay lại fix nó sau. Khi bạn hứa điều này với chính bạn, với team, với khách hàng, nghĩa là bạn có chủ ý như vậy. Nhưng hầu như thì, sẽ nảy sinh ra những vấn đề mới ở những công việc tiếp theo và bạn sẽ tập trung vào những vấn đề này. Cái kiểu trì hoãn lại công việc để làm sau như vậy thường được coi là “technical debt” (nợ kĩ thuật) và nó không được tốt cho lắm. Đặc biệt Martin Fowler gọi cái kiểu tech debt này là tech debt có chủ ý (deliberate technical debt) để tránh nhầm lẫn với tech debt vô ý (inadvertent technical debt).

Tech debt giống như một món vay lãi: bạn nhận được lợi ích ngắn hạn từ nó, nhưng bạn sẽ phải trả cả lãi cho đến khi nó được thanh toán hoàn toàn. Những lối tắt trong code làm cho việc thêm feature và refactor code khó khăn hơn. Chúng là nơi sản sinh ra những sai sót và những test case dễ đổ bể. Để chúng đó càng lâu thì càng nguy hiểm. Cho đến lúc bạn phải fix cái chỗ ban đầu mà bạn tạo ra, có thể có tới một đống bùi nhùi được xây dựng trên phần code sai ban đầu, làm cho nó khó sửa và refactor. Thực tế, thường chỉ khi mọi thứ trở nên quá tệ đến nỗi bạn phải fix cái lỗi ban đầu thì bạn mới fix. Còn trước đó thì thường là khó để bạn có thể dành thời gian hoặc cân nhắc rủi ro để sửa nó.

Có những lúc bạn sẽ phải gánh chịu các tech debt để kịp deadline hoặc làm một feature nhỏ. Cố đừng để vào tình huống đó, nhưng nếu tình huống thực sự đòi hỏi phải làm vậy thì vẫn phải làm thôi. NHƯNG bạn phải track các tech debt và trả nợ nhanh chóng, hoặc mọi thứ sẽ lao dốc không phanh. Ngay sau khi quyết định gánh tech debt,

viết ngay một task hoặc lưu lại trong hệ thống track issue của bạn để đảm bảo là bạn không quên nó.

Nếu bạn lên kế hoạch trả nợ trong tuần làm việc tới, giá phải trả sẽ là thấp nhất. Để nợ đó mà không trả sẽ sinh lãi, lãi này nên được theo dõi một cách minh bạch. Việc làm này sẽ giúp nhấn mạnh được mức độ ảnh hưởng của tech debt tới business value và sẽ giúp đánh giá cao việc trả nợ. Làm thế nào để tính và theo dõi phần nợ lãi dự vào project của bạn, nhưng bạn phải track nó.
Trả nợ tech càng sớm càng tốt. Còn không thì đồng nghĩa với việc bạn thiếu thận trọng.

Phần 2. Ứng dụng các nguyên tắc của Functional Programming

Functional Programming gần đây lấy lại được sự yêu thích từ cộng đồng lập trình. Một phần là bởi vì những tính chất mới của functional programming được đưa ra để giải quyết các khó khăn gặp phải trong ngành của chúng ta. Tuy nhiên, dù đấy là một ứng dụng quan trọng, nhưng nó cũng không phải là lý do buộc bạn phải biết functional programming.

Việc làm chủ được functional programming có thể cải thiện được chất lượng code của bạn. Nếu bạn hiểu sâu và áp dụng được mô hình này, thiết kế của bạn sẽ rất minh bạch.

Referential transparency là một yếu tố rất được mong muốn: nó cho thấy rằng một hàm luôn có kết quả giống nhau khi được thực thi với đầu vào giống nhau, mọi lúc mọi nơi. Nghĩa là, việc tính toán hàm ít (hoặc không) dựa vào side effect của mutable state.

Một trong những nguyên nhân gây lỗi trong code kiểu mệnh lệnh là do các biến có thể bị mutate. Những ai đang đọc bài này thì chắc cũng đã từng kiểm tra tại sao thi thoảng trong một số trường hợp thì giá trị ra không mong muốn. Sự rõ ràng trong ngữ nghĩa có

thể giúp làm giảm bớt những lỗi âm thầm như vậy, hoặc ít nhất thì cũng thu hẹp được phạm vi của nó, nhưng nguyên nhân sâu xa của những lỗi này là do các thiết kế mà cho phép mutate quá nhiều.

Và chúng ta khó có thể nhận được sự giúp đỡ từ xung quanh cho các lỗi kiểu này. Lập trình hướng đối tượng âm thầm gây ra vấn đề này, bởi vì nó thường cho thấy những kết nối giữa những object gọi qua lại những method mutate lẫn nhau, việc này khá nguy hiểm.

Tuy nhiên, với kiểu thiết kế test-driven, đặc biệt khi tuân theo phương pháp “Mock Roles, not Objects”, các mutability không cần thiết sẽ bị loại bỏ.

Kết quả cuối cùng sẽ là thiết kế mà có phân chia trách nhiệm rõ ràng, cùng với nhiều function nhỏ hơn, chỉ thực hiện tính toán trên các đầu vào của chúng thay vì tham chiếu đến các biến có thể bị mutate. Làm như vậy sẽ có ít lỗi hơn, và hơn nữa là sẽ dễ debug, vì nó có thể dễ dàng chỉ ra vị trí của giá trị bị sai trong những kiểu thiết kế như vậy hơn là phải lần ra những ngữ cảnh cụ thể dẫn đến cái lỗi đó. Việc này làm tăng đáng kể tính minh bạch trong code, và việc học một ngôn ngữ functional sẽ giúp cho bạn hiểu thấu tận xương.
Tất nhiên là functional programming không phải tối ưu trong mọi bài toán. Ví dụ như trong các hệ thống hướng đối tượng, kiểu lập trình này thường cho kết quả tốt hơn với việc phát triển domain model hơn là giao diện người dùng.

Hãy làm chủ functional programming để có thể khéo léo ứng dụng các bài học này cho các lĩnh vực khác. Những hệ thống object sẽ cộng hưởng với những cái tốt của sự minh bạch và sẽ gần với những hệ thống functional hơn là bạn nghĩ. Trong thực tế, có những người khẳng định rằng, functional programming và hướng đối tượng chỉ đơn giản là sự phản chiếu lẫn nhau, một dạng âm dương trong xử lý máy tính.

Bài viết này hơi trừu tượng với người mới học nên mình sẽ thêm các tài liệu tham khảo dưới đây.

- Các thuật ngữ cần nắm: functional programming, mutability/immutability, pure function/impure function, referential transparency, imperative programming,

object-oriented programming, side effect, function composition, first-class function, high order function

- Functional programming: https://medium.com/javascript-scene/master-the-javascript-interview-what-is-functional-programming-7f218c68b3a0

Phần 3. Hãy hỏi “Người dùng họ sẽ làm gì?” (Bạn không phải là người dùng)

CHÚNG TA THƯỜNG CÓ XU HƯỚNG CHO RẰNG NGƯỜI KHÁC CŨNG NGHĨ NHƯ MÌNH. Nhưng họ không. Các nhà tâm lý học gọi đây là “false consensus bias”. Khi người khác nghĩ và làm khác với chúng ta, chúng ta thường cho rằng họ không bình thường.

Cái việc suy nghĩ như vậy giải thích tại sao nhiều lập trình viên gặp khó khăn trong việc đặt mình vào vị trí của người dùng. Người dùng không nghĩ như lập trình viên. Đầu tiên là họ ít dùng máy tính hơn chúng ta nhiều. Họ không biết mà cũng không quan tâm máy tính hoạt động ra sao. Điều này có nghĩa là họ không quen với những kỹ năng giải quyết vấn đề như lập trình viên. Họ không nhận ra những cái dạng mẫu mà lập trình viên hàng ngày thấy.

Cách tốt nhất để biết người dùng suy nghĩ thế nào là ngồi theo dõi họ. Yêu cầu một người dùng sử dụng một phần mềm nào đó giống cái bạn đang làm để giải quyết một công việc bất kì. Hãy đảm bảo công việc này là một công việc có thật: ví dụ “Cộng một cột gồm các số” có vẻ OK. Ví dụ “Tính chi tiêu tháng trước của bạn” thì tốt hơn. Tránh những công việc quá đặc thù, ví dụ như “Bạn có thể chọn những ô trong bảng tính và điền một công thức tính tổng vào bên dưới không?” - nó quá rõ ràng. Hãy để người dùng nói lên suy nghĩ của họ. Đừng ngắt họ. Đừng cố giúp họ. Hãy tự hỏi, “Tại sao họ lại làm vậy?” và “Tại sao họ không làm khác?”.

Điều đầu tiên bạn sẽ để ý thấy là mọi người dùng đều làm giống nhau. Họ sẽ thử hoàn thành công việc theo thứ tự giống nhau, cùng gặp phải những lỗi tương tự nhau ở cùng một chỗ. Bạn cần phải thiết kế dựa vào những hành vi đó. Cái này khác với những buổi họp về thiết kế, chỗ mà mọi người thường nghe ai đó nói “Nếu người dùng muốn…thì sao?”. Điều này dẫn đến những tính năng phức tạp và khó hiểu hơn là cái người dùng muốn. Theo dõi người dùng sẽ xoá bỏ những sự khó hiểu này.

Bạn sẽ thấy người dùng gặp vướng mắc. Khi bạn bị tắc, bạn sẽ ngó xung quanh. Khi người dùng bị tắc, họ sẽ thu hẹp tầm nhìn của họ. Nó làm cho họ khó thấy được giải pháp ở đâu đó trên màn hình của họ. Đấy là một lý do tại sao dùng những đoạn text dùng để giải thích là phương án không tốt trong UI design. Nếu như bạn phải dùng đến những chỉ dẫn hoặc text trợ giúp, hãy đảm bảo là nó nằm ngay tại chỗ vấn đề xảy ra. Sự thu hẹp tập trung chú ý của người dùng là lý do tại sao tool tips thường hiệu quả hơn menu trợ giúp.

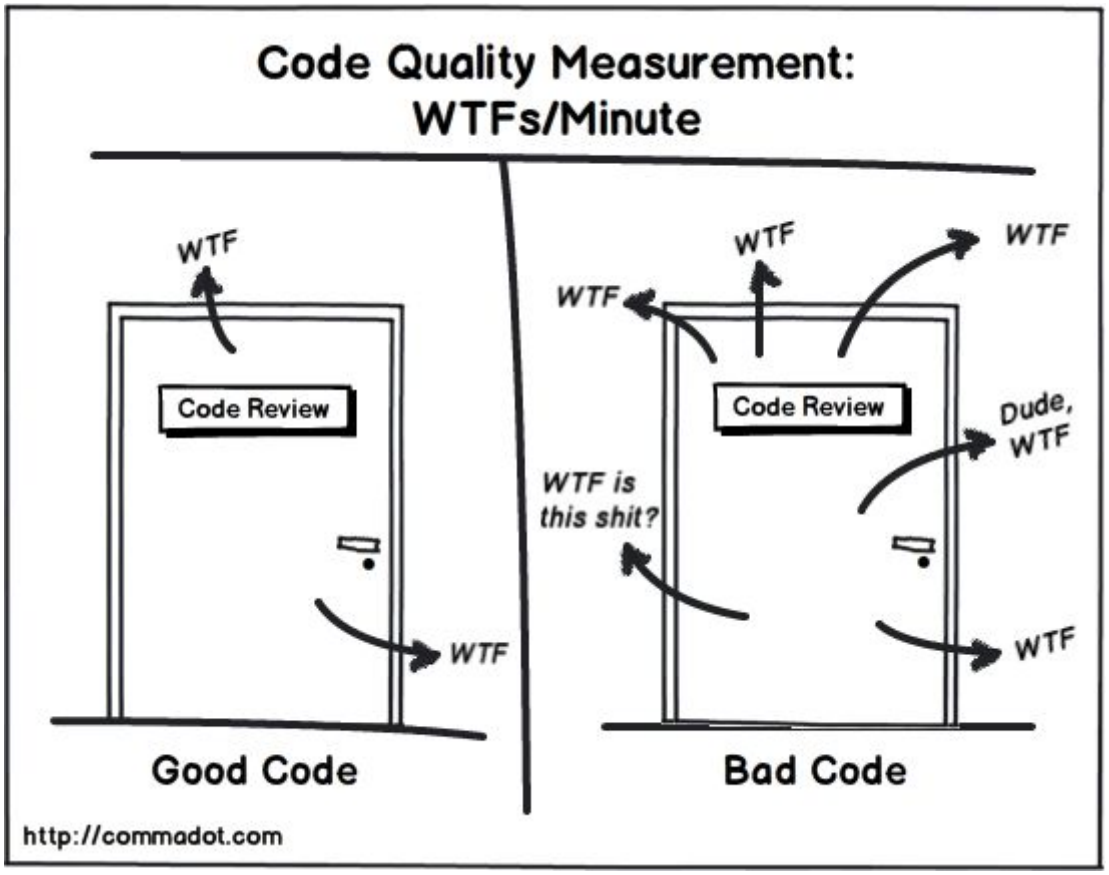
Người dùng thường có xu hướng bị loạn. Họ tìm một cách làm cho nó hoạt động được và sẽ dính với nó mãi, bất kể nó lòng vòng ra sao. Tốt hơn thì nên đưa ra một cách làm thật rõ ràng hơn là vài ba phím tắt.

Bạn cũng sẽ thấy là có một khoảng cách giữa cái người dùng nói là họ muốn và cái họ làm. Điều này khá quan ngại, vì cách thông thường mà ta thu thập mong muốn từ người dùng là hỏi họ. Đấy là lí do tại sao cách tốt nhất để thu thập nhu cầu người dùng là theo dõi họ. Dành hàng giờ để theo dõi người dùng thì mang lại nhiều thông tin hơn là dành cả ngày đoán xem họ muốn gì.

Tham khảo:

Từ khoá: user tracking, user behavior, user analytics, user interface, user experience, user testing, A/B test

Phần 4. Tự động hoá tiêu chuẩn code



Có thể bạn cũng đã từng rơi vào trường hợp này rồi. Đó là ở giai đoạn đầu dự án, mọi người có chủ ý tốt - gọi là “những quy tắc trong dự án”. Hầu hết những quy tắc này được ghi lại trong tài liệu. Những quy tắc về code thì sẽ rơi vào phần coding standard (tiêu chuẩn code) của dự án. Trong buổi họp kick-off dự án, lead dev đọc qua một lần tài liệu này và trong trường hợp tốt nhất, mọi người cùng đồng ý là sẽ tuân theo. Khi dự án bắt đầu thì những ý định tốt ban đầu bị bỏ rơi, từng tí một. Khi dự án được bàn giao, code trông như một đống rác, và không ai biết tại sao nó lại trở nên như vậy.

Mọi thứ tuột dốc khi nào? Khả năng là tại lúc kick-off meeting của dự án. Một vài thành viên không chú ý. Những người khác thì không hiểu mục đích. Tệ hơn, có người không đồng ý và còn lên kế hoạch đi ngược lại với những coding standard này. Cuối cùng, một vài người hiểu và đồng ý, nhưng khi áp lực của dự án tăng lên, họ phải đi lệch ra khỏi quỹ đạo. Code đẹp không ghi điểm trong mắt khách hàng, người muốn nhiều chức năng hơn là code. Hơn nữa, tuân theo chuẩn code có thể sẽ gây chán nếu như nó

không được làm tự động. Không tin thì bạn cứ thử căn hàng thủ công cho một đoạn code class rối rắm.

Nhưng nếu nó đã là vấn đề như vậy, tại sao chúng ta vẫn muốn có một chuẩn code ngay từ đầu? Lý do cho việc format code một cách đồng bộ là để cho không ai có thể "nắm giữ" một phần code cho chính họ bằng việc format code theo ý mình. Hoặc chúng ta có thể muốn tránh việc dev sử dụng các antipattern để tránh những lỗi thường gặp. Suy cho cùng, một chuẩn code nên làm cho việc làm việc trong cùng dự án dễ dàng hơn, và giữ được tốc độ dev từ đầu tới cuối. Nếu xác định tuân theo thì mọi người cũng nên đồng ý với chuẩn code - nó sẽ giúp tránh trường hợp dev thì dùng 3 space để căn hàng, dev khác thì dùng 4 space.

Có một loạt các tool giúp báo cáo chất lượng code, bảo trì và lưu giữ chuẩn code nhưng chúng không phải giải pháp. Nó cần phải được tự động hoá và thực thi khi cần. Dưới đây là một số ví dụ:

- Đảm bảo việc format code là một phần của quá trình build để mọi người có thể chạy nó một cách tự động mỗi lần họ compile code.
- Dùng các tool phân tích code để rà soát các antipattern không mong muốn, nếu thấy thì huỷ bỏ bản build đó.
- Học cách cấu hình các tool đó để có thể scan các antipattern trong project của bạn.
- Đừng chỉ đo đạc test coverage, mà tự động check kết quả nữa. Huỷ bản build nếu test coverage thấp.

Cố gắng làm việc này với tất cả các dự án bạn cho là quan trọng. Bạn không thể tự động hoá mọi thứ mà bạn muốn. Với những thứ bạn không thể tự động đánh dấu hoặc fix, xem chúng như một tập các chỉ dẫn bổ sung cho chuẩn code đã được tự động hoá, nhưng hãy chấp nhận việc đồng nghiệp sẽ có lúc không tuân theo.

Cuối cùng, chuẩn code nên linh hoạt thay vì cứng nhắc. Khi project phát triển, sẽ có những thay đổi, và những thứ có vẻ là đúng đắn lúc đầu thì chưa chắc vài tháng sau vẫn còn là như vậy.

Bài viết này có tính bao quát, áp dụng cho mọi dự án. Đối với JavaScript, bạn có thể dùng các công cụ sau để đảm bảo chuẩn code trong team:

- Eslint giúp tránh các lỗi trong code bằng việc đưa ra các quy tắc. Ví dụ như no-dupe-keys rule sẽ báo lỗi khi bạn khai báo 2 property trùng tên trong cùng một literal object
- Prettier giúp mọi người code chung một format, bạn có thể đưa ra các config riêng cho team mình. Ví dụ rule "singleQuote": true nghĩa là dùng ' thay vì " cho string.
- Các trình duyệt như VSCode có plugin cho Eslint và Prettier để detect lỗi và format code trong quá trình code. Nhưng nếu bạn dùng các trình code khác không có các plugin này thì sao? Bạn không thể ép mọi người trong team cùng dùng một trình duyệt code. Thay vào đó bạn có thể dùng husky để tạo pre-commit hook (đọc thêm về githook), giúp kiểm tra và format code lúc commit code tại local. Tham khảo: https://medium.com/@joshuacrass/javascript-linting-and-formatting-with-eslint-prettier-and-airbnb-30eb746db862 (đọc kỹ và làm theo nhé)

Phần 5. Vẻ đẹp nằm trong sự đơn giản

Có câu nói của Plato mà tôi nghĩ khá tốt cho mọi dev nếu họ nắm được và giữ lấy cho mình:
Beauty of style and harmony and grace and good rhythm depends on simplicity.

Bằng một câu duy nhất, nó nói lên được giá trị mà chúng ta, những người dev, nên hướng tới.
Có vài thứ chúng ta vẫn phấn đấu khi code:

- Dễ đọc
- Dễ bảo trì
- Tốc độ phát triển
- The elusive quality of beauty (không biết dịch sao cho hay)

Plato nói với chúng ta rằng cái nhân tố quyết định tới những thứ nói trên đó là sự đơn giản.
Code đẹp là code thế nào? Câu hỏi này khá là chủ quan. Sự nhận thức về cái đẹp dựa vào kiến thức của mỗi người, giống như là nhân thức về mọi thứ của chúng ta đều dựa vào kinh nghiệm, kiến thức đã qua. Những người học nghệ thuật thì có nhận thức khác (hoặc cách tiếp cận khác) về vẻ đẹp so với người học khoa học. Những người học nghệ thuật thường tiếp cận về vẻ đẹp trong phần mềm bằng cách so sánh phần mềm với tác phẩm nghệ thuật, trong khi người học khoa học có xu hướng nói về tính đối xứng và tỉ lệ vàng, cố gắng thu gọn mọi thứ về các công thức. Theo kinh nghiệm của tôi, sự đơn giản là căn cứ cho phần lớn các lập luận của cả hai bên.

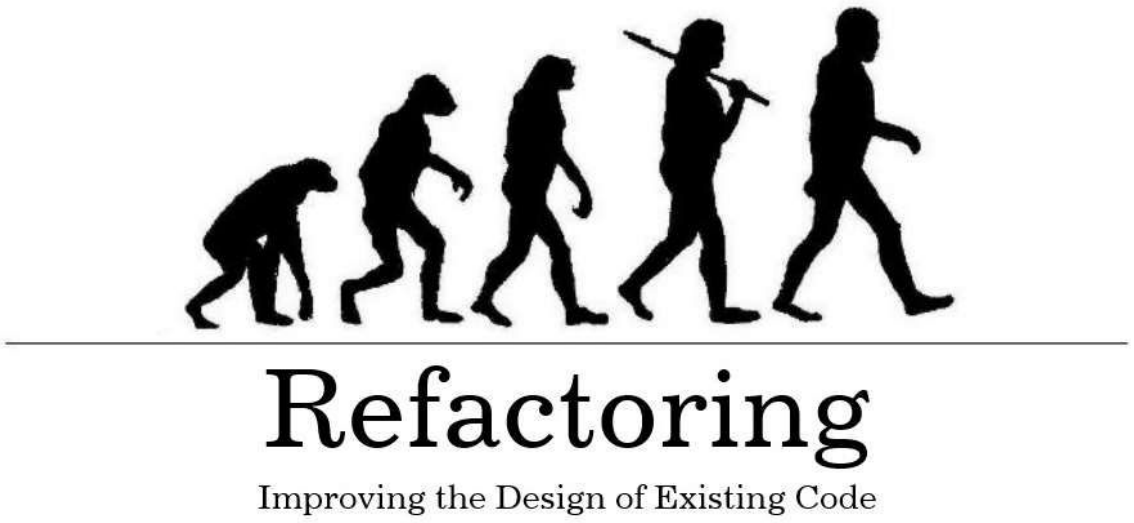
Hãy nghĩ về những đoạn code mà bạn đã học. Nếu bạn chưa dành thời gian xem code người khác, dừng đọc bài này lại và tìm code ai đó mà đọc. Tôi nói nghiêm túc đó! Tìm trên web các code tuỳ ngôn ngữ bạn muốn, viết bởi ai đó nổi tiếng, chuyên gia đã được biết đến.

Bạn quay lại rồi à? Tốt. Chúng ta đang nói đến đâu nhỉ? À ừ… Tôi tìm được đoạn code hợp với tôi, và tôi nghĩ là nó đẹp, có một vài điểm chung. Điểm chung nhất là tính đơn giản. Tôi thấy rằng không quan trọng toàn bộ hệ thống phức tạp ra sao, mỗi phần riêng cần phải được làm đơn giản: những object đơn giản đảm nhiệm một việc đơn giản chứa các method cũng đơn giản với những cái tên có ý nghĩa. Có người nghĩ viết những method ngắn 5-10 dòng là khoai, và khó làm đối với một vài ngôn ngữ, nhưng tôi nghĩ sự xúc tích như vậy là một mục tiêu đáng có.

Cái mà tôi muốn nói là code đẹp là code đơn giản. Mỗi phần nhỏ được làm đơn giản, đảm nhiệm công việc đơn giản, có các mỗi quan hệ đơn giản với các phần khác trong hệ thống. Đây là cách mà chúng ta giữ cho hệ thống có thể bảo trì được trong thời gian dài, cùng với code test được, đơn giản, sạch sẽ, đảm bảo tốc độ phát triển trong toàn bộ thời gian hoạt động của hệ thống.

Vẻ đẹp được sinh ra và tìm thấy trong sự đơn giản.

Phần 6. Trước khi bạn refactor



Một lúc nào đó, mỗi lập trình viên sẽ cần phải refactor code hiện tại. Nhưng trước khi làm việc này, hãy cân nhắc những việc dưới đây, bởi nó có thể giảm bớt thời gian và công sức cho bạn và người khác

- Phương pháp tốt nhất cho việc tái cấu trúc bắt đầu từ việc xem xét kỹ lưỡng code hiện tại và các đoạn test liên quan đến code đó. Việc này sẽ làm cho bạn hiểu về ưu nhược điểm của code hiện tại, sau đó bạn có thể chắc rằng sẽ giữ lại những điểm mạnh trong khi tránh được các lỗi. Chúng ta đều tưởng là mình có thể làm tốt hơn hệ thống hiện tại… cho đến cuối cùng chúng ta lại làm ra một cái mới không những không tốt hơn mà có khi còn tệ hơn cái cũ vì chúng ta không học được từ những sai sót của hệ thống hiện tại.
- Tránh viết lại từ đầu mọi thứ. Tốt nhất là tái sử dụng được càng nhiều code càng tốt. Bất kể code có xấu đến đâu, nó đã được test và review. Vứt bỏ code cũ - đặc biệt khi nó đang ở production - đồng nghĩa với việc bạn đang vứt đi công sức hàng tháng viết ra code đã được test, sửa lỗi và có thể có những bug fix,

workaround mà bạn không biết tới. Nếu bạn không cân nhắc vấn đề này, code mới sẽ có thể rơi vào tình trạng có các bug bí hiểm mà đã được fix ở code cũ. Nó sẽ làm cho bạn tốn thời gian, công sức, những kiến thức đã được tích luỹ qua nhiều năm.

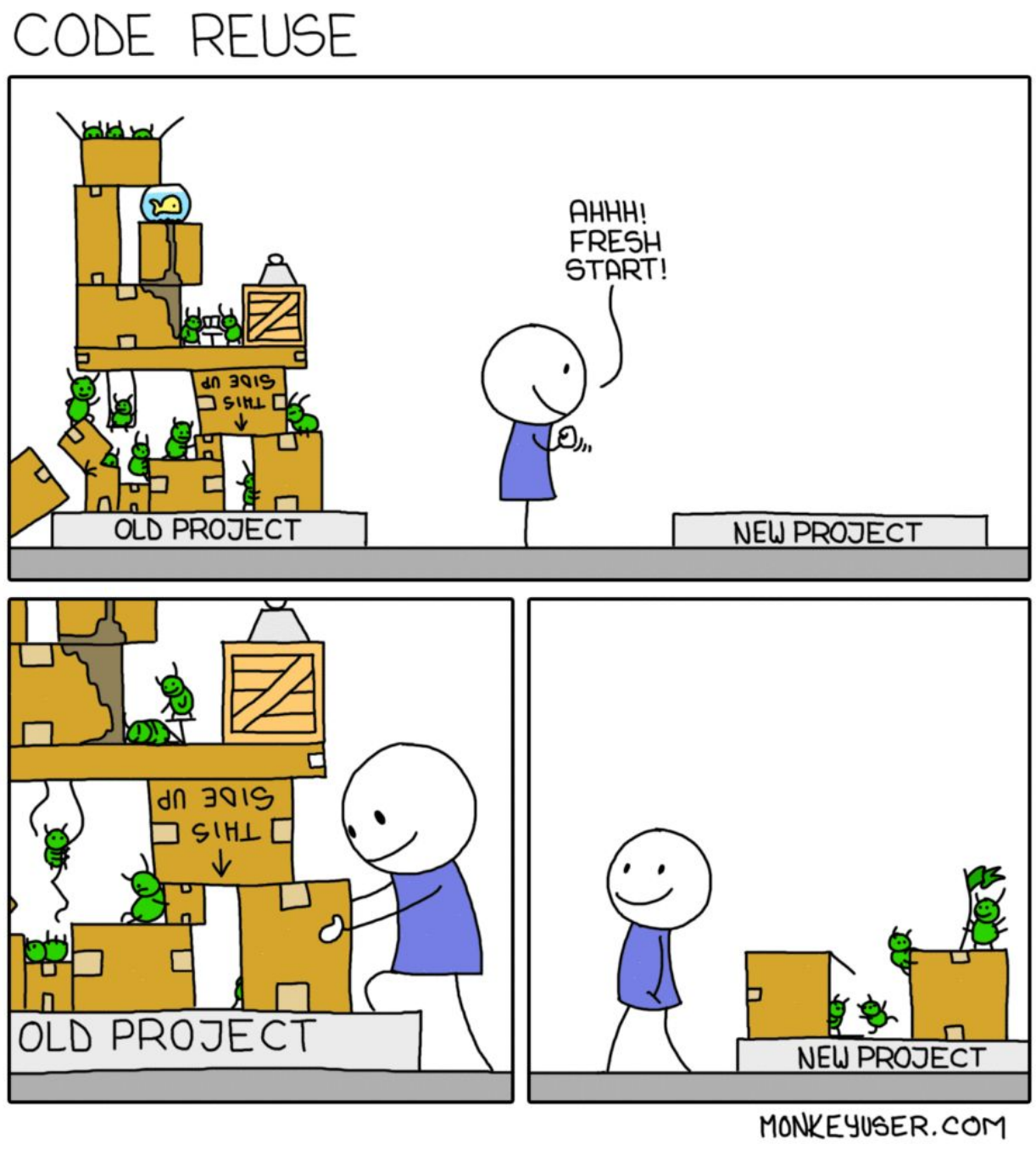
- Nhiều thay đổi dần dần tốt hơn là một sự thay đổi lớn đột ngột. Những thay đổi dần dần cho phép bạn đánh giá được sự ảnh hưởng lên hệ thống một cách dễ dàng qua những phản hồi, chẳng hạn từ việc test. Xem hàng trăm test case bị fail là việc bạn không muốn thấy. Nó dẫn đến sự bực bội, áp lực có thể gây ra các quyết định tồi sau đó. Một vài test case bị fail thì dễ đối phó hơn, dễ đưa ra giải pháp tốt hơn.
- Sau mỗi kì dev, đảm bảo các test case đều pass là việc cần phải làm. Viết các test mới nếu những test cũ không cover được các phần thay đổi mới. Đừng bỏ những test cũ nếu không có những cân nhắc thoả đáng. Ở bề ngoài, những test này có vẻ không áp dụng cho thiết kế mới của bạn, nhưng bạn nên dành thời gian để tìm hiểu lý do vì sao các test case đã được thêm trước đây.
- Đừng mang lựa chọn cá nhân và cái tôi vào. Nếu một thứ không hỏng, tại sao fix nó? Phong cách hoặc cấu trúc code không phù hợp với cá nhân bạn không phải là lý do chính đáng để tái cấu trúc code. Việc nghĩ rằng bạn có thể làm tốt hơn dev cũ cũng không phải là lý do chính đáng.
- Công nghệ mới không phải là lý do để refactor. Một trong những lý do tệ nhất để refactor là bởi vì code hiện tại quá cũ so với những công nghệ ngầu hơn chúng ta có ngày nay, và chúng ta tin vào ngôn ngữ hay framework mới có thể làm mọi thứ một cách nuột hơn. Nếu việc phân tích chi phí-lợi ích không cho thấy việc ngôn ngữ, framework mới đem lại cải thiện đáng kể về tính năng, khả năng bảo trì, hiệu quả làm việc, thì chúng ta nên giữ nguyên code.
- Nhớ là con người luôn có sai sót. Tái cấu trúc không đảm bảo rằng code mới sẽ tốt hơn - hoặc thậm chí tốt bằng - code cũ. Tôi đã từng thấy và trải qua vài lần tái cấu trúc không thành công. Nó không đẹp, nhưng nó là do người làm.

Qua bài viết này chúng ta có thể thấy việc “đập đi làm lại” là khá rủi ro, cần phải được cân nhắc kĩ càng. Thông thường khi quyết định viết lại code cũ, nếu code chưa được test thì chúng ta nên viết test để hiểu code cũ hoạt động ra sao. Sau khi test đã cover

được code cũ rồi, chúng ta bắt đầu refactor lại code và phải đảm bảo test vừa viết không break.

Một cái bẫy khác mà nhiều dev hay rơi vào là chạy theo công nghệ. Khi một công nghệ mới ra mắt, thường thì nó sẽ có nhiều bug mà chúng ta chưa lường tới. Hãy bình tĩnh và đưa ra quyết định sáng suốt: liệu việc chuyển sang công nghệ mới có thực sự giúp được không, và giúp được bao nhiêu?

Phần 7. Cẩn thận với việc dùng chung code



Đó là dự án đầu tiên của tôi trong công ty. Tôi vừa mới tốt nghiệp và khao khát được chứng minh bản thân, ở lại công ty muộn để xem những dòng code đang có. Khi làm tính năng đầu tiên, tôi đã hết sức cẩn thận vận dụng những gì đã học - ghi chú, log, lôi

những đoạn code xài chung ra thành các thư viện bất cứ chỗ nào có thể. Tôi đã rất sẵn sàng cho code review đã làm tôi ngạc nhiên - việc tái sử dụng code đã không được chấp nhận.

Làm sao có thể như thế được? Suốt khi học đại học, tái sử dụng được coi như là hình mẫu trong lập trình. Tất cả những bài viết mà tôi đã từng đọc, các cuốn sách, những chuyên gia phần mềm những người đã dạy tôi, không lẽ tất cả đều sai?

Hoá ra là tôi đã bỏ qua một thứ quan trọng.

Ngữ cảnh (Context).

Thực tế cho việc 2 phần khác nhau của hệ thống thực hiện 1 logic theo cùng một cách mang ít ý nghĩa hơn tôi tưởng. Trước lúc tôi lôi đoạn code chung ra thành thư viện, các phần này không phụ thuộc vào nhau. Mỗi phần sẽ có thể biến đổi khác nhau trong tương lai. Mỗi phần có thể thay đổi logic riêng của nó để phù hợp với sự thay đổi về business của hệ thống. 4 dòng code giống nhau chỉ là tình cờ.

Cái thư viện code dùng chung mà tôi tạo ra giống như việc buộc dây của chiếc giày này vào chiếc giày bên kia. Hai chân phải bước cùng nhau. Chi phí bảo trì cho các hàm độc lập thường không đáng kể, nhưng những thư viện chung thì sẽ đòi hỏi phải test khá nhiều.
Khi số dòng code tôi viết được giảm đi, ngược lại tôi đã làm cho nhiều thứ phụ thuộc vào nhau hơn.

Những cái lỗi này nó nguy hiểm ở chỗ, trông thì có vẻ là ý tưởng hay. Khi ứng dụng vào đúng chỗ, thì nó phát huy giá trị. Ngược lại sai chỗ thì nó tăng chi phí hơn là giá trị. Khi gặp phải một codebase mà không biết chắc là các phần được dùng ở những chỗ nào, tôi giờ cẩn thận hơn nhiều về việc dùng chung.

Cẩn thận với việc xài chung. Hãy kiểm tra ngữ cảnh. Sau đó mới tiếp tục.

Khi bạn thấy 2 đoạn code giống nhau, ngay lập tức bạn sẽ nghĩ tới việc tách chúng ra thành một function để có thể dùng chung. Hãy cân nhắc xem ngữ cảnh của chúng có giống nhau không. Liệu trong tương lai khi business của 1 trong 2 bên thay đổi thì phần dùng chung đó có còn được dùng chung nữa không, nếu không thì bạn nên xem lại.

Phần 8. The Boy Scout Rule

Giải thích: BSA (The Boy Scouts of America) là hội hướng đạo sinh tại Mỹ, thành lập với mục tiêu phát triển đức tính, đào tạo công dân, và thể chất cá nhân.

Hội BSA có một quy tắc: “Luôn làm cho nơi cắm trại sạch hơn lúc bạn đến”. Nếu bạn thấy rác trên đó, bạn dọn nó bất kể ai đã xả ra đó. Bạn có chủ đích cải thiện môi trường cho nhóm cắm trại tiếp theo. (Đúng ra câu ban đầu viết bởi Robert Stephenson Smyth Baden-Powell, cha đẻ của hướng đạo, viết là “Hãy làm cho thế giới này tốt hơn một chút so với lúc bạn thấy nó.”)

Nếu áp dụng quy tắc này cho code thì sẽ là: “Luôn làm cho một module sạch sẽ hơn khi bạn mở chúng ra”? Sẽ ra sao nếu như chúng ta luôn cố gắng nỗ lực ít nhiều cải thiện một module bất kể tác giả là ai? Kết quả sẽ như thế nào?

Tôi nghĩ nếu tất cả chúng ta đều tuân theo quy tắc này, chúng ta sẽ thấy hệ thống của chúng ta ngày một tốt lên. Chúng ta cũng sẽ thấy cả team quan tâm đến hệ thống thay vì chỉ vài cá nhân quan tâm đến phần code của họ.

Tôi không nghĩ quy tắc này là yêu cầu quá cao. Bạn không cần phải làm mọi module tốt lên trước khi bạn hoàn thành. Bạn chỉ cần làm cho chúng tốt hơn lên một chút so với lúc đầu. Tất nhiên nó có nghĩa là bất cứ đoạn code nào bạn thêm vào cũng phải sạch sẽ. Nó cũng có nghĩa bạn sẽ dọn dẹp một phần nhỏ khác trong module đó trước khi hoàn thành công việc. Ví dụ bạn chỉ cần cải thiện tên của một biến nào đó, hoặc tách function lớn ra thành 2 function nhỏ. Hoặc bạn gỡ bỏ một cái circular dependency nào đó, etc.

Thật ra mà nói, nghe có vẻ như là một phép tắc - giống như việc rửa tay sau khi đi vệ sinh, hoặc bỏ rác vào thùng chứ đừng ném ra nhà. Đúng là việc tạo ra một đoạn code ẩu giống với một việc không được xã hội chấp nhận như xả rác. Không nên làm vậy.

Nhưng còn hơn thế nữa. Quan tâm đến code của bạn là một việc. Quan tâm tới code của cả team là việc khác nữa. Team giúp lẫn nhau và cùng nhau dọn dẹp. Chúng ta tuân theo The Boy Scout Rule bởi nó tốt cho tất cả mọi người, không chỉ riêng ai.

Phần 9. Nhìn lại code của mình trước khi định đổ lỗi cho người khác

Tất cả developer chúng ta! - thường khó tin vào việc code của mình bị lỗi. Còn chuyện compiler bị lỗi thì khó có thể xảy ra, dù chỉ một lần.

Trong thực tế, rất hiếm gặp trường hợp code bị lỗi bởi bug của compiler, interpreter, OS, app server, database, memory manager (mấy từ này mình xin phép không dịch vì nó là từ chuyên ngành khá thông dụng), hoặc các phần khác của hệ thống phần mềm. Đúng, chúng có tồn tại, nhưng cực kì hiếm so với chúng ta tưởng.

Một lần tôi gặp một con bug thật sự của compiler khi nó tối ưu một biến của vòng lặp, nhưng từ trước đến giờ tôi hay nghĩ compiler hoặc OS có nhiều bug hơn. Tôi đã lãng phí rất nhiều thời gian và cuối cùng té ra đều là lỗi của mình, mỗi lần như vậy tôi cảm thấy hơi thốn một chút.
Với những tool mà được sử dụng rộng rãi, hoàn thiện, dùng trong nhiều hệ thống công nghệ, ít có lý do gì để nghi ngờ về chất lượng. Tất nhiên, nếu tool đó mới được ra lò, hoặc chỉ được dùng bởi vài người trên thế giới, hoặc chỉ có một vài lượt tải, phiên bản 0.1, mã nguồn mở (có vẻ tác giả muốn nói phần mềm mã nguồn mở dễ có bug hơn, make sense nếu như project đó không được maintain, fix bugs), thì có thể có lý do chính đáng để nghi ngờ phần mềm đó. (hoặc phiên bản alpha của một phần mềm trả phí cũng có thể bị nghi ngờ).

Do compiler bug là hiếm, bạn tốt hơn nên dùng thời gian và sức lực để tìm lỗi ở code của bạn hơn là cố gắng chứng tỏ compiler bị lỗi. Hãy thử các cách debug thông thường như: tách riêng các vấn đề, các hàm gọi, viết test cho chúng; kiểm tra cách gọi hàm, các thư viện dùng chung, số phiên bản; giải thích vấn đề cho người khác; kiểm tra stack corruption, các kiểu dữ liệu không khớp nhau; thử chạy code trên các máy tính khác nhau và các cấu hình build khác nhau, ví dụ như cấu hình cho debug hoặc release.

Tự vấn những giả thiết của mình và của người khác. Công cụ từ những nhà cung cấp khác nhau có thể sẽ có những giả thiết khác nhau - hoặc những công cụ khác nhau từ cùng nhà cung cấp cũng vậy.

Khi ai đó báo cáo về một vấn đề mà bạn không thể tái hiện lại nó, đi ra chỗ họ và xem họ làm cái gì. Họ có thể làm việc gì đó mà bạn chưa từng nghĩ tới hoặc làm theo trình tự khác.

Quy tắc cá nhân của tôi là nếu có bug mà không biết chính xác vì sao và lúc bắt đầu nghĩ đến lỗi của compiler thì tôi sẽ kiểm tra lại cái stack lúc lỗi. Điều này càng đúng nếu việc thêm code để dò lỗi làm cho vấn đề dịch chuyển.

Những lỗi liên quan đến đa luồng là những kiểu bug làm cho chúng ta tổn thọ và phải gào thét rất nhiều. Tất cả những khuyến khích cho việc viết code đơn giản được nhân lên gấp bội khi hệ thống đa luồng. Không dựa được vào debug và unit test để tìm lỗi cho những kiểu lỗi như thế, do vậy việc thiết kế đơn giản là cấp thiết.

Do vậy, trước khi vội đổ lỗi cho compiler, hãy nhớ lời khuyên của Sherlocklock Holme: “Khi bạn đã loại bỏ những điều không thể xảy ra thì điều cuối cùng, dù khó tin đến đâu, chắc chắn là sự thật.”, và của Dirk Gently “Khi bạn đã loại bỏ những điều khó tin thì điều cuối cùng, dù có vẻ bất khả thi, cũng phải là sự thật.”

Phần 10. Chọn tool một cách cẩn thận

CÁC ỨNG DỤNG HIỆN NAY THƯỜNG HIẾM KHI ĐƯỢC BUILD TỪ CON SỐ 0. Chúng được ráp lại từ những công cụ sẵn có - các component, các thư viện, framework - vì một số các lý do:

- Ứng dụng lớn lên theo kích thước, độ phức tạp, tinh xảo, trong khi thời gian phát triển chúng không tăng nhiều. Nếu người ta có thể tập trung vào viết nhiều hơn code phục vụ business, ít code hệ thống hơn thì sẽ tận dụng được trí não và thời gian của dev.
    - Những component và framework được sử dụng rộng rãi thường ít bug hơn “của nhà trồng”.
    - Sản xuất và bảo trì phần mềm là công việc đòi hỏi sức lực con người rất nhiều, do vậy mua thì có thể sẽ rẻ hơn tự làm.

Tuy nhiên, chọn các công cụ phù hợp cho ứng dụng của bạn có thể là một việc khó đòi hỏi chút tư duy. Thực tế thì khi đưa ra một lựa chọn, bạn nên lưu ý một số thứ:

- Các công cụ khác nhau có thể dựa vào những giả thiết khác nhau về context của chúng - ví dụ như hạ tầng, control mdoel, data model, cách thức giao tiếp,

v.v… - có thể dẫn đến sự không khớp nhau về kiến trúc giữa ứng dụng và công cụ. Những cái không khớp đó dẫn đến các đoạn code hack hoặc workaround mà sẽ làm cho code phức tạp hơn mức cần thiết.

- Các công cụ khác nhau có vòng đời khác nhau, và việc nâng cấp một trong số chúng có thể trở thành một công việc khó và tốn rất nhiều thời gian vì các chức năng mới, các thay đổi trong thiết kế, hoặc thậm chí các bug fix có thể dẫn đến sự không tương thích với các công cụ khác. Càng nhiều công cụ thì vấn đề càng trở nên khó khăn.
- Một số công cụ thì đòi hỏi phải có cấu hình riêng, thường thì là một vài file XML, cái mà có thể sẽ vượt khỏi tầm kiểm soát nhanh chóng. Cuối cùng ứng dụng trông sẽ như là được viết bằng XML kèm theo một vài dòng code ở một ngôn ngữ nào đó. Tính phức tạp của cấu hình sẽ làm cho ứng dụng khó bảo trì và mở rộng.
- Vendor lock-in (bị phụ thuộc vào code của bên thứ 3) xuất hiện khi code phụ thuộc vào các sản phẩm của các bên thứ 3, dẫn đến việc bị ràng buộc bởi chúng ở vài điểm: tính dễ bảo trì, hiệu năng, khả năng phát triển, giá, v.v…
- Nếu bạn có kế hoạch dùng phần mềm miễn phí, bạn sẽ có thể phát hiện ra là chúng không hoàn toàn miễn phí. Bạn có thể sẽ phải mua để được hỗ trợ, chưa chắc nó đã rẻ.
- Các điều khoản về license quan trọng, kể cả với phần mềm miễn phí. Ví dụ, trong vài công ty phần mềm, họ không được sử dụng các phần mềm có license GNU bởi vì tính tự nhiên vốn có của nó - phần mềm phát triển dùng nó phải được phân phối cùng với mã nguồn của nó.

Cách mà tôi tránh các vấn đề trên là bắt đầu từ nhỏ bằng việc sử dụng các công cụ thực sự cần thiết. Thường thì mục tiêu ban đầu sẽ là loại bỏ việc phải động đến các đoạn code hạ tầng low-level. Sau đó thêm vào sau nếu cần. Tôi cũng có xu hướng tách riêng những công cụ bên ngoài với các object của business domain bằng việc sử dụng interface, chia lớp, để mà sau này tôi có thể ít tốn công thay thế các tool này nhất. Một tác dụng phụ tích cực của việc này là tôi thường có ứng dụng gọn hơn và ít dùng các tool bên ngoài hơn dự đoán ban đầu.

Phần 11: Thông thạo lĩnh vực của bạn

Hãy xem qua đoạn code này:

if (portfolioIdsByTraderId.get(trader.getId()).containsKey(portfolio.getId())) {…}

Bạn gãi đầu tự hỏi rằng công dụng của đoạn code gì? Có vẻ nó lấy ID từ trader, và dùng dữ liệu đó để danh mục mới, đồng thời kiểm tra ID trong dữ liệu ban đầu xem chúng có xuất hiện trong danh mục mới không. Bạn tìm sự định nghĩa của porfolioIdsByTraderId và thấy thứ này:

Map<int, Map<int, int>> portfolioIdsByTraderId;

Dần bạn cảm thấy hẳn phải việc gì liên quan đến việc liệu trader có truy cập vào một danh mục cụ thể nào đó. Và dĩ nhiên là bạn sẽ tìm thấy các đoạn tương tự, hoặc một đoạn code tương tự nào đó nhưng có đôi chút khác biệt, bất cứ khi nào một thứ gì đó quan tâm liệu trader có truy cập vào một danh mục cụ thể.

Và trong đoạn codebase khác bạn lướt qua thứ này:

If (trader.canView(portfolio)) {…}

Không băn khoăn, bạn không cần phải biết trader hoạt động như thế nào. Có lẽ có một trong những map của maps được giấu đâu đó bên trong. Tuy nhiên, đấy không phải là việc của bạn.
Và bây giờ bạn sẽ lựa chọn codebase nào để làm việc cùng?

Ngày xửa ngày xưa, chúng ta chỉ có một vài cấu trúc dữ liệu vô cùng đơn giản: bits, bytes và ký tự(thật sự chỉ là bytes nhưng chúng ta giả định chúng là những ký tự chữ cái và kí tự đặc biệt). Hệ thập phân có hơi khó khăn bởi vì chữ số hệ thập phân của chúng ta không thể làm việc tốt với hệ nhị phân của máy tính được, chính vì lý do đó mà chúng ta có nhiều kiểu dữ liệu số khác nhau với kích thước khác nhau. Thế rồi tiếp

tục đến với mảng và chuỗi ký tự(chỉ là một dạng khác của mảng). Thế rồi chúng ta có kiểu stack, quene, bảng băm(hashes), và danh sách liên kết và skip lists và còn nhiều nhiều các cấu trúc dữ liệu khác nữa mà chúng không xuất hiện trong thế giới thực. “Khoa học máy tính” chính là nỗ lực hết mình chuyển hoá thế giới thực thành những cấu trúc dữ liệu trừu tượng. Những guru thực thụ còn có thể nhớ cách thức mà họ đã hoàn thành nó.

Tiếp theo chúng ta có các kiểu dữ liệu do người dùng tự định nghĩa. OK, đây không hẳn là điều mới, nhưng nó đã thay đổi cuộc chơi theo một cách nào đó. Nếu lĩnh vực của bạn chứa nhiều khái niệm như trader, hồ sơ(portfolio), bạn có thể mô hình hóa nó thành một kiểu khác gọi là, có thể nói, Trader và Portfolio. Tuy nhiên, điều quan trọng ở đây chính là bạn có thể mô hình mối quan hệ giữa chúng bằng những kỹ thuật kĩ thuật chuyên môn.

Nếu bạn lập trình mà không tuân theo nguyên tắc chuyên môn thì bạn đang tạo ra một tacit(đọc: bí mật). Hãy hiểu rằng kiểu int này mang ý là một cách để định dạng trader, đồng thời kiểu int đó mang đến ba ý nghĩa định dạng danh mục.(tốt nhất là đừng nên nhầm lẫn chúng!). Và nếu bạn giới thiệu một khái niệm kinh doanh mới(“ một vài thương nhân (trader) không được phép xem một số danh mục – đó là phạm pháp) cùng với một đoạn mã thuật toán, nói rằng mối quan hệ giữa sự tồn tại của những chiếc chìa khóa trong bản đồ, bạn không làm một sự kiểm tra nào và tuân thủ những ý tưởng của người khác.

Người lập trình viên tiếp theo có thể không hiểu được điều này vậy thì tại sao chúng ta không khiến nó trở nên dễ hiểu hơn. Sử dụng một cú pháp để tìm kiếm một cú pháp khác thực hiện việc kiểm tra sự có mặt có chính xác hay không. Và làm sao để áp dụng để một người nào đó hướng dẫn rằng những luật lệ trong kinh doanh chính là ngăn ngừa sự rối loạn trong nhận thức?

Khiến cho những khái niệm trong ngành trở nên dễ hiểu trong code của bạn nghĩa là những lập trình viên khác có thể dễ dàng làm việc với code thay vì phải cố gắng chỉnh sửa thuật toán để thích với code của bạn theo cách mà họ hiểu về nó. Đồng thời nó cũng mang ý nghĩa khi mà dự án của bạn trở nên tiên tiến thì sự hiểu biết của bạn về

lĩnh vực trong ngành cũng gia tăng có nghĩa là bạn đang trong tư thế sẵn sàng nhất phát triển code.

Cùng với những gì tốt đẹp nhất, thay đổi là cần thiết nhưng những luật lệ chỉ xuất hiện ở một nơi duy nhất, và bạn có thể thay đổi nó không cần sự phụ thuộc chính là khôn ngoan.

Một lập trình viên mà chỉ đến làm việc cùng bạn trong vài tháng sẽ cảm ơn bạn. Và người đó có thể là chính bạn.

Phần 12: Code is design

Thử tưởng tượng rằng ngày mai bạn thức dậy và nhận ra mình đang theo học ngành xây dựng, một ngành đã làm nên bước đột phá của thế kỷ. Hàng triệu con robot giá rẻ, cực kì nhanh có thể chế tạo vật liệu từ không khí, chi phí gần như bằng không và có thể tự sửa chữa. Hơn thế nữa, chúng có thể đưa ra một kế hoạch chi tiết cho dự án cũng như tự xây dựng mà không cần sự can thiệp của con người, với chi phí là không đáng kể.

Chúng ta có thể tưởng tượng tác động của nó đến ngành xây dựng, nhưng điều gì sẽ xảy ra theo chiều ngược lại? Các kiến trúc sư và các nhà thiết kế sẽ thay đổi như thế nào khi chi phí xây dựng không đáng kể ? Ngày nay, mô hình mô phỏng bằng máy tính và mô hình vật lý được xây dựng và được kiểm tra nghiêm ngặt trước khi được đầu tư xây dựng. Liệu chúng ta có thấy phiền muộn khi chi phí cho việc xây dựng về cơ bản là miễn phí? Nếu như 1 thiết kế sụp đổ, đó sẽ chẳng là vấn đề gì lớn – khi đó tìm hiểu điều gì sai và những con robot ma thuật của chúng ta sẽ xây dựng thêm một cái khác. Với những mẫu lỗi thời, hay chưa được hoàn thiện sẽ được phát triển bằng cách liên tục xây dựng và cải thiện dựa trên sự gần đúng của thiết kế hoàn chỉnh. Một người bình thường có thể sẽ gặp khó khăn để phân biệt một thiết kế dở dang và một thiết kế hoàn chỉnh.

Và rồi, khả năng dự đoán thời gian của chúng ta sẽ dần biết mất. Chi phí xây dựng sẽ dễ dàng tính toán hơn so với chi phí thiết kế - chúng ta biết chi phí xấp xỉ của việc lắp đặt dầm, và chúng ta cần phải lắp bao nhiêu cái dầm. Khi việc dự đoán dường như là chẳng còn cần thiết nữa thì những công việc thiết kế ít tính dự đoán sẽ dần chiếm ưu thế. Kết quả được tạo ra nhanh hơn, nhưng sự đáng tin cậy của những kết quả đó cũng mất dần đi theo thời gian.

Tất nhiên, vẫn có áp lực của nền kinh tế cạnh tranh. Với việc loại bỏ các chi phí xây dựng, một công ty có thể nhanh chóng hoàn thành một thiết kế đạt được từ lợi thế trên thị trường. Các thiết kế với tiến độ nhanh sẽ trở thành động lực chính của các công ty kỹ thuật. Chắc chắn với những người không có hiểu biết sâu về thiết kế sẽ thấy một phiên bản vô giá trị, chỉ thấy lợi thế của việc phát hành sớm và nói: “Nhìn nó có vẻ đủ tốt đấy.”.

Một số các dự án sinh tử sẽ siêng năng hơn, tuy nhiên trong nhiều trường hợp, khách hàng sẽ học cách chịu đựng bằng cách sống chung các thiết kế không hoàn chỉnh. Các công ty luôn có thể đưa các con robot ma thuật để “sửa chữa” các toà nhà và các phương tiện có lỗi mà họ đã bán. Tất cả những điều này chỉ ra một kết luận trái ngược đáng kinh ngạc rằng: tiền đề duy nhất của chúng ta là giảm đáng kể chi phí xây dựng, với kết quả có chất lượng tệ hơn.

Chúng ta không nên ngạc nhiên vì câu chuyện này trước đó đã diễn ra trong phần mềm. Nếu chúng ta chấp nhận code là thiết kế - một quy trình sáng tạo chứ không phải là quy trình cơ học – thì khủng hoảng phần mềm đã được giải thích. Chúng ta bây giờ có một cuộc khủng hoảng về thiết kế: nhu cầu về những thiết kế chất lượng, được xác nhận đã vượt quá khả năng của chúng ta để tạo ra chúng. Áp lực này khiến việc sử dụng thiết kế không hoàn chỉnh là khá lớn.

May thay, mô hình này cũng có những thứ giúp cho chúng ta có thể trở nên tốt hơn. Mô phỏng vật lý giống như những thử nghiệm tự động; thiết kế phần mềm sẽ không thể hoàn hảo cho đến khi nó vượt qua được các thử nghiệm. Để khiến cho các thử nghiệm hiệu quả hơn, chúng ta đang tìm cách để kiềm chế trong không gian trạng thái khổng lồ của các hệ thống lớn. Cải thiện ngôn ngữ và thực hành thiết kế cho chúng ta niềm hy

vọng. Cuối cùng, có một thực tế không thể chối cãi là: những thiết kế tuyệt vời được tạo ra bởi những nhà thiết kế vĩ đại làm chủ được công việc của họ. Với code cũng vậy.

Phần 13: Các vấn đề về cấu trúc code

Rất lâu trước đây, tôi đã làm việc trên một hệ thống Cobol nơi nhân viên không được phép thay đổi thụt đầu dòng trừ khi họ có lý do để thay đổi code, bởi vì ai đó đã từng phá vỡ một vài thứ bằng cách để một dòng trượt vào một trong những cột đặc biệt ở đầu dòng. Điều này bị áp dụng ngay khi bố cục sai lệch, đôi khi nó là như vậy, vì vậy chúng tôi phải đọc code rất cẩn thận vì chúng tôi không thể tin tưởng được. Các chính sách phải có chi phí rất lớn trong việc lôi kéo lập trình viên.

Có nghiên cứu cho rằng tất cả chúng ta dành nhiều thời gian lập trình để điều hướng và đọc code - tìm nơi để thực hiện thay đổi - hơn là thực sự gõ, vì vậy đây là điều chúng tôi muốn tối ưu hóa.

- Dễ nhìn. Mọi người rất giỏi trong việc kết hợp các mô hình trực quan(một phần còn sót lại từ hồi chúng ta phải nhận ra con sư tử trên thảo nguyên). Vì vậy tôi có thể tự giúp bản thân mình bằng cách làm cho mọi thứ không được liên quan trực tiếp đến đường dẫn, làm cho tất cả "sự phức tạp ngẫu nhiên" đi kèm với hầu hết các ngôn ngữ thương mại, mờ dần đi bằng cách chuẩn hóa nó. Nếu code hoạt động giống nhau mà nhìn giống nhau, thì hệ thống nhận thức của tôi sẽ giúp tôi chọn ra sự khác biệt. Đó là lý do tại sao tôi cũng quan sát các quy ước về cách bố trí các phần của một lớp trong một đơn vị biên dịch: hằng, trường, phương thức public, phương thức private.
- Bố cục có hàm ý. Tất cả chúng ta đã học cách dành thời gian để đặt tên để code của chúng ta thể hiện rõ ràng nhất có thể những gì nó làm, thay vì chỉ liệt kê các bước - phải không? Bố cục của code cũng là một phần của tính có hàm ý này. Việc cắt giảm đầu tiên là để nhóm đồng ý về một trình định dạng tự động cho những điều cơ bản, sau đó tôi có thể điều chỉnh bằng tay trong khi tôi đang viết code. Trừ khi có sự bất đồng chính kiến, một nhóm sẽ nhanh chóng hội tụ theo kiểu "hoàn thành thủ công" chung. Một định dạng không thể hiểu được ý định của tôi (tôi nên biết, tôi đã từng viết một lần) và điều quan trọng hơn với tôi là

việc ngắt dòng và nhóm các phản ánh mục đích của code lại, không chỉ là cú pháp của ngôn ngữ. (Kevin McGuire đã giải thoát cho tôi khỏi sự trói buộc với các trình định dạng code tự động.)

- Quy ước định dạng. Càng nhìn vào màn hình nhiều, tôi càng có thể nhận thấy mà không phá vỡ ngữ cảnh bằng cách cuộn hoặc chuyển đổi tập tin, điều đó có nghĩa là tôi có thể giữ trạng thái ít hơn trong đầu. Nhận xét thủ tục dài và nhiều khoảng trắng có ý nghĩa đối với tên 8 ký tự và in dòng, nhưng bây giờ tôi sống trong một IDE có tô màu cú pháp và liên kết chéo. Điểm ảnh là yếu tố giới hạn của tôi vì vậy tôi muốn mọi người đóng góp theo hướng hiểu biết của tôi về code. Tôi muốn bố trí để giúp tôi hiểu code, nhưng không nhiều hơn thế.

Một người bạn không phải lập trình viên đã từng nhận xét rằng code trông giống như thơ vậy. Tôi có được cảm giác đó từ code thực sự tốt, rằng mọi thứ trong văn bản đều có mục đích và nó ở đó để giúp tôi hiểu ý tưởng. Thật không may, viết code không có hình ảnh lãng mạn giống như viết thơ.

Phần 14: Review code

Vì sao bạn nên làm điều này? Review code giúp code sạch và đẹp hơn. Tuy nhiên các lập trình viên thường bỏ qua việc này vì cho rằng nó không quá cần thiết.

Có thể trước đây nhiều lập trình viên có trải nghiệm không tốt với việc review code nên họ thường có xu hướng không hào hứng với việc này. Các công ty, tổ chức sẽ yêu cầu tất cả code trước khi được deploy thực tế phải vượt qua được buổi review chính thức. Thường thì những buổi review sẽ là technical architect hoặc lead developer thực hiện và sẽ review tất cả mọi thứ. Vì đây là một công đoạn trong quy trình phát triển phần mềm nên mọi lập trình viên sẽ phải tuân thủ theo hướng dẫn có sẵn. Một số công ty, tổ chức có quy trình cứng nhắc hơn nhưng hầu hết thì không, vì cách tiếp cận như vậy thường sẽ phản tác dụng. Những người được review sẽ thấy mình như đang bị đánh giá bởi một hội đồng đặc xá. Họ cần thời gian để đọc cũng như cập nhật hệ thống nếu không họ sẽ bị mắc kẹt trong dự án của mình và tiến trình thực hiện dự án sẽ bị đình trệ.

Ngoài đơn giản chỉ là sửa lỗi code, mục đích của một buổi review code là chia sẻ kinh nghiệm và đưa ra các hướng code chung. Chia sẻ code của bạn với những người cho phép dùng những đoạn code chung. Hãy để một người bất kỳ trong nhóm xem qua code của mọi người. Đừng chỉ tìm lỗi trong code của họ, bạn nên đánh giá bằng cách cố gắng học và hiểu nó.

Giữ tâm thế nhẹ nhàng trong một buổi review, đảm bảo rằng những đóng góp của bạn có tính chất xây dựng và thân thiện chứ không phải chê bai một cách tiêu cực. Nên có những vai trò khác nhau trong buổi review, tránh việc kinh nghiệm làm việc khác nhau giữa các thành viên trong nhóm ảnh hưởng tới buổi review. Chẳng hạn chúng ta có thể chia ra những vai trò như sau: Một người đảm nhiệm việc tập trung review vào document, một người tập trung vào những ngoại lệ có thể xảy ra, một người khác thì tập trung vào các chức năng của dự án,...Điều này giúp ích cho mọi người trong nhóm đều có thể tham gia đánh giá cũng như chia đều gánh nặng cho cả nhóm chứ không phải chỉ một người làm hết mọi thứ.

Đều đặn mỗi tuần nên có một ngày hoặc có thể là vài giờ trong những buổi họp để review code. Luân phiên người được review code trong các buổi họp và đừng quên chuyển đổi vai trò của những người review cho nhau. Hãy khuyến khích những người mới trong những buổi review, họ có thể chưa có nhiều kinh nghiệm, tuy nhiên những kiến thức ở đại học có thể mang đến một góc nhìn khác. Mời các chuyên gia có kinh nghiệm để họ có thể giúp xác định những dòng code dễ gây ra lỗi nhanh và chính xác hơn. Kiểm tra coding conventions bằng các công cụ kiểm thử cũng sẽ giúp buổi review trở nên suôn sẻ và dễ dàng hơn, vì những dòng code sau khi được format bằng các công cụ này sẽ không cần phải đem ra thảo luận nữa.

Việc tạo một không khí vui vẻ trong những buổi review cũng rất quan trọng vì dù sao những người review cũng không phải những cỗ máy. Chắc chắn là sẽ chẳng ai thấy hào hứng nếu tham gia một buổi review nặng nề và buồn tẻ cả. Chia sẻ kinh nghiệm với nhau giữa các thành viên trong những buổi review không chính thức, thay vì mỉa mai code của nhau hãy mang theo hoa quả bánh kẹo và thân thiện với mọi người.

Phần 15: Coding với lý luận

Cố gắng lý luận về tính chính xác của phần mềm bằng tay dẫn đến một bằng chứng chính thức còn dài hơn cả code và có khả năng chứa nhiều lỗi hơn. Công cụ tự động thích hợp, nhưng không phải lúc nào cũng khả thi. Những điều sau đây mô tả một vấn đề: lý luận bán chính thức về tính đúng đắn.

Cách tiếp cận cơ bản là chia tất cả các code cần được xem xét thành các phần nhỏ- từ một dòng duy nhất, chẳng hạn một function call, cho đến những đoạn có ít hơn mười dòng- và tranh luận về tính chính xác của chúng. Cuộc tranh luận chỉ cần đủ mạnh để thuyết phục người bạn lập trình viên ủng- hộ- sự- bào- chữa- của- quỷ.

Một section nên được chọn sao cho tại mỗi điểm cuối, trạng thái của chương trình (cụ thể là bộ đếm chương trình và giá trị của các đối tượng) thỏa mãn thuộc tính được mô tả. Và chức năng của nó (chuyển đổi trạng thái) có thể dễ dàng mô tả một nhiệm vụ duy nhất. Những điều này sẽ giúp cho lý luận trở nên đơn giản hơn. Các thuộc tính như vậy khái quát các khái niệm như điều kiện tiên quyết và hậu điều kiện cho các hàm và bất biến cho các vòng lặp và class (với sự tôn trọng các trường hợp của chúng). Cố gắng để các section độc lập với nhau giúp đơn giản hóa lý luận và là điều không thể thiếu khi cần sửa đổi.
Nhiều thực hành coding nổi tiếng (mặc dù có lẽ ít được theo dõi) và được coi là "tốt" sẽ giúp cho việc suy luận trở nên dễ dàng hơn. Do đó, chỉ bằng cách có dự định về những lý luận cho code của bạn, bạn đã bắt đầu suy nghĩ về một phong cách và cấu trúc tốt hơn. Không có gì đáng ngạc nhiên, hầu hết các thực hành này có thể được kiểm tra bằng các máy phân tích code tĩnh:

- Tránh sử dụng các câu lệnh goto, vì chúng làm cho các section phụ thuộc lẫn nhau quá nhiều.
- Tránh sử dụng các biến toàn cục có thể sửa đổi, vì sẽ dẫn đến các section liên quan phụ thuộc vào chúng.
- Mỗi biến nên có phạm vi nhỏ nhất có thể. Ví dụ, một đối tượng cục bộ có thể được khai báo ngay trước khi sử dụng lần đầu.
- Làm cho các đối tượng bất biến bất cứ khi nào có liên quan.

- Làm cho code trở nên dễ đọc bằng cách sử dụng khoảng cách, cả ngang và dọc. Ví dụ: căn chỉnh các cấu trúc liên quan và sử dụng một dòng trống để phân biệt hai phần riêng.
- Làm cho code tự ghi lại bằng cách chọn tên mô tả (nhưng tương đối ngắn) cho các đối tượng, loại, hàm, v.v.
- Nếu bạn cần nested section, hãy cho nó một function.
- Làm cho các chức năng của bạn ngắn và tập trung vào một nhiệm vụ duy nhất. Áp dụng giới hạn 24 dòng. Mặc dù kích thước và độ phân giải màn hình đã thay đổi, nhưng không có gì thay đổi trong nhận thức của con người kể từ những năm 1960.
- Các hàm nên có ít tham số (bốn là giới hạn tốt nhất). Điều này không hạn chế dữ liệu được truyền đến các hàm: Gộp các tham số liên quan vào chung một đối tượng thu lợi từ các đối tượng bất biến và lý luận, chẳng hạn như sự gắn kết và tính nhất quán của chúng.
- Tổng quát hơn, mỗi đơn vị code, từ một đoạn đến cả thư viện, nên có một giao diện hẹp. Ít giao tiếp giảm lượng lý luận cần thiết. Điều này có nghĩa là các getter trả lại trạng thái nội bộ là một trách nhiệm pháp lý. Đừng yêu cầu đối tượng cung cấp thông tin để làm việc; thay vào đó, hãy yêu cầu đối tượng thực hiện công việc với thông tin đã có. Nói cách khác, đóng gói là tất cả về giao diện hẹp .
- Để duy trì các class bất biến, việc sử dụng các setter nên được khuyến khích, vì các setter có xu hướng cho phép các bất biến chi phối trạng thái của một đối tượng bị phá vỡ.

Cũng như lý luận về tính đúng đắn, tranh luận về code giúp bạn hiểu thêm về nó. Truyền đạt những hiểu biết bạn có vì lợi ích của mọi người.

Phần 16: Bàn về việc comment code

Trong lớp học lập trình đầu tiên của tôi ở đại học, giáo viên đã phát cho mỗi người hai tờ giấy lập trình (coding sheet) sau đó ông ấy viết bài tập lên bảng rồi rời khỏi phòng, bài tập như sau : " Viết một chương trình nhập vào 10 điểm của trò chơi bowling và tính điểm trung bình cộng". Bài toán này có thể khó tới mức nào? Tôi không nhớ cách giải

là gì, nhưng tôi chắc nó có sử dụng vòng lặp FOR/NEXT và tổng cộng không quá 15 dòng. Mỗi tờ Coding Sheet chỉ có thể chứa khoảng 70 dòng - Dành cho những ai đang đọc cái này, đúng vậy, chúng tôi đã từng viết code trên giấy rất lâu trước khi thực sự đưa chúng vào máy tính. Tôi đã rất bối rối tại sao giáo viên lại đưa cho chúng tôi 2 tờ trong khi thuật toán chưa tới 15 dòng code. Vì chữ viết của tôi không được đẹp nên tôi đã sử dụng tờ coding sheet còn lại để copy bài code ra một cách rõ ràng hơn với hy vọng có thêm một vài điểm cộng về phong cách.

Tôi đã rất ngạc nhiên khi nhận lại bài tập của mình vào đầu giờ của tiết học tiếp theo, điểm số của tôi chỉ vừa đủ để qua môn. (Nó đã trở thành điềm báo với tôi về khoảng thời gian còn lại ở đại học.) Nguệch ngoạc phía trên bài code mà tôi đã sao chép là dòng chữ " Không có comments ?" ("No comment?")

Việc giáo viên và tôi đều biết nhiệm vụ của chương trình là gì vẫn chưa đủ .Một phần, mục đích của bài tập này đã dạy tôi rằng code của tôi phải có thể " tự giải thích" với các lập trình viên khác chức năng của nó. Đó là một bài học mà tôi không bao giờ quên.

Comment không phải là xấu. Chúng cần thiết cho những bài lập trình như cấu trúc rẽ nhánh hoặc cấu trúc lặp. Hầu hết các ngôn ngữ hiện đại đều có công cụ gần giống với Javadoc có thể phân tích các comment đã được định dạng một cách chính xác để tự động xây dựng một thư mục API. Đây là một khởi đầu tốt, nhưng chưa đủ. Trong code của bạn nên có những giải thích về chức năng mà nó đang thực hiện. Theo như một câu ngạn ngữ cũ :" Nếu như nó khó để viết , thì cũng sẽ khó để đọc," Nếu bạn code như vậy thì bạn không những gây khó khăn cho khách hàng, cấp trên hay đồng nghiệp, mà còn gây không ít trở ngại cho bạn sau này.

Mặt khác, bạn không nên lạm dụng comment. Đảm bảo rằng comment làm rõ code của bạn chứ không phải làm cho nó trở nên khó hiểu. Hãy viết những comment thích hợp để giải thích mục đích của code là gì. Phần comment tiêu đề (header comment) bạn nên đưa ra đầy đủ thông tin để bất kỳ lập trình viên nào cũng đều có thể sử dụng đoạn code của bạn mà không cần đọc nó, trong khi đó, các inline comment nên hỗ trợ tốt các developer tiếp theo trong việc sửa chữa hoặc mở rộng code.

Trong một lần bàn bạc về công việc, tôi đã không đồng tình với một quyết định về bản thiết kế của cấp trên. Cũng như cách mà các lập trình viên nghiệp dư thường làm, tôi đã copy đoạn văn bản từ email hướng dẫn sử dụng bản thiết kế của họ vào phần header comment của file project của mình mặc dù trước đó đã tỏ ra sự bất đồng với cấp trên, điều đó khá là xấu hổ. Hóa ra ngay sau khi tôi bàn giao lại sản phẩm, thì các quản lý của shop đã review lại code, tôi nhận ra việc thêm các hướng dẫn đó vào phần header comment đã thực sự cứu tôi. Và đó cũng là lần đầu tiên tôi biết đến thuật ngữ career-limiting move (Hành động hoặc lỗi lầm gây trở ngại trong việc thăng tiến sự nghiệp).

Phần 17: Chỉ nên comment khi nào mà code không thể giải thích

Sự khác biệt giữa lý thuyết và thực tiễn trong thực tế lớn hơn rất nhiều so với trên lý thuyết – theo như quan sát thì điều này chắc chắn áp dụng với mọi comments . Trên lý thuyết, ý tưởng cơ bản của comment code nghe có vẻ hữu ích, như: cung cấp chi tiết hơn cho người đọc, giải thích về cái gì đang diễn ra. Còn điều gì có thể hữu ích hơn những điều này cơ chứ? Tuy nhiên, trong thực tế comment đôi khi lại trở thành một điều không tốt. Cũng như với bất kì hình thức viết nào khác, có một loại kỹ năng để viết tốt comments. Phần lớn của kỹ năng đó là biết khi nào không nên viết chúng.

Khi code bị sai cú pháp, trình biên dịch, trình thông dịch và những tool khác chắc chắn sẽ báo lỗi. Nếu đoạn code theo một cách nào đó không chính xác về mặt chức năng, thì việc review, static analysis, testing, và sử dụng trong công việc hằng ngày sẽ văng hết các lỗi ra. Nhưng còn comments thì sao? Trong cuốn **The Elements of Programming Style (Computing McGraw-Hill)** của tác giả Kernighan và Plauger có viết rằng “Một comments sẽ có giá trị bằng 0 (thậm chí là số âm) nếu nó sai.” Và những bình luận như thế thường tạo thành rác và tồn tại bên trong codebase theo cách mà các lỗi mã hóa không bao giờ xảy ra. Chúng tạo cho chúng ta một nguồn gây xao lãng liên tục và cả thông tin sai lệch.

Nói thế nào khi mà những comment không sai về mặt kỹ thuật, nhưng lại không thêm chút giá trị nào cho đoạn code? Những comments như thế thực sự là vô dụng và thừa thãi. Comment lặp lại đoạn code nó chẳng cung cấp thêm thông tin gì cho người đọc đâu – tức là nói điều gì đó một lần bằng code và sau đó lặp lại bằng ngôn ngữ tự nhiên không làm nó đúng hơn hoặc chân thực hơn. Comment lặp lại như kiểu con vẹt không phải là một đoạn mã thực thi, nó không có tác dụng hữu ích cho người đọc hoặc thời gian chạy. Nó cũng trở nên cũ rất nhanh. Kể cả những comments liên quan đến phiên bản và những comments ghi chú cho code để cố gắng giải quyết các câu hỏi về phiên bản và lịch sử cập nhật. Comment làm gì khi mà những câu hỏi này đã được trả lời (hiệu quả hơn rất nhiều) bởi các công cụ quản lí phiên bản rồi.

Sự xuất hiện ngày càng nhiều của những comment mang tính dư thừa hoặc những comments mang thông tin sai lệch bên trong codebase đã dần làm cho các lập trình viên làm ngơ toàn bộ các comments, hoặc bỏ qua chúng, kể cả thực hiện các biện pháp tích cực nhằm che giấu chúng. Các lập trình viên họ rất sáng tạo và sẽ làm mọi cách để bỏ đi những gì họ cho rằng có thể sẽ gây ảnh hưởng đến hiệu suất công việc, ví dụ như: thu các comments lại; thay đổi lại màu sắc để các comments và background có màu giống nhau; dùng script để lọc bỏ các comments, vân vân và mây mây... mục đích là để “cứu” codebase từ những việc không nên làm của các lập trình viên gà mờ. Vì thế để giảm bớt khả năng bỏ lỡ những comments mang lại giá trị thực sự, comment nên được xem như thể chúng là code. Bằng cách mỗi comment nên thêm vào một vài giá trị cho người đọc, nếu không thì chúng chẳng khác gì là rác rưởi và đã là rác rưởi thì nên vứt đi hoặc là viết lại mà thôi.

Nói tóm lại, khi comment cần hạn chế những điều gì? Comments nên nói những điều mà code không nói ra được hoặc không thể nói. Một comment giải thích ý nghĩa của một đoạn code để sẵn sàng cho việc thay đổi cấu trúc đoạn code hoặc lập trình theo những quy ước chung để code có thể tự giải thích nghĩa cho chính nó. Thay vì dùng để giải thích cho những method hay class đặt tên dở, thì hãy đổi tên của chúng đi. Và thay vì comment từng phần từng phần trong những function dài ơi là dài thì hãy tách chúng ra thành những function nhỏ hơn sao cho tên của chúng vẫn giữ được những mục đích cũ. Hãy cố gắng diễn đạt càng nhiều càng tốt thông qua các đoạn code. Nếu có bất kỳ sự thiếu sót nào giữa việc diễn tả code hay diễn tả những nội dung mang tính bao quát

tổng thể thì đây là lúc chúng ta nên comment. Comment những gì mà code không thể nói ra, nó không đơn giản là những gì code không nói được.

Phần 18: Học hỏi không ngừng

Chúng ta đang sống trong những khoảng thời gian thú vị. Khi sự phát triển được phân phối trên toàn cầu, bạn học được rằng có rất nhiều người có khả năng làm công việc của bạn. Bạn cần tiếp tục học hỏi để ở lại với thị trường. Nếu không bạn sẽ trở thành một con khủng long bị mắc kẹt. Cho đến một ngày, trong cùng một công việc, bạn sẽ không còn cần thiết nữa hay công việc của bạn sẽ được chuyển giao cho một số nguồn rẻ hơn.
Vậy bạn nên làm gì với nó? Một số nhà tuyển dụng đủ hào phóng để cung cấp, đào tạo để mở rộng bộ kỹ năng của bạn. Những người khác có thể không có thời gian rảnh rỗi hay tiền bạc cho bất kì đào tạo nào cả. Vậy để thận trọng, bạn nên có trách nhiệm hơn với việc học tập của mình.

Đây là các cách để bạn có thể tiếp tục học tập. Nhiều trong số này có thể tìm thấy trên Internet:

- Đọc sách, tạp chí, blog và trên các websites. Nếu bạn muốn đi sâu hơn vào môn một chủ đề, hãy xem xét gia nhập một nhóm cùng đam mê.
- Nếu bạn thực sự muốn được đắm mình vào một công nghệ, hãy bắt tay vào viết một số code.
- Luôn luôn cố gắng làm việc với người cố vấn, vì là người đứng đầu có thể cản trở giáo dục của bạn. Mặc dù bạn có thể học được một số điều từ bất kì ai, nhưng bạn có thể học hỏi nhiều hơn từ một người thông minh hoặc có kinh nghiệm hơn. Nếu bạn không thể tìm kiếm một người cố vấn, hãy cân nhắc việc tiếp tục.
- Sử dụng những cố vấn ảo, Tìm kiếm tác giả và những nhà phát triển trên mạng, những người mà bạn thật sự thích và đọc mọi thứ họ viết. Hãy theo dõi blog của họ.
- Nhận biết những frameworks và các thư viện bạn sử dụng. Tìm hiểu cách thức hoạt động của nó để bạn có thể sử dụng nó tốt hơn. Nếu họ chia sẻ mã nguồn,

bạn thật sự may mắn. Sử dụng các debugger chuyển qua code để xem những gì đang diễn ra.

- Bất cứ khi nào bạn mắc lỗi, sửa lỗi hoặc gặp sự cố, hãy cố gắng thật sự hiểu những gì đang diễn ra. Cũng có khả năng là có người khác cũng gặp lỗi đó và đã đăng nó lên mạng tìm sự giúp đỡ. Vậy nên google thật sự hữu ích lúc này.
- Cách tốt nhất để học một cái gì đó là dạy và nói về nó. Khi người khác lắng nghe và đặt câu hỏi cho bạn, bạn sẽ cảm thấy có động lực hơn để học. Hãy thử ăn trưa và học khi làm việc, kết bạn với những người có cùng mục tiêu hoặc đến các hội thảo liên quan.
- Tham gia hoặc bắt đầu một nhóm nghiên cứu(cộng đồng mẫu) về ngôn ngữ, công nghệ mà bạn quan tâm.
- Đi đến các hội thảo. Hoặc nếu bạn không thể đi thì có rất nhiều hội thảo được phát trực tuyến miễn phí.
- Đi lại có thể làm cho một vài thứ dễ dàng, nhẹ nhàng hơn.
- Hiểu những phân tích và cảnh báo khi sử dụng các static analysis tools trong IDE.
- Làm theo lời khuyên của các lập trình viên thực dụng và mỗi năm hãy học một ngôn ngữ mới. Học ít nhất một công nghệ hoặc công cụ mới. Sự chia nhỏ ra cung cấp cho bạn những ý tưởng có thể sử dụng trong kho công nghệ hiện tại của mình.
- Không phải tất cả những gì bạn học phải là về công nghệ. Tìm hiểu lĩnh vực bạn đang làm việc để hiểu rõ hơn các yêu cầu và giúp đỡ giải quyết vấn đề kinh doanh. Học cách làm việc hiệu quả hơn - làm thế nào để làm việc tốt hơn là một sự lựa chọn tốt.
- Quay trở lại trường học.

Sẽ thật tuyệt khi có khả năng mà Neo có trong The Matrix, chúng ta sẽ có nhiều thời gian hơn cho bản thân mình không dành quá nhiều cho công việc và học tập.
Công nghệ thay đổi nhanh chóng. Đừng để bị tụt lại phía sau.

Phần 19: Convenience Is not an -ility

Nhiều điều đã được đề cập khi nói về tầm quan trọng và thách thức của việc thiết kế API tốt. Thật khó để thành công ngay từ lần đầu và thậm chí còn khó khăn hơn để thay đổi sau này. Đại khái nó giống như việc nuôi dạy một đứa trẻ. Hầu hết các lập trình viên có kinh nghiệm đã học được rằng một API tốt tuân theo mức độ trừu tượng nhất định, thể hiện tính nhất quán, tính đối xứng và hình thành từ vựng cho một ngôn ngữ. Than ôi, nhận thức được các nguyên tắc hướng dẫn không tự động chuyển thành hành vi thích hợp. Ăn nhiều đồ ngọt không tốt cho bạn.

Thay vì thuyết giảng, tôi muốn chọn một chiến lược thiết kế API cụ thể, một chiến lược mà tôi đã gặp nhiều lần: tranh luận về sự thuận tiện. Nó thường bắt đầu bằng một trong những suy nghĩ sau:

- Tôi không muốn các class khác phải thực hiện hai call riêng biệt chỉ để thực hiện một việc.
- Tại sao tôi phải tạo thêm một phương thức khác nếu nó gần giống với phương thức này? Tôi chỉ cần thêm một lệnh switch đơn giản.
- Nhìn xem, rất dễ: Nếu tham số chuỗi thứ hai kết thúc bằng ".txt", phương thức sẽ tự động giả định rằng tham số đầu tiên là tên tệp, vì vậy tôi thực sự không cần đến hai phương thức.

Mặc dù có chủ đích tốt, các đối số như vậy có xu hướng làm giảm khả năng đọc code sử dụng API. Một phương thức như

parser.processNodes(text, false);

hầu như vô nghĩa nếu như không biết cách thực hiện hay tham khảo tài liệu. Phương thức này có thể được thiết kế để thuận tiện cho người triển khai, không phải cho caller. "Tôi không muốn caller phải thực hiện hai call riêng biệt" được dịch thành "Tôi không muốn mã hóa hai phương thức riêng biệt". Về cơ bản tiện lợi không có gì sai nếu đó là liều thuốc giải cho sự tẻ nhạt, vụng về hoặc phiền phức. Tuy nhiên, nếu chúng ta suy nghĩ kỹ hơn một chút, thuốc giải cho những triệu chứng trên là sự hiệu quả, tính nhất

quán và sự thanh lịch, chứ không nhất thiết phải là tiện lợi. Các API được cho là sẽ che giấu sự phức tạp tiềm ẩn, vì vậy thiết kế API tốt sẽ đòi hỏi nhiều cố gắng.

Phép ẩn dụ coi API như một ngôn ngữ có thể hướng chúng ta tới các quyết định tốt hơn trong những tình huống tương tự. Một API nên cung cấp ngôn ngữ, cung cấp cho class tiếp theo lượng từ vựng đủ để hỏi và trả lời những câu hỏi hữu ích. Điều này không có nghĩa là nó sẽ cung cấp câu trả lời chính xác cho mỗi câu hỏi. Vốn từ vựng đa dạng cho phép chúng ta thể hiện sự tinh tế trong ngữ nghĩa. Ví dụ: chúng tôi muốn nói chạy thay vì đi bộ, mặc dù về cơ bản nó giống nhau, chỉ khác ở tốc độ. Vốn từ vựng API nhất quán và được suy nghĩ kỹ sẽ giúp code trở nên cảm tính hơn và dễ hiểu hơn trong layer tiếp theo. Quan trọng hơn, vốn từ tổng hợp cho phép lập trình viên sử dụng API theo những cách mà bạn không hề nghĩ tới- thực sự rất tiện lợi cho người dùng API! Lần tới khi bạn muốn gộp một vài thứ lại với nhau thành một phương thức API, hãy nhớ rằng tiếng Anh không có một từ mang nghĩa MakeUpYourRoomBeQuietAndDoYourHomeWork, mặc dù nghe có vẻ thuận tiện cho hoạt động thường xuyên được yêu cầu như vậy.

Phần 20: Deploy sớm và thường xuyên

Việc debug các quá trình sử dụng thực tế và cài đặt thường bị trì hoãn cho đến khi gần kết thúc dự án. Trong một số dự án, việc viết một công cụ cài đặt được giao cho Released Engineer (*) người dù không muốn nhưng vẫn phải đảm nhận công việc này. Sự đánh giá và minh họa phải được làm thủ công để đảm bảo rằng mọi thứ đều hoạt động. Kết quả là nhóm không có kinh nghiệm về việc triển khai sử dụng thực tế lại làm nó cho đến khi quá muộn để thay đổi tình hình.

Quá trình cài đặt hay triển khai sử dụng là điều đầu tiên mà khách hàng đánh giá hoặc đơn giản là bước đầu tiền để có một sản phẩm đáng tin (hoặc ít nhất là dễ debug). Việc triển khai sử dụng phần mềm chính là những gì khách hàng sẽ trải nghiệm. Với việc không triển khai sử dụng phần mềm tốt, bạn sẽ khiến khách hàng cảm thấy nghi ngờ trước khi họ sử dụng sản phẩm của bạn.

Khởi đầu dự án với quy trình cài đặt sẽ cho bạn thời gian để phát triển quy trình đó khi bắt đầu phát triển sản phẩm và thay đổi code cho việc cài đặt sẽ dễ dàng hơn. Chạy và kiểm tra trên một “môi trường sạch” thường xuyên cũng giúp bạn kiểm tra code của mình dựa trên môi trường phát triển hay thử nghiệm. (development or test environments)
Để việc triển khai sử dụng đến cuối cùng đồng nghĩa với việc cần phức tạp hơn để giải quyết các vấn đề trong code của bạn. Nghe có vẻ tuyệt vời trong một IDE, nơi bạn có toàn quyền kiểm soát, có thể làm cho quá trình triển khai sử dụng phức tạp hơn nhiều. Tốt nhất là nên suy nghĩ đến sự đánh đổi này sớm hơn.

Mặc dù việc “triển khai sử dụng thực tế” dường như không có nhiều giá trị kinh tế ngay trước mắt so với việc thấy một sản phẩm chạy trên máy tính của developer, nhưng sự thật là cho đến khi bạn chứng minh sản phẩm của bạn trên một môi trường cụ thể, có rất nhiều việc phải làm để nhận được giá trị kinh tế đó. Nếu lý do bạn đưa ra để trình hoãn việc triển khai sử dụng là vì nó không quan trọng thì hãy làm điều đó vì nó có chi phí thấp. Nếu điều đó quá phức tạp hoặc có nhiều điều chưa chắc chắn, hãy làm những gì bạn sẽ làm với code của mình: thử nghiệm, đánh giá và cấu trúc lại quá trình triển khai sử dụng của bạn.

Quá trình cài đặt hay triển khai sử dụng là điều cần thiết cho chất lượng sản phẩm hoặc sự chuyên nghiệp trong dịch vụ mà khách hàng nhắm đến, vì thế bạn nên kiểm tra và tái cấu trúc lại quá trình này. Chúng tôi kiểm tra và cấu trúc lại code trong suốt dự án và việc triển khai sử dụng cũng cần được thực hiện như vậy.

Giải nghĩa:
(*) - *Kỹ sư chịu trách nhiệm việc compile, kết hợp và đưa vào sản phẩm cuối cùng.*

Phần 21: Phân biệt Business exception và Technical exception

(Exception không phải là lỗi, nó là những vấn đề xảy ra một cách ngẫu nhiên không được trông đợi trước, ví dụ như khách hàng cố gắng chuyển tiền trong khi không còn đồng nào trong tài khoản ngân hàng)

Có hai lý do cơ bản dẫn đến việc bị exception khi đang runtime (runtime là từ chỉ khoảng thời than chương trình đang biên dịch): đó là lý do kĩ thuật xuất phát từ những dòng code trong ứng dụng của bạn, thường những lỗi này sẽ khiến chương trình bị đứng và không chạy nữa, và lý do về business logic thường xuất phát từ phía người dùng như là dùng sai các chức năng trong ứng dụng. Và trong hầu hết các ngôn ngữ hiện đại ngày nay như là LISP, Java, Smalltalk và C#, những ngôn ngữ này dùng exception để thông báo khi có lỗi xảy ra từ hai vấn đề trên. Tuy nhiên, bởi vì hai lý do trên rất khác biệt với nhau, vì vậy chúng ta nên xử lý thật cẩn thận từng vấn đề riêng lẽ với nhau.
Như các bạn biết, một thông báo lỗi sẽ xuất hiện khi chương trình phát hiện một lỗi nào đó khi đang biên dịch. Ví dụ như, nếu bạn truy cập phần tử thứ 83 từ một mảng chỉ có 17 phần từ, khi đó chắc chắc chương trình sẽ dừng lại và một vài exception sẽ được thông báo ra màn hình. Đôi khi, bạn sử dụng thư viện bên ngoài và chúng sẽ gọi những đối số nghịch nhau từ chính những thư viện ấy, và đó là nguyên nhân xảy ra lỗi ngay bên trong thư viện bạn gọi vào.

Và đừng bao giờ cố tự mình giải quyết tất cả những exception này. Thay vào đó chúng ta sẽ để những exception này ở những mức thông báo cao nhất và hãy chắc rằng hệ thống vẫn ở trạng thái an toàn. Ví dụ như quay ngược lại một phiên làm việc, hay là gửi thông báo tới adminnistrator hoặc là gửi báo cáo ngược lại cho user.

Một biến thể khác của trường hợp này được gọi là 'library situation', và một caller đã làm hỏng contract method (các bạn có thể tìm hiểu thêm với từ khóa 'contract programming') ví dụ như khiến cho chương trình gửi đi những argument lạ hoặc tồn tại

những đối tượng không được định nghĩa rõ ràng. Ví dụ sẽ có một đoạn text khi bạn cố truy cập phần tử thứ 83 từ mảng có 17 phần tử từ ví dụ trên như sau: 'the caller should have checked'. Rõ ràng đây không phải là một lỗi từ phía client, mà thường là chúng ta sẽ thrown ra một technical exception.

Một tình huống khác, nhưng vẫn thuộc về technical exception, tình huống khi mà một chương trình nào đó có nảy sinh vấn đề trong môi trường thực thi, ví dụ như database không phản hồi. Trong trường hợp này, bạn phải giả sử rằng các thành phần trong chương trình đã làm mọi thứ để xử lý exception này - ví dụ như repair lại kết nối giữa client và sever một số lần hữu hạn - và vẫn thất bại. Thậm chí có cả những nguyên nhân gây ra lỗi khác thì tình huống bạn mắc phải vẫn tương tự nhau: thật ra chúng ta không làm được gì nhiều về việc này. Vì thế, chúng ta gửi những vấn đề phát sinh này thông qua những exception và xử lý chúng.

Ngược lại, chúng ta có tình huống không thể hoàn thành một caller bởi những lý do về domain-logical. Trong trường hợp này chúng ta giải quyết những tình huống trên như một exception. Ví dụ, trông những lỗi này kì lạ và khá phiền phức nhưng không mang tính programming như error. Ví dụ như khi bạn rút nhiều tiền hơn số tiền trong tài khoản của bạn. Mặt khác kiểu trường hợp này chỉ là một phần trong contract, và việc thrown ra một exception cũng chỉ là một cách để cảnh báo người dùng. Với những trường hợp như trên thì thường chúng ta sẽ tạo ra những exception đặc biệt hoặc tạo ra những cấp bậc exception khác nhau và để người dùng có thể tự giải quyết các exception này.

Trộn cả hai technical exception và business exception một cách không rõ ràng và lộn xộn là cách mà method contract hoạt động. Có những điều kiện phải được đáp ứng để đảm bảo trước khi gọi (caller), và lường trước những tình huống exception mà có thể sẽ gặp. Phân chia ra những trường hợp như vậy sẽ khiến cho mọi thứ rõ ràng và tăng cơ hội những technical exception được xử lý bởi application framework, trong khi đó business exception sẽ được xử lý bên phía client.

Phần 22. Luyện tập có chủ đích

Luyện tập có chủ đích không phải chỉ đơn giản là thực hiện một tác vụ. Nếu bạn tự hỏi, “Tại sao tôi lại thực hiện tác vụ này?” và câu trả lời của bạn là, “Để hoàn thành tác vụ”, thì bạn đang không luyện tập có chủ đích. Bạn luyện tập có chủ đích để cải thiện khả năng thực hiện một tác vụ. Luyện tập có chủ đích bao gồm về mặt kỹ năng lẫn kỹ thuật. Luyện tập có chủ đích có nghĩa là sự lặp lại. Nó có nghĩa là thực hiện tác vụ với mục đích tăng khả năng làm chủ của bạn về một hoặc nhiều khía cạnh của tác vụ. Nó có nghĩa là lặp lại sự lặp lại. Dần dần, lặp đi lặp lại, cho đến khi bạn đạt được mức độ thành thạo mong muốn. Bạn luyện tập có chủ đích để làm chủ tác vụ, không phải để hoàn thành tác vụ. Mục đích chính của phát triển có trả tiền là hoàn thành một sản phẩm, trong khi mục đích chính của luyện tập có chủ đích là cải thiện hiệu suất của bạn. Chúng không giống nhau. Hãy tự hỏi, bạn dành bao nhiêu thời gian để phát triển sản phẩm của người khác? Làm thế nào nhiều phát triển bản thân? Cần bao nhiêu luyện tập có chủ đích để có được chuyên môn?

- Peter Norvig viết rằng đó có thể là 10.000 giờ ... là con số kỳ diệu. (*)
- Trong Phát triển phần mềm tinh gọn (Addison-Wesley Professional) , Mary Poppendieck lưu ý rằng, cần những người điều hành giỏi tối thiểu 10.000 giờ luyện tập một cách tập trung và có chủ đích để trở thành chuyên gia.

Chuyên môn tích luỹ dần dần theo thời gian, không phải tất cả đến cùng một lúc trong 10.000 giờ! Tuy nhiên, 10.000 giờ là rất nhiều: khoảng 20 giờ mỗi tuần trong 10 năm. Với mức độ cam kết này, bạn có thể lo lắng rằng bạn không phải là tài liệu chuyên gia. Bạn là tài liệu. Sự vĩ đại phần lớn là vấn đề của sự lựa chọn có ý thức. Lựa chọn của bạn. Nghiên cứu trong hai thập kỷ qua đã chỉ ra rằng yếu tố chính trong việc thu nhận chuyên môn là thời gian dành cho việc luyện tập có chủ đích. Khả năng bẩm sinh không phải là yếu tố chính.

- **Mary**: “Có sự đồng thuận rộng rãi giữa các nhà nghiên cứu về hiệu suất của chuyên gia rằng tài năng bẩm sinh không chiếm nhiều hơn một ngưỡng; bạn phải có một

lượng nhỏ khả năng tự nhiên để bắt đầu trong một môn thể thao hoặc nghề nghiệp. Sau đó, những người xuất sắc là những người làm việc chăm chỉ nhất.”

Không có nhiều lý do để luyện tập có chủ đích một cái gì đó mà bạn đã là một chuyên gia. Luyện tập có chủ đích có nghĩa là luyện tập một cái gì đó bạn không giỏi.

- **Peter**: “Chìa khóa [để phát triển chuyên môn] là luyện tập có chủ đích: không chỉ làm đi làm lại mà còn thử thách bản thân với một tác vụ vượt quá khả năng hiện tại của bạn, thử nó, phân tích hiệu suất của bạn trong và sau khi thực hiện nó và sửa chữa bất cứ sai lầm nào.”
- **Mary**: “Luyện tập có chủ đích không có nghĩa là làm những gì bạn giỏi; nó có nghĩa là thử thách bản thân bạn, làm những gì bạn không giỏi. Vì vậy, nó không nhất thiết phải là niềm vui”

Thực hành có chủ ý là học tập. Về việc học cách thay đổi bạn; học cách thay đổi hành vi của bạn. Chúc bạn may mắn.

Phần 23: Domain-Specific Languages

Bất cứ khi nào bạn nghe được một cuộc thảo luận của những chuyên gia trong nghề, có thể là những cờ thủ, giáo viên mẫu giáo, hoặc đại lý bảo hiểm, bạn sẽ thấy rằng họ dùng những từ khá khác so với ngôn ngữ hằng ngày. Đó là một phần của câu trả lời cho câu hỏi ngôn ngữ miền chuyên biệt (DSLs) là gì: Một tên miền cụ thể có từ vựng chuyên ngành để mô tả những thứ có trong tên miền đó.

Trong thế giới phần mềm, DSLs bao gồm các biểu thức thực thi trong một ngôn ngữ cụ thể chuyên dùng cho một miền với vốn từ vựng và ngữ pháp hạn chế, có thể đọc được, hiểu được và - hy vọng - có thể ghi bởi các chuyên gia nghiệp vụ. DSLs nhắm mục tiêu vào những nhà phát triển phần mềm hoặc nhà khoa học đã được một khoảng thời gian dài. Ví dụ: Unix ‘ngôn ngữ nhỏ bé’ được tìm thấy trong các tập tin cấu hình hay những ngôn ngữ được tạo ra bởi sức mạnh của các macro LISP là một trong những ví dụ cũ hơn.

DSLs thường được phân thành hai loại là internal hoặc external:

- Internal DSLs được viết bằng ngôn ngữ lập trình đa năng (general purpose programming language) với cú pháp đã được uốn nắn để trông giống ngôn ngữ tự nhiên hơn. Điều này dễ dàng hơn đối với các ngôn ngữ cung cấp nhiều cú pháp đặc biệt và khả năng định dạng (ví dụ: Ruby và Scala) so với những ngôn ngữ không có (ví dụ: Java). Hầu hết internal DSLs bao bọc các API, thư viện hoặc business code và cung cấp một trình bao bọc (a wrapper) để truy cập chức năng ít hơn. Họ có thể thực hiện trực tiếp bằng cách chỉ cần chạy chúng. Tùy thuộc vào việc triển khai và miền, chúng được sử dụng để xây dựng cấu trúc dữ liệu, định nghĩa các phụ thuộc, chạy các quy trình hoặc tác vụ, giao tiếp với các hệ thống khác hoặc xác thực đầu vào của người dùng. Cú pháp của một internal DSL bị ràng buộc bởi hệ ngôn ngữ chủ (host language). Có nhiều mẫu (patterns) - ví dụ: trình tạo biểu thức, chuỗi phương thức và chú thích - có thể giúp bạn uốn nắn hệ ngôn ngữ chủ thành DSL của bạn. Nếu hệ ngôn ngữ chủ không yêu cầu biên dịch lại, một internal DSL có thể được phát triển khá nhanh, hoạt động song song với chuyên gia nghiệp vụ.

- External DSLs là biểu thức văn bản hoặc biểu thức đồ họa của ngôn ngữ - mặc dù DSLs văn bản (textual DSLs) có xu hướng phổ biến hơn so với đồ họa. Biểu thức văn bản có thể được xử lý bằng chuỗi công cụ bao gồm lexer, parser, model transformer, generator và bất kì loại xử lý hậu kỳ nào khác. External DSLs chủ yếu được đọc vào các mô hình bên trong tạo thành cơ sở để xử lý thêm. Nó rất hữu ích khi xác định ngữ pháp (ví dụ: trong EBNF). Một ngữ pháp cung cấp điểm bắt đầu để tạo các phần của chuỗi công cụ (ví dụ: editor, visualizer, parser generator). Đối với DSLs đơn giản, có thể chỉ cần trình phần tích thủ công (handmade parser) là đủ - ví dụ: sử dụng các biểu thức thông thường. Các trình phân tích cú pháp tùy chỉnh (custom parser) có thể trở nên khó dùng nếu chúng bị hỏi quá nhiều, vì vậy sẽ rất hợp lý khi xem xét các công cụ được thiết kế riêng để làm việc với ngữ pháp và DSLs - ví dụ: openArchitectureWare, ANTlr, SableCC, AndroMDA. Định nghĩa external DSLs như XML cũng khá phổ biến, mặc dùng khả năng đọc thường là một vấn đề - đặc biệt đối với những người đọc không thuộc chuyên môn.

Bạn phải luôn đưa đối tượng mục tiêu của DSL của bạn vào một tài khoản. Họ là nhà phát triển, nhà quản lý, khách hàng doanh nghiệp hay người dùng cuối? Bạn phải điều chỉnh trình độ kỹ thuật của ngôn ngữ, các công cụ có sẵn, trợ giúp cú pháp (ví dụ: intellisense), xác thực sớm, trực quan hóa và đại diện cho đối tượng dự định. Bằng cách ẩn đi chi tiết kỹ thuật, DSLs có thể trao quyền cho người dùng bằng cách cung cấp cho họ khả năng thích ứng các hệ thống theo nhu cầu của họ mà không cần sự trợ giúp của các nhà phát triển. Nó cũng có thể tăng tốc độ phát triển vì phân phối công việc tiềm năng sau khi khung ngôn ngữ ban đầu được đưa ra. Ngôn ngữ có thể được phát triển dần dần. Ngoài ra còn có các đường di chuyển khác nhau cho các biểu thức hiện và ngữ pháp hiện có.

Phần 24: Đừng sợ đột phá

Những người có kinh nghiệm trong nghề chắc hẳn đã từng làm ở những project mà chất lượng codebase kém. Dự án được thiết kế một cách tồi tệ, và chỉnh sửa một vấn đề luôn luôn làm ảnh hưởng ít hay nhiều đến những tính năng hoàn toàn không liên quan khác. Bất cứ khi nào một module mới được thêm vào, mục đích của người lập trình viên là làm thay đổi ít nhất có thể, và “nín thở” mỗi khi phát hành phiên bản mới. Điều này khiến chương trình tương tự việc chơi trò rút gỗ với những thanh sắt I trong những tòa nhà chọc trời và chính nó là cội nguồn của tai hoạ.

Lý do làm cho những sự thay đổi trở nên đáng lo lắng chính là do dự án của bạn có vấn đề. Nó cần một vị “bác sĩ” nếu không thì nó sẽ ngày càng trở nên tồi tệ hơn. Bạn đã biết chuyện gì xảy ra với dự án của bạn nhưng bạn lại sợ “đập vỡ cái trứng để làm một cái trứng rán”. Một bác sĩ phẫu thuật giỏi biết rằng vết cắt phải thực hiện để quá trình phẫu thuật có thể diễn ra đồng thời anh ta cũng biết rằng vết cắt ấy là tạm thời và sẽ hồi phục. Và kết quả cuối cùng của ca phẫu thuật là làm tiêu biến cơn đau, và bệnh nhân sẽ hồi phục khỏe mạnh hơn họ khi chưa thực hiện ca phẫu thuật.

Đừng sợ hãi code của bạn. Không ai quan tâm nếu có thứ gì đó tạm thời hỏng trong khi bạn đang thay đổi thứ gì đó. Nỗi sợ phải thay đổi là nguyên nhân khiến cho dự án của bạn chìm trong trạng thái lỗi. Đầu tư thời gian cho việc chỉnh sửa chúng sẽ giúp chúng

ta tiết kiệm thời gian trong những lần bảo trì trong suốt vòng đời dự án của bạn. Thêm vào đó bạn còn có thể nâng cao kinh nghiệm của cả team bạn khi xử lý những dự án bị hỏng khiến bạn trở thành “chuyên gia” trong việc hiểu nó nên hoạt động như thế nào. Hãy ứng dụng kiến thức này thay vì khó chịu khi đối mặt với nó. Làm việc trong một dự án mà bạn không thích thì cực kỳ tốn thời gian.

Hãy tái định nghĩa về giao diện, tái cấu trúc những khối lệnh, chỉnh sửa những đoạn code mà bạn copy và paste, và đơn giản hoá dự án của bạn khiến nó bớt phụ thuộc hơn. Bạn còn có thể giảm độ phức tạp thuật toán của bạn một cách rõ rệt bằng việc xét các trường hợp gốc cái thường là kết quả của việc kết hợp không vững vàng giữa các tính năng. Chậm rãi chuyển từ những cấu trúc đã lỗi thời bằng những cấu trúc tân tiến và song song kiểm tra sự chính xác của chúng. Đừng ôm quá nhiều việc một lúc chúng sẽ khiến bạn gặp rất nhiều rắc rối dẫn đến dễ chán nản và bỏ cuộc giữa chừng.

Hãy trở thành vị bác sĩ không sợ việc phải giải phẫu sáu bộ phận để tạo tiền giúp bệnh nhân hồi phục. Thái độ của bạn sẽ truyền cho mọi người và tạo cho họ cảm hứng bắt đầu làm việc trở lại cải tiến những dự án mà họ đã dẹp sang một bên. Hãy giữ một danh sách những công việc “dọn dẹp” mà nhóm của bạn cảm thấy những công việc ấy giúp ích cho lợi ích chung của dự án của team bạn. Hãy thuyết phục mọi người rằng mặc dù chúng ta không thể thấy kết quả của những việc ấy nhưng chính chúng sẽ góp phần làm giảm chi phí và đẩy nhanh việc phát hành tính năng mới. Hãy luôn quan tâm đến tình trạng code của bạn như sức khoẻ của chính bản thân ta.

Phần 25: Don't Be Cute with Your Test Data

Trời đã khuya. Tôi đã đưa vào một số dữ liệu giữ chỗ để kiểm tra bố cục trang tôi đang làm việc.

Tôi dùng tên của các thành viên trong ban nhạc The Clash để làm tên người dùng. Tên công ty thì sao? Tên các bài hát của Sex Pistols sẽ làm điều đó. Bây giờ tôi cần một số biểu tượng chứng khoán - chỉ cần bốn chữ cái trong bảng chữ in hoa.

Tôi đã sử dụng bốn chữ cái đó.

Nó có vẻ vô hại. Chỉ cần một cái gì đó để tự giải trí, và có lẽ các nhà phát triển khác vào ngày hôm tới trước khi tôi kết nối với các nguồn dữ liệu thực sự.

Sáng hôm sau, một ông quản lý dự án đã lấy một số ảnh chụp màn hình cho bài thuyết trình.*

Lịch sử lập trình tràn ngập các loại câu chuyện chiến tranh. Những điều mà các nhà phát triển và thiết kế đã làm "mà không ai khác sẽ thấy", điều bất ngờ trở nên hữu hình. Loại rò rỉ có thể khác nhau, nhưng khi nó xảy ra, nó có thể gây thiệt hại nặng nề cho người, nhóm hoặc công ty chịu trách nhiệm. Những ví dụ bao gồm:

Trong suốt cuộc họp, khách hàng nhấp vào nút chưa được thực hiện. Họ được thông báo: "Đừng nhấp vào đó một lần nữa, đồ ngu ngốc."

Một lập trình viên duy trì một hệ thống cũ đã được yêu cầu thêm một hộp thoại báo lỗi và quyết định sử dụng đầu ra của nhật ký hậu trường hiện có để cung cấp năng lượng cho nó. Người dùng bất ngờ phải đối mặt với các thông điệp như "Cơ sở dữ liệu thần thánh thất bại, Batman!" khi một cái gì đó toang.

Ai đó trộn lẫn các giao diện kiểm tra và quản trị trực tiếp và thực hiện một số mục nhập dữ liệu "hài hước". Khách hàng nhận thấy một "máy mát xa cá nhân hình Bill Gates" trị giá 1 triệu USD được bán trong cửa hàng trực tuyến của bạn.

Để phù hợp với câu nói cũ rằng "một lời nói dối có thể đi được nửa vòng trái đất trong khi sự thật đang đi trên đôi giày của nó", trong thời đại ngày nay, một vụ lừa đảo có thể là Dugg, Twittered và Flibflarbed trước khi bất kỳ ai trong múi giờ của nhà phát triển thức dậy để làm bất cứ điều gì về nó.

Ngay cả mã nguồn của bạn cũng không nhất thiết phải xem xét kỹ lưỡng. Vào năm 2004, khi một tệp tin nén của mã nguồn Windows 2000 xuất hiện trên các mạng chia sẻ tệp, một số người vui vẻ tiếp cận nó vì những lời tục tĩu, lăng mạ và nội dung hài hước khác. (Nhận xét này, tôi sẽ thừa nhận, thỉnh thoảng bị tôi chiếm đoạt!)

// TERRIBLE HORRIBLE NO GOOD VERY BAD HACK

Tóm lại, khi viết bất kỳ văn bản nào trong mã của bạn - cho dù bình luận, ghi nhật ký, hộp thoại hoặc dữ liệu thử nghiệm - luôn tự hỏi bản thân nó sẽ trông như thế nào nếu nó trở nên công khai. Nó sẽ lưu một số khuôn mặt đỏ tròn đấy.

Phần 26: Đừng bỏ qua những cảnh báo lỗi

Một đêm, tôi đang đi bộ đến quán bar để gặp vài người bạn. Chúng tôi đã không uống bia cùng nhau một thời gian và tôi đang rất mong gặp lại họ. Trong lúc hối hả, tôi không để ý những thứ tôi lướt qua. Tôi vấp vào một mô đất và ngã dập mặt. Vâng, tôi đã bất cẩn và lãnh hậu quả ngay tức khắc.

Mặc cho cái chân đau, tôi vẫn vội vã đi đến điểm hẹn gặp các bạn. Tôi đứng dậy và tiếp tục đi. Càng đi xa, cơn đau càng trở nên tồi tệ. Mặc dù cú ngã không khiến tôi quá bận tâm, nhưng tôi cảm thấy có gì đó không ổn.

Tôi vẫn đến quán bar như đã hẹn. Và khi tôi đến nơi, nó thật sự đau đớn. Tôi không thể có một đêm vui chơi như mong đợi, bởi vì cơn đau cứ hành hạ tôi. Đến sáng, tôi đi khám và nhận ra xương cẳng chân đã bị gãy. Tôi đúng ra đã tránh được những tổn thương này, nếu đêm hôm đó tôi quyết định dừng lại thay vì tiếp tục chuyến đi ngay khi cảm nhận được cơn đau. Đó thật là buổi sáng tồi tệ nhất trên đời!!

Rất nhiều lập trình viên đang viết code giống như cách tôi tạo nên cái đêm thảm họa kia.

Lỗi?? Lỗi nào? Nó không quan trọng. Trung thực mà nói, tôi có thể bỏ qua nó. Đó không phải là phương châm đúng đắn để viết lên những đoạn code chặt chẽ. Trên thực tế, nó chẳng qua là cách biện minh cho sự lười biếng thì đúng hơn. (Một lựa chọn sai lầm.) Bất kể bạn nghĩ như thế nào về lỗi trong đống code của mình, bạn luôn cần kiểm tra, và luôn cần xử lý nó. Mọi lần như một. Nếu bạn bỏ qua việc này, chẳng những không giúp bạn tiếp kiệm thời gian mà nó còn ẩn chứa những nguy cơ tiềm ẩn về sau.

Chúng ta kiểm tra lỗi trong code theo nhiều cách, bao gồm:

Mã trả về được sử dụng như là kết quả cuối cùng của một hàm (function) với hàm ý rằng “nó không chạy được”. Lỗi này rất dễ bị bỏ qua. Bởi vì nó chẳng hề được thông báo gì cả. Trên thực tế, đối với các functions trong ngôn ngữ C, việc bỏ qua những mã trả về kiểu này đã trở thành một thói quen chung mất rồi. Bạn có mấy khi sử dụng hàm printf để kiểm tra mã trả về đâu, đúng không?

Errno (error number - mã số lỗi) đây là một biến số đặc trưng của C, một biến global riêng biệt được thiết lập để thông báo lỗi. Nó cũng hay bị bỏ qua, cũng rất khó xài, và thường mang tới một mớ vấn đề gây nhức óc. Ví dụ, sẽ thế nào nếu bạn gọi đa luồng trong một hàm? Một số môi trường có thể ngăn bạn phạm vào những lỗi thế này. Nhưng hầu hết thì không

Exception (ngoại lệ) là một phương án được thiết kế để hỗ trợ cho việc thông báo và xử lý lỗi trong ngôn ngữ lập trình. Và nó không thể bị bỏ qua. Mà thực ra vẫn có thể!! Tôi đã chứng kiến nhiều đoạn code như thế này.

```
try {
    // ...do something...
}
catch (...) {} // ignore errors
```

Ít ra cái cấu trúc khủng khiếp này cũng mang lại đôi chút ý nghĩa. Đó là nó đang cảnh báo một thực tế là bạn đang làm một việc gì đó thiếu lương tâm.

Nếu bạn bỏ qua một thông báo lỗi, giả mù và vờ như chẳng có gì sai sót xảy ra, thì bạn đang gây ra những rủi ro lớn. Làm việc bất chấp các cảnh báo có thể dẫn tới những sự hủy hoại rất nghiêm trọng, giống như cái cẳng chân của tôi đã phải gánh hậu quả nặng nề, vì tôi đã không dừng ngay việc đi lại. Vì vậy, hãy luôn xử lý các vấn đề ngay khi nó xuất hiện. Chớ để đêm dài lắm mộng.
Việc không xử lý những cảnh báo lỗi có thể dẫn tới:

- Code không chặt chẽ. Code theo cảm tính và rất khó phát hiện bugs
- Code không an toàn. môt lỗi không được xử lý tốt có thể bị kẻ khác lợi dụng để đột nhập vào hệ thống.
- Cấu trúc tồi. Nếu như code của bạn bị lỗi và chúng cứ lặp đi lặp lại liên tục, thì chắc chắn là cách tiếp cận của bạn không ổn. Khắc phục nó sẽ giúp những thông báo lỗi trở nên dễ đọc, và việc xử lý chúng cũng sẽ đỡ vất vả hơn.

Cũng như việc kiểm tra lỗi trong code, việc dự liệu trước các điều kiện, khả năng gây phát sinh lỗi cũng rất cần thiết. Không nên trốn tránh việc này và làm như thể ứng dụng của bạn sẽ hoạt động trong mọi tình huống.
Tại sao chúng ta không kiểm tra những cảnh báo lỗi? Có nhiều lý do. Lý do nào bạn đồng ý? Nếu không đồng ý, bạn sẽ phản bác chúng như thế nào?

- Việc xử lý những thông báo lỗi sẽ làm code trở nên hỗn loạn, khó đọc, và khó xác định những dòng code có chức năng “bình thường” ở đâu.
- Đó là việc phụ, còn tôi đang bị deadline dí rồi!!!
- Tôi biết hàm này sẽ không bao giờ trả về lỗi (hàm printf luôn hoạt động trôi chảy, hàm malloc luôn trả về vùng nhớ mới -– mà giả sử nó không hoạt động thì chúng ta sẽ gặp một vấn đề khác còn nghiêm trọng hơn…)
- Đây chỉ là một chương trình làm giải trí thôi, Không cần phải nâng level nó lên như thế.

Phần 27: Đừng chỉ học ngôn ngữ, hãy hiểu văn hóa của nó

Ở trường trung học, tôi phải học ngoại ngữ.Vào thời điểm đó, tôi đã nghĩ rằng mình đã giỏi tiếng Anh nên tôi đã chọn (ngủ) quên trong suốt ba năm học tiếng Pháp.Vài năm sau tôi đến Tunisia vào kỳ nghỉ.Ở đây tiếng Ả Rập là ngôn ngữ chính thức , ngoài ra thì tiếng pháp cũng được sử dụng phổ biến do nơi đây trước kia từng là một thuộc địa cũ của Pháp .Tiếng Anh chỉ được nói ở các khu vực du lịch.Vì sự thiếu hiểu biết về ngôn ngữ của mình, tôi thấy mình bị giam cầm tại bể bơi khi đang đọc cuốn Finnegans Wake - một kiệt tác thành công của James Joyce về hình thức lẫn ngôn ngữ.Sự pha trộn vui tươi của hơn bốn mươi ngôn ngữ của Joyce là một trải nghiệm mệt mỏi đáng ngạc nhiên.Nhận ra cách mà các từ và cụm từ nước ngoài đan xen đã tạo cho tác giả những cách thể hiện mới về bản thân là điều mà tôi đã giữ trong sự nghiệp lập trình của mình.

Trong cuốn sách SEMINAL của họ,The Pragmatic Programmer ,Andy Hunt and Dave Thomas khuyến khích chúng ta học một ngôn ngữ lập trình mới mỗi năm.Tôi đã cố gắng sống theo lời khuyên của họ và trong suốt những năm qua tôi đã có kinh nghiệm lập trình bằng nhiều ngôn ngữ. Bài học quan trọng nhất của tôi từ những cuộc phiêu lưu đấy là cần nhiều hơn(so với việc) là chỉ học cú pháp hay công thức để học một ngôn ngữ: Bạn cần hiểu văn hóa của nó.Bạn có thể viết Fortran bằng bất kỳ ngôn ngữ nào, nhưng để thực sự học một ngôn ngữ, bạn phải (Thực sự hiểu) ngôn ngữ đó. Đừng viện cớ nếu mã nguồn C# là một phương thức Main với các phương thức trợ giúp tĩnh chủ yếu, nhưng tìm hiểu lý do tại sao các lớp có ý nghĩa. Đừng né tránh nếu bạn gặp khó khăn trong việc hiểu các biểu thức lambda được sử dụng trong các ngôn ngữ chức năng, hãy ép mình sử dụng chúng.

Khi bạn đã học được các ngôn ngữ mới, bạn sẽ ngạc nhiên về cách bạn sẽ bắt đầu sử dụng các ngôn ngữ bạn đã biết theo những cách mới.Tôi đã học cách sử dụng delegate một cách hiệu quả trong C # từ lập trình Ruby, release toàn bộ tiềm năng (full

of potential) của .NET generics đã cho tôi ý tưởng về cách tôi có thể làm cho Java generics trở nên hữu ích hơn và LINQ đã tự mình dạy Scala.

Bạn cũng sẽ hiểu rõ hơn về design patterns bằng cách di chuyển giữa các ngôn ngữ khác nhau. Các lập trình viên C thấy rằng C # và Java đã thương mại hóa mẫu lặp.

Trong Ruby và các ngôn ngữ động khác, bạn vẫn có thể sử dụng một khách truy cập, nhưng việc triển khai của bạn sẽ không giống như ví dụ từ cuốn sách Gang of Four.

Một số người có thể lập luận rằng cuốn Finnegans Wake là không thể đọc được, trong khi những người khác hoan nghênh nó vì vẻ đẹp phong cách của nó. Để làm cho cuốn sách trở nên ít khó đọc hơn thì cuốn sách đã có sẵn các bản dịch ngôn ngữ. Trớ trêu thay , đầu tiên trong số này là bằng tiếng Pháp. Code theo nhiều cách tương tự nhau. Nếu bạn viết code Wakese bằng một ít Python, một số Java và một gợi ý về Erlang, các dự án của bạn sẽ là một mớ hỗn độn.Thay vào đó, nếu bạn khám phá các ngôn ngữ mới để mở rộng tâm trí và có được những ý tưởng mới về cách bạn có thể giải quyết mọi thứ theo nhiều cách khác nhau, bạn sẽ thấy rằng code bạn viết bằng ngôn ngữ cũ đáng tin cậy của bạn trở nên đẹp hơn cho mọi ngôn ngữ mới bạn đã học.

Phần 28: Đừng cố gắng rập khuôn chương trình của bạn.

Tôi đã từng viết một bài kiểm tra C++, trong bài kiểm tra đó tôi đề xuất một cách để xử lý ngoại lệ:

Đó là thêm rất nhiều cấu trúc try…catch ở trong toàn bộ codebase. Thi thoảng chúng tôi đã ngăn chặn lỗi trong ứng dụng của chúng tôi. Do đó chúng tôi đã nghĩ về trạng thái kết quả là “rập khuôn”

Mặc dù hiện tại tôi khá giàu, tôi đã thật sự tóm tắt lại một bài học mà tôi đã nhận được từ những kinh nghiệm rất cay đắng.
Đó là 1 ứng dụng hết sức cơ bản trong thư viện mà chúng tôi tự làm nên, … Nó chứa các đoạn mã để có thể đối phó với tất cả các ngoại lệ. Và dẫn đầu từ Yossian trong cuốn tiểu thuyết Catch-22 (Catch-22 là một cụm từ chỉ một hoàn cảnh, tình huống khó xử mà người ta không thể thoát ra được vì bị mắc kẹt bởi những logic và ràng buộc mâu thuẫn nội tại.

Từ này bắt nguồn từ tên cuốn tiểu thuyết Catch-22 (1961) của Joseph Heller kể về một anh chàng phi công tên là Yossarian. Yossarian là phi công lái máy bay chiến đấu cho quân đội Ý trong Thế Chiến II. Do lo sợ, Yossarian đã cố tìm cách không phải bay bằng cách tuyên bố mình bị điên. Tuy nhiên, người ta lại bảo anh ta rằng, chỉ có người điên mới muốn bay lúc đó, vì anh ta không muốn bay nên điều đó chứng tỏ anh ta không bị điên, và vì anh ta không bị điên nên anh ta phải tiếp tục bay!). Vì thế chúng tôi quyết định rằng, hay đúng hơn là cảm thấy chúng tôi nên tiếp tục phát triển thư viện này hoặc để nó chết dần chết mòn trong sự cố gắng. và nỗ lực.

Đến cuối cùng, chúng tôi đan xen nhiều trình xử lý ngoại lệ. Chúng tôi đã trộn cấu trúc xử lý ngoại lệ của Windows với loại ngôn ngữ khác. (Bạn nhớ cấu trúc try … except trong C++ chứ? , tôi nhớ đấy) . Khi xảy ra ngoại lệ một cách không mong muốn, chúng tôi đã cố gắng gọi lại chúng một lần nữa, và cho chúng nhận các tham số khó hơn. Và nhìn lại, chúng tôi thấy khi viết một chuỗi xử lý try … catch trong khối catch của một chuỗi try...catch khác. Một loại nhận thức nào đó len lỏi vào tôi rằng tôi có thể đã vô tình đi từ con đường cao tốc trơn của sự thực hành tốt vào một con đường vô cảm của sự mất trí. Tuy nhiên, đây có lẽ là hồi tưởng khôn ngoan.

Không cần phải nói, bất cứ khi nào có sự cố xảy ra trong ứng dụng mà có sử dụng thư viện của chúng tôi. Những sự cố đó sẽ biến mất như các nạn nhân của mafia ở bến cảng. Không để lại bất cứ một dấu vết nào để dẫn đến những sự kinh khủng kế tiếp xảy ra. Bất chấp các thói quen được cho là thảm họa. Cuối cùng chúng tôi đã lưu lại những gì chúng tôi đã làm, một sự đáng hổ thẹn. Chúng tôi đã thay thế toàn bộ mỡ hỗn độn ở trong thư viện đó bằng một cơ chế thông báo lỗi tốt và mạnh mẽ. Nhưng có rất nhiều tai nạn có thể xảy ra.

Tôi sẽ không làm phiền bạn về điều này - bởi vì tôi biết không ai có thể ngu ngốc như chúng tôi. Nhưng trong một cuộc tranh luận trực tuyến gần đây với một người có chức danh học thuật nên hiển nhiên anh ta biết rõ hơn. Chúng tôi đã thảo luận về đoạn mã Java khi giao dịch từ xa. Và nếu không thành công, anh ta đã lập luận, nên bắt và chặn ngoại lệ ngay tại chỗ. (Và sau đó làm gì với nó? Tôi hỏi: “Nấu cho nó bữa tối?”

Anh ta đã trích nguyên văn nguyên tắc thiết kế giao diện người dùng. KHÔNG BAO GIỜ CHO NGƯỜI DÙNG NHÌN THẤY THÔNG BÁO NGOẠI LỆ. Thay vì giải quyết vấn đề, liệu chuyện gì sẽ xảy ra khi làm phức tạp mọi thứ. Tôi tự hỏi liệu anh ta có chịu trách nhiệm về mã ở một trong những máy ATM bị màn hình xanh ở trong những bức ảnh được đăng trên blog không, và có thể chúng đã bị hỏng vĩnh viễn. Dù sao thì khi gặp anh ta bạn nên cúi đầu cười, giống như đi về phía cửa.

Phần 29: Đừng dựa vào ‘’Phép màu”

Nếu bạn nhìn vào bất cứ hoạt động, quá trình hay sự rèn luyện nào đó một cách mơ hồ thì trông chúng có vẻ đơn giản. Những tên quản lý không có kinh nghiệm về lập trình nghĩ rằng mọi việc mà lập trình viên làm thì đều thật đơn giản, và những lập trình viên không có kinh nghiệm về quản lý cũng nghĩ theo cách tương tự như vậy.

Lập trình là một thứ gì đó mà có lẽ chỉ thiểu số có thể làm được- có thể là như vậy. Và mảng hóc búa nhất- tư duy- khó có thể nhận thấy hoặc được đánh giá cao ở những lập trình viên không chuyên. Trong hàng thập kỷ qua đã có rất nhiều nỗ lực để xóa bỏ cái tư duy sắc bén ấy . Một trong những thành quả sớm và đáng nhớ nhất là sự cố gắng của Grace Hopper - làm cho ngôn ngữ lập trình trở nên dễ hiểu hơn- điều mà nhiều bài báo đã dự đoán rằng nó sẽ loại bỏ đi những nhu cầu về chuyên gia lập trình viên. Kết quả này (COBOL - Common Business-Oriented Language) đã đang góp phần cải thiện thu nhập của các chuyên gia lập trình viên trong thập kỷ nay.

Theo lập trình viên- người mà hiểu được bản chất của phát triển phần mềm, lối suy nghĩ cố chấp rằng phát triển phần mềm có thể được đơn giản hóa bằng cách gỡ bỏ

phần mềm là một suy nghĩ thực sự ngờ nghệch. Tuy nhiên quá trình tâm lý mà dẫn đến sự sai sót này là một phần bản năng của con người và các lập trình viên cũng chỉ đang phạm phải nó như bao người khác.

Trong bất cứ dự án nào, có thể có rất nhiều việc mà một lập trình viên không tham gia một cách trực tiếp: khảo sát nhu cầu của khách hàng, gọi vốn, thiết lập máy chủ, triển khai phần mềm trên môi trường kiểm thử và trên môi trường Production, thay đổi sản phẩm đề phù hợp với business hiện hành,... Khi bạn không trực tiếp tham gia vào những việc đó, một suy nghĩ vô thức cho giả thiết rằng chúng rất đơn giản và xảy ra bởi 'phép màu". Phép màu xảy ra là một điều tốt, tuy nhiên khi mà phép màu ấy dừng lại thì dự án sẽ gặp một rắc rối lớn.

Tôi đã biết đến rất nhiều dự án mà chiếm đến hàng tuần của các developer bởi vì không ai hiểu được họ đã tin tưởng vào phiên bản ''chuẩn'' của DLL (thư viện liên kết động - Dynamic Link Library) đang được hoạt động như thế nào. Khi mọi thứ bắt đầu xảy ra một cách trì trệ, các thành viên trong nhóm đều đang lơ là, trước khi ai đó chú ý đến phiên bản ''sai'' của DLL đang được hoạt động.

Các bộ phận chạy một cách trôi chảy- dự án được phân phối đúng hạn, không còn thức khuya sửa lỗi các phiên bản, không còn fix các lỗi khẩn cấp. Một cách thật trôi chảy, rằng, các senior đã quyết định mọi thứ là để ''tự chúng chạy'' và chúng có thể thực hiện mà không cần tới project manager. Trong vòng 6 tháng các phần còn lại trong một dự án nhìn chung chỉ là việc còn lại của công ty – chậm trễ, các lỗi sẽ tiếp tục được vá lại.

Bạn không phải hiểu hết các phép màu mà giúp cho project của hoạt động, nhưng sẽ là ích lợi khi hiểu được một số phép màu đó, hoặc biết được thêm ai đó người mà hiểu được những mẹo mà bạn chưa từng biết.

Quan trọng nhất, hãy chắc chắn rằng khi phép màu dừng lại, nó vẫn có thể được bắt đầu trở lại.

Phần 30: Nguyên tắc DRY: Don't repeat yourself

Trong tất cả các nguyên tắc lập trình , Don't Repeat Yourself (DRY) có lẽ là một trong những điều cơ bản nhất. Nguyên tắc này được nhắc tới lần đầu trong cuốn sách The Pragmatic Programmer viết bởi Andy Hunt và Dave Thomas , và cũng là nền tảng cho việc phát triển các software development và design pattern khác. Những developer nào nhận ra được các sự trùng lặp, và biết cách để loại bỏ nó thông qua thực tiễn và trừu tượng hóa một cách phù hợp thì những người đó có thể có thể viết code sạch sẽ và gọn (clean code) hơn nhiều so với những người viết code lặp đi lặp lại một cách thừa thãi.

Trùng lặp là một sự thừa thãi

Mỗi một dòng code tạo nên ứng dụng đều cần được bảo trì, và nó cũng là nguy cơ tiềm ẩn cho sự xuất hiện của những con bug sau này. Sự trùng lặp sẽ "thổi phồng" codebase của bạn ,và vô tình làm cho hệ thống thêm phức tạp hơn. Sự tăng kích cỡ codebase cũng sẽ làm cho các developer khó có thể làm việc khi không thể hiểu rõ toàn bộ hệ thống, hoặc không thể chắc chắn rằng nếu sửa đổi code ở một vị trí thì có cần phải thay đổi ở những chỗ có logic tương tự hay không. Nguyên lý DRY yêu cầu " mỗi giải thuật trong một hệ thống chỉ được có một đại diện duy nhất, rõ ràng và có thẩm quyền" (câu gốc :" every piece of knowledge must have a single, unambiguous, authoritative representation within a system")

Tự động hóa sự lặp lại trong process call

Nhiều quy trình trong phát triển phần mềm được lặp đi lặp lại và dễ dàng tự động hóa. Nguyên tắc DRY phù hợp để áp dụng vào những trường hợp này cũng như trong source code của ứng dụng. Test thủ công rất tốn thời gian, dễ bị lỗi, và khó để lặp lại, vì vậy các bộ kiểm tra tự động nên được sử dụng ở những vị trí khả thi. Tích hợp phần mềm có thể sẽ mất thời gian và dễ gây lỗi nếu được làm một cách thủ cộng, do đó một quy trình xây dựng nên được cho chạy một cách thường xuyên, lý tưởng nhất là với

mỗi lần check-in. Bất cứ chỗ nào có những quy trình thủ công bị hư hại mà có thể tự động hóa, thì chúng nên được tự động hóa và tiêu chuẩn hóa. Mục đích của việc này là để đảm bảo chỉ có duy nhất một cách hoàn thành công việc, và là một cách ít gây hại nhất.

Trừu tượng hóa sự lặp lại trong Logic Call

Sự lặp lại ở trong logic có thể có nhiều dạng. copy-Paste, If-then hoặc switch-case, chúng thuộc những loại logic dễ nhận ra và dễ sửa lỗi nhất. Nhiều design pattern có mục tiêu rõ ràng về việc giảm thiểu hoặc loại bỏ sự trùng lặp logic trong một application. Nếu một object điển hình require nhiều thứ xảy ra trước khi nó được sử dụng tới thì nó có thể được làm bằng cách sử dụng pattern Abstract Factory hoặc Factory Method. Nếu một object có nhiều biến khác nhau trong behavior của nó, thì những behavior này nên được truyền vào bằng cách sử dụng Strategy pattern thay vì sử dụng một đống cấu trúc if-then. Trong thực tế, việc tạo ra các design pattern là một sự nỗ lực giảm thiểu sự lặp cấu trúc để giải quyết các vấn đề thường gặp và thảo luận về các giải pháp đó. Hơn nữa, nguyên lý DRY có thể được áp dụng trực tiếp vào cấu trúc như là database schema để đơn giản hoá nó.

Một vấn đề về nguyên lý DRY

Các nguyên tắc phần mềm khác cũng đều có liên quan tới DRY. Như nguyên tắc một và chỉ một (Once and Only Once principle), là nguyên tắc chỉ áp dụng cho các behavior function , nó cũng được coi như là một nguyên tắc bắt nguồn từ nguyên tắc DRY. Nguyên tắc đóng mở (Open/Closed Principle), là nguyên tắc chỉ ra rằng :" Các software entity nên được mở ra cho sự mở rộng, nhưng đóng lại cho sự thay đổi", trong thực tế, nguyên tắc này chỉ hoạt động khi tuân theo nguyên tắc DRY. Cũng tương tự như vậy, Nguyên tắc được biết đến khá nhiều đó là nguyên tắc Single Responsibility, nguyên tắc này yêu cầu một class " chỉ có một lý do để thay đổi ", đó là dựa trên DRY.

Khi tuân theo cấu trúc, logic, quy trình và chức năng, nguyên tắc DRY cung cấp sự hướng dẫn cơ bản cho các software developer và hỗ trợ tạo ra các phần mềm đơn giản, dễ bảo trì và có chất lượng cao hơn. Mặc dù có những trường hợp việc lặp lại có thể cần thiết để đáp ứng hiệu suất hoặc các yêu cầu khác (ví dụ : không chuẩn hóa dữ

liệu trong database), nhưng nó chỉ nên được sử dụng khi trực tiếp giải quyết một vấn đề thực tế chứ không phải là một vấn đề tưởng tượng.

Phần 31: Đừng sửa đoạn code đó

Điều này sẽ diễn ra với bất cứ ai tại một thời điểm nào đó. Code của bạn đã được chuyển đến Server cho hệ thống kiểm tra và quản lí trả lời lại rằng nó có lỗi. Phản ứng đầu tiên của bạn là “Nhanh lên, để tôi fix nó, tôi biết sai ở đâu mà!”.

Nhìn rộng ra, sai lầm ở đây là với tư cách một developer, bạn nghĩ rằng bạn có quyền truy cập vào máy chủ kiểm thử (staging server).

Trong hầu hết các môi trường được phát triển dựa trên nền web, cấu trúc có thể được chia nhỏ như sau:

- Phát triển cục bộ (local development) và đơn vị kiểm tra (unit testing) trên máy của developer.
- Máy chủ phát triển (development server) là nơi được tạo ra bởi việc tích hợp kiểm tra thủ công và tự động.
- Máy chủ kiểm thử (staging server) là nơi mà team QA và người dùng tiến hành việc kiểm tra.
- Máy chủ ứng dụng (production server)

Vâng, có nhiều máy chủ và dịch vụ được cài vào hệ thống như là kiểm soát source code và dịch vụ mua bán (ticketing). Nhưng bạn biết đó, sử dụng mô hình này, với một developer – thậm chí là một senior developer – cũng không bao giờ có quyền truy cập vào máy chủ phát triển. Hầu hết các quá trình phát triển được thực hiện trên máy cục bộ của developer bằng cách sử dụng hỗn hợp các IDE, máy ảo và thậm chí là cả “ma thuật huyền bí” (black magic) để có được may mắn.

Sau khi được đăng ký vào SCC, dù là tự động hay thủ công, việc này nên được chuyển đến máy chủ phát triển, nơi mà nó có thể được kiểm tra và điều chỉnh nếu cần thiết để

đảm bảo mọi thứ hoạt động tốt. Tuy vậy, bắt đầu từ thời điểm này, developer chỉ có nhiệm vụ theo dõi quá trình đó.

Quản lý của staging cần nén và gửi code đến máy chủ staging cho team QA. Cũng giống như các developer, QA và người dùng không cần phải truy cập vào máy chủ phát triển. Nếu đã sẵn sàng thử nghiệm, hãy tạo và gửi một bản đi, đừng yêu cầu người dùng “Chỉ nhìn những thứ chạy thật nhanh” trên máy chủ phát triển. Hãy nhớ rằng, trừ khi bạn đang tự code một dự án, những người khác cũng đang code, không ai rảnh để xem người dùng thấy sao về sản phẩm. Người quản lí phát hành chỉ là một người mà làm cả hai việc đó.

Trong mọi trường hợp, đôi khi là tất cả, một developer nên có quyền truy cập vào máy chủ của ứng dụng. Nếu có vấn đề gì đó, nhân viên hỗ trợ nên khắc phục hoặc yêu cầu bạn khắc phục nó. Sau khi được đăng ký vào SCC, họ sẽ gửi bản vá đi. Một số thảm họa lập trình tồi tệ nhất mà tôi từng gặp phải là khi ai đó *ho ho các kiểu* đã vi phạm quy tắc cuối cùng này. Nếu quy tắc này bị bỏ qua, sản phẩm sẽ không còn có thể sửa chữa được.

Phần 32: Đóng gói phương thức, không chỉ là trạng thái

Trong lý thuyết hệ thống, kiểm soát là một trong những cách xử lý tối ưu nhất khi xử lý các hệ thống có cấu trúc lớn và phức tạp. Trong ngành công nghiệp phần mềm, việc kiểm soát là cực kỳ cần thiết. Việc kiểm soát được hỗ trợ bởi các cấu trúc của ngôn ngữ lập trình như subroutines (các chương trình con) và functions (các hàm), modules (các module) và packages (các gói), classes (các lớp), v.v..

Modules và packages được sử dụng để giải quyết các nhu cầu lớn hơn cho việc kiểm soát, trong khi các class, subroutine và function lại được sử dụng để giải quyết các khía cạnh chi tiết hơn của vấn đề. Trong những năm qua, tôi phát hiện ra rằng các class

dường như là một trong những cấu trúc kiểm soát khó nhất cho các nhà phát triển trong việc lấy quyền. Điều này không phải là hiếm khi tìm thấy một class chỉ với duy nhất 1 main method (phương thức chính) với 3000 dòng lệnh hoặc một class chỉ có duy nhất các phương thức get và set cho các thuộc tính ban đầu của nó. Những ví dụ này chứng minh rằng các nhà phát triển đã không hiểu đầy đủ về tư duy hướng đối tượng, đã không tận dụng được sức mạnh của các đối tượng như các cấu trúc mô hình. Đối với các nhà phát triển đã quen thuộc với thuật ngữ POJO (Plain Old Java Object) và POCO (Plain Old C# Object hay Plain Old CLR Object), thì đây là sự quay trở lại các khái niệm cơ bản của OO như một mô hình hoá các đối tượng rõ ràng và đơn giản, nhưng không hề ngu ngốc.

Một đối tượng gói gọn cả trạng thái và phương thức, trong đó phương thức được xác định bởi thuộc tính thực tế. Ví dụ như một object cửa. Nó sẽ có 4 trạng thái: đóng, mở, đang đóng, đang mở. Nó sử dụng 2 quá trình: mở và đóng. Phụ thuộc vào trạng thái, các quá trình mở và đóng sẽ khác nhau. Thuộc tính vốn có của 1 đối tượng làm cho quá trình thiết kế đơn giản hơn về mặt khái niệm. Nó nắm 2 nhiệm vụ đơn giản: phân bổ và giao trách nhiệm cho các đối tượng khác nhau bao gồm các giao thức tương tác xen kẽ.

Cách nó hoạt động trong thực tế là một ví dụ minh hoạ rõ ràng nhất. Ví dụ chúng ta có 3 class: Customer, Order và Item. Customer là đối tượng yêu cầu xác thực thông tin tốt nhất cho việc giới hạn tín dụng và các quy tắc xác thực tín dụng. Order là đối tượng xác định mối liên kết giữa khách hàng và quá trình addItems của nó bằng cách gọi đến hàm *customer.validateCredit(item.price())*, để kiểm tra tín dụng thực tế của đối tượng Customer. Nếu như hàm đó không thành công, nó sẽ tạo ra 1 ngoại lệ và huỷ bỏ quá trình mua.

Các nhà phát triển hướng đối tượng ít kinh nghiệm hơn có thể quyết định gói tất cả các quy tắc kinh doanh vào một đối tượng như OrderManager hoặc OrderService. Trong các thiết kế này, Order, Customer và Item sẽ ghi ít bản ghi hơn. Tất cả các xử lý logic sẽ bao gồm các class và các liên kết của chúng trong một phương thức với nhiều cấu trúc if-then-els. Các method gần như không thể duy trì và dễ dàng bị phá vỡ. Còn lý do ? Do kiểm soát không trọn vẹn.

Do vậy, đừng phá vỡ sự đóng gói và sử dụng sức mạnh của ngôn ngữ lập trình mà bạn sử dụng để duy trì nó.

Phần 33: Các số dấu chấm phẩy động không phải là số thực

Số dấu phẩy động (Floating-point numbers) không phải là một dạng “số thực” như trong cách nói của Toán học, cho dù chúng vẫn được gọi là “thực” trong một số ngôn ngữ lập trình, như là Pascal hay Fortran. Các số thực có độ chính xác tuyệt đối, và do đó chúng luôn liên tục và không bị sai số; nhưng những số dấu phẩy động có tính chính xác nhất định, vì vậy chúng khá hạn chế, và được coi như là những số nguyên “kém cỏi”.
Để minh họa cho vấn đề này, ta gán thử 2147483647 (số nguyên lớn nhất trong hệ 32-bit) cho một biến kiểu float 32-bit (x,say), và in nó lên màn hình. Bạn sẽ nhận được số 2147483648. Giờ thì hãy in lên màn hình x – 64. Vẫn là con số 2147483648. Tiếp tục hãy thử in lên x – 65 và bạn sẽ có được số 2147483520! Tại sao vậy nhỉ? Bởi vì khoảng trống ở giữa các biến float trong dãy đó là 128, và thao tác của dấu phẩy động chính là làm tròn số dấu phẩy động.

Những số dấu phẩy động IEEE là những số có độ chính xác cố định dựa trên hai kiểu kí tự khoa học: 1.d1d2...dp-1 × 2e, mà p là sự chính xác (24 cho kiểu float và 53 cho kiểu double). Khoảng cách giữa hai số liên tiếp là 21-p+e, xấp xỉ bằng ε|x|, ε chính là Machine Epsilon (giới hạn trên của sai số tương đối do làm tròn trong số học điểm nổi) (21-p).

Am hiểu về sự phân bố trong số dấu phẩy động có thể giúp bạn tránh được những sai số thường gặp. Nếu bạn đang thực hiện một vòng lặp, ví dụ khi tìm nghiệm của phương trình, thật là vô lý khi bạn yêu cầu độ chính xác cao hơn trong độ sai số mà hệ thống số có thể biểu diễn. Vì vậy, hãy chắc chắn rằng giới hạn chính xác bạn yêu cầu không nhỏ hơn khoảng cách đó, bằng không thì bạn sẽ thực hiện một vòng lặp vô hạn.

Từ lúc số dấu phẩy động được xem như gần giống với số thực, chắc hẳn vẫn sẽ có một chút sai sót. Sai sót này gọi là roundoff (Làm tròn số), và có thể sẽ dẫn đến một vài kết quả rất bất ngờ. Khi bạn thực hiện phép trừ hai số gần bằng nhau, các số có ý nghĩa quan trọng nhất sẽ triệt tiêu lẫn nhau, sau đó chữ số ít quan trọng nhất (nơi xảy ra lỗi sai số) sẽ được đưa lên vị trí quan trọng nhất trong kết quả dấu phẩy động. Về cơ bản thì nó làm lệch đi bất kì hoạt động tính toán nào có liên quan (một hiện tượng được ví như là " sự nhòe (smearing)"). Bạn cần quan sát kĩ hơn các thuật toán của mình để tránh khỏi một tình huống "hủy chương trình một cách thê thảm" (catastrophic cancellation). Để làm rõ hơn, bạn hãy thử giải phương trình x2 - 100000x + 1 = 0 bằng công thức nghiệm bậc hai. Khi các toán hạng trong biểu thức -b + sqrt(b2 - 4) gần bằng nhau về độ lớn, bạn có thể tính ra nghiệm r1 = -b + sqrt(b2 - 4), và sau đó tìm được nghiệm r2 = 1/r1, trong bất kì phương trình bậc hai nào, ax2 + bx + c = 0, các nghiệm đều thỏa mãn r1r2 = c/a.

Hiện tượng Smearing thậm chí có thể xảy ra theo cả những cách tinh tế hơn. Giả sử một library tính ex theo công thức 1 + x + x2/2 + x3/3! + ... Điều này rất ổn cho một nghiệm x dương, nhưng hãy tưởng tượng điều gì sẽ xảy ra khi x lại là một số âm rất lớn. Phần chẵn trong một số dương lớn nếu có trừ cho nhiều phần lẻ cũng không ảnh hưởng đến kết quả. Vấn đề ở đây là trong việc làm tròn một số lớn, phần dương trong chữ số có vai trò quan trọng hơn nhiều so với kết quả cuối cùng. Câu trả lời là sẽ tiến tới dương vô cực Cách giải quyết cũng rất đơn giản: với x âm, ta tính được ex = 1/e|x|. Rõ ràng là bạn không nên sử dụng số dấu phẩy động cho các ứng dụng tài chính, đó là những gì mà các lớp học ngôn ngữ thuật toán như Python và C# làm việc. Số dấu phẩy động được sử dụng để việc tính toán khoa học được hiệu quả. Nhưng hiệu quả sẽ vô nghĩa nếu không có sự chính xác, vì vậy hãy ghi nhớ nguồn gốc phát sinh các lỗi và code một cách hợp lí!

Phần 34: Hiện thực hoá tham vọng của bạn với Open Source

Có lẽ bạn đang phát triển phần mềm ở nơi mà không thỏa mãn được giấc mơ đầy tham vọng của mình. Có thể bạn đang phát triển phần mềm cho một công ty bảo hiểm nào đó trong khi nơi bạn thực sự muốn làm việc là Google, Apple, Microsoft hay công ty khởi nghiệp của riêng bạn để làm nên điều vĩ đại hơn. Mọi thứ sẽ chẳng đi đến đâu cả khi bạn phải phát triển phần mềm cho các hệ thống mà bạn không mấy hứng thú.

May mắn thay, có một câu trả lời cho vấn đề của bạn: Open Source. Có hàng nghìn dự án Open Source ngoài kia- rất nhiều trong số đó khá thiết thực, cung cấp bất kỳ loại trải nghiệm phát triển phần mềm nào bạn muốn. Nếu bạn có ý tưởng phát triển một hệ điều hành mới, hãy thử sức với một trong hàng tá dự án hệ điều hành. Nếu bạn muốn làm việc với phần mềm âm nhạc, hoạt hình, mật mã, robot, trò chơi PC, trò chơi trực tuyến, điện thoại di động hay bất cứ điều gì, chắc chắn bạn sẽ tìm thấy ít nhất một dự án Open Source dành riêng cho sở thích đó.

Và tất nhiên, chẳng có gì là miễn phí cả. Bạn sẽ phải từ bỏ thời giờ nghỉ ngơi của mình, vì bạn không thể làm việc với dự án Open Source về một trò chơi nào đó trong giờ làm việc- bạn vẫn phải có trách nhiệm với ông chủ của mình. Ngoài ra, rất ít người thực sự kiếm được tiền nhờ việc đóng góp cho các dự án Open Source. Cũng có một số người kiếm chác được chút đỉnh nhưng chỉ chiếm một phần rất nhỏ. Bạn nên xác định sẵn rằng sẽ phải từ bỏ thời gian rảnh của mình (tất nhiên, cắt giảm chút thời gian chơi game và xem TV thì bạn vẫn sống tốt thôi). Với tư cách là một lập trình viên, bạn càng làm việc năng suất với các dự án Open Source, bạn càng sớm nhận ra tham vọng thực sự của mình. Điều quan trọng là phải xem xét kĩ hợp đồng của bạn- một số nhà tuyển dụng có thể hạn chế những gì bạn có thể đóng góp, thậm chí cả thời gian của bạn. Ngoài ra, bạn cũng nên cẩn thận với việc vi phạm luật sở hữu trí tuệ liên quan đến vấn đề bản quyền, bằng sáng chế, nhãn hiệu và bí mật thương mại.

Open Source cung cấp nhiều cơ hội cho các lập trình viên năng nổ. Trước tiên, bạn có thể xem người khác triển khai vấn đề mà bạn quan tâm như thế nào- bạn có thể học được nhiều điều thông qua việc đọc code của người khác. Thứ hai, bạn có thể đóng góp code và ý tưởng của riêng mình cho dự án. Đương nhiên không phải mọi ý tưởng của bạn đều được chấp nhận mà chỉ có một số có thể. Và bạn sẽ học được nhiều điều mới mẻ thông qua việc xử lý các vấn đề và đóng góp code. Thứ ba, bạn sẽ gặp những người tuyệt vời có chung niềm đam mê với bạn- những tình bạn có thể tồn tại suốt đời. Thứ tư, nếu bạn là người đóng góp nhiều cho project, bạn có thể thêm kinh nghiệm thực tế vào công nghệ thực sự khiến bạn quan tâm.

Bắt tay vào làm việc với Open Source khá dễ dàng. Có sẵn rất nhiều tài liệu về các công cụ bạn cần (ví dụ: quản lý mã nguồn, biên tập viên, ngôn ngữ lập trình, xây dựng hệ thống, v.v.). Bắt đầu với việc tìm kiếm dự án bạn muốn làm và tìm hiểu các công cụ mà dự án sử dụng. Những tài liệu về các dự án sẽ khá nhẹ nhàng trong phần lớn trường hợp, nhưng điều này có lẽ ít quan trọng hơn vì cách tốt nhất để học là tự điều tra code. Nếu bạn muốn tham gia, bạn có thể đưa ra lời đề nghị giúp đỡ. Hoặc bạn có thể bắt đầu bằng cách tình nguyện viết test code. Mặc dù điều đó nghe có vẻ không mấy thú vị, sự thật là bạn học nhanh hơn thông qua việc viết test code cho phần mềm của người khác so với hầu hết các hoạt động khác. Hãy biết test code thật tốt. Tìm bugs, đề xuất sửa lỗi, kết bạn, làm việc với phần mềm bạn yêu thích và thực hiện tham vọng phát triển phần mềm của bạn!

Phần 35: Nguyên tắc vàng trong thiết kế API

Thiết kế API không bao giờ dễ dàng, đặc biệt là những API lớn. Nếu bạn đang thiết kế một API mà có hàng trăm ngàn người dùng, bạn phải nghĩ về vấn đề làm thế nào để bạn có thể thay đổi nó trong tương lai và liệu việc đó có thể làm hỏng đến phần nào của client hay không? Hơn nữa, bạn phải nghĩ về sự ảnh hưởng đến từ người dùng của bạn lên bản thân. Nếu một trong những lớp của API bạn tạo ra dùng chính phương

thức nội bộ, bạn phải nhớ rằng một người dùng có thể tạo lớp con từ lớp của bạn và ghi đè nó, và điều đó chính là một thảm hoạ. Bạn sẽ không còn có thể thay đổi phương thức đó nữa bởi vì một vài người dùng của bạn đã thay đổi bản chất của nó, khiến nó mang một ý nghĩa khác. Sự lựa chọn cài đặt tính năng của bạn phụ thuộc vào người dùng.

Những nhà phát triển API giải quyết vấn đề này bằng nhiều cách khác nhau, nhưng cách đơn giản nhất là khoá API(lockdown API). Nếu bạn đang làm việc với Java bạn có thể cố gắng áp dụng từ khóa final cho hầu hết các lớp và phương thức. Tương tự trong C#, bạn có thể dùng từ khóa seal. Bất kể ngôn ngữ của bạn đang sử dụng là gì, bạn có thể thử thể hiện API của bạn bằng phương thức Singleton(đơn điệu) hay Static factory (dùng từ khóa static) nhờ đó bạn có thể ngừa nó khỏi bị ghi đè và sử dụng code của bạn theo cách về sau có thể giới hạn những lựa chọn của bạn. Tất cả điều này nghe có vẻ hợp lý đấy, nhưng có thật sự như thế ?

Trong thập kỷ qua, chúng ta đã dần dần nhận ra rằng tạo testing là một phần vô cùng quan trọng trong luyện tập, nhưng bài học ấy vẫn chưa thật sự tác động tích cực đến mọi ngóc ngách của ngành. Và bằng chứng cho việc ấy có ở khắp mọi nơi. Hãy chọn tuỳ ý bất kỳ một lớp nào chưa được kiểm tra và thử tạo bộ thử cho nó. Và hầu hết mọi trường hợp, bạn đều gặp rắc rối. Bạn sẽ hiểu được rằng đoạn code đó dính chặt với API như thể thống nhất không thể tách rời. Và không có một cách nào có thể sao chép những lớp của API đó để bạn có thể nhận thấy code của bạn đang tương tác với nó hay hỗ trợ giá trị trả về cho việc testing.

Theo thời gian, việc này sẽ ngày càng phát triển hơn, nhưng chỉ khi chúng ta bắt đầu nhìn nhận testing như là một trường hợp thực tế khi thiết kế APIs. Không may thay điều đó chỉ tiến bộ hơn việc testing cho code của chúng ta một chút thôi. Và đó chính là nơi để chúng ta áp dụng Nguyên tắc vàng trong thiết kế API: “Chúng ta không thể tạo ra đủ bộ thử cho API của chúng ta phát triển; chúng ta phải tạo bộ thử cho chính đoạn code dùng API của chúng ta.Và khi bạn làm điều đó, bạn sẽ biết được những trở ngại đầu tiên mà người dùng của bạn phải vượt qua khi họ muốn kiểm tra code của họ một cách độc lập.”

Không có cách nào có thể giúp những nhà phát triển có thể kiểm tra đoạn code mà chúng sử dụng API của họ. “static”, “final”, và “sealed” đều không phải là cấu trúc xấu. Chúng có thể hữu dụng trong thời gian nào đó. Nhưng điều quan trọng là chúng ta phải luôn cẩn trọng đối với những vấn đề đối với testing, và để làm điều đó, bạn phải tự trải nghiệm lấy. Một khi bạn đã trải nghiệm, bạn có thể tiếp cận nó bằng cách mà bạn muốn với bất kỳ thử thách thiết kế nào khác.

Phần 36: Thần Thoại Guru

Bất kì ai đã làm việc đủ lâu trong phần mềm đều từng nghe những câu hỏi như thế này:

- Ôi đang có sự bất bình thường XYZ. Bạn có biết đây là vấn đề gì không ?

Những người có câu hỏi ấy đều hiếm khi bận tâm các về stack traces, error logs, hoặc là các nội dung liên quan dẫn đến vấn đề đó. Họ dường như nghĩ bạn ở một mức độ khác giỏi hơn, và các tình huống đó đều xuất hiện với bạn mà không có sự phân tích nào dựa trên dấu hiệu. Họ nghĩ bạn là một thần thoại.

Chúng ta luôn được hỏi những câu hỏi đó từ những người chưa tìm hiểu về phần mềm: với họ thì những hệ thống ấy có vẻ thật thần kỳ. Điều khiến tôi quan ngại là thậm chí nhìn thấy điều này từ chính cộng đồng của chúng ta. Những câu hỏi tương tự cũng nảy sinh trong chương trình thiết kế, như là “Tôi đang phát triển việc quản trị hàng tồn kho, tôi có nên sử dụng optimistic locking không?” Trớ trêu thay, những người đặt câu hỏi thường được trang bị kiến thức tốt hơn để trả lời chúng hơn là những người nhận câu hỏi. Những người hỏi có thể sẽ biết được ngữ cảnh, hiểu rõ các yêu cầu và có khả năng thấy được những lợi thế và bất lợi trong các kế hoạch, chiến lược khác nhau. Song họ lại mong được nhận câu trả lời khôn ngoan từ bạn mà không cần kể đến bối cảnh đấy. Họ mong đợi sự kì diệu.

Đã đến lúc cho nền công nghiệp phần mềm xóa đi vị thần Guru này. “Gurus” là con người. Họ sử dụng logic và phân tích các vấn đề một cách có hệ thống như chúng ta.

Họ sử dụng lối suy nghĩ ngắn gọn và cả trực giác. Hãy để ý đến người lập trình viên giỏi nhất mà bạn biết: anh ta đã từng không giỏi về phần mềm như bạn bây giờ. Nếu có ai trông có vẻ như là một guru thì đó chính là thành quả của nhiều năm liền nỗ lực học tập và điều chỉnh tư duy. Một “guru” chỉ đơn giản là một người thông minh luôn học hỏi không ngừng.

Tất nhiên, nó vẫn còn một sự khác biệt lớn về khả năng bẩm sinh. Nhiều hackers ngoài kia rất thông minh, hiểu biết hơn và nhiều năng suất hơn bao giờ hết. Ngay cả như vậy, việc xóa đi hình ảnh thần thoại guru vẫn gây ảnh hưởng tích cực. Ví dụ, khi làm việc với những người thông minh hơn tôi, tất nhiên tôi phải làm những công việc cần di chuyển, để cung cấp những ngữ cảnh cần thiết để họ có thể áp dụng khả năng của anh/cô ấy một cách hiệu quả. Xóa bỏ thần thoại guru cũng đồng nghĩa với việc gỡ bỏ rào cản nhận thức cho sự phát triển. Thay vì rào cản ma thuật, tôi thấy tôi có thể tiến lên liên tục.

Cuối cùng, một trong những chướng ngại vật lớn nhất của phần mềm chính là những người thông minh cố tình tuyên truyền vị thần thoại guru. Việc này có thể được thực hiện từ cái tôi, hoặc là chiến thuật để nâng cao giá trị của một người theo cảm nhận của khách hàng hay nhà tuyển dụng. Trớ trêu thay, thái độ này có thể giảm giá trị của những người thông minh, bởi vì họ không đóng góp cho sự phát triển của đồng nghiệp họ. Chúng ta không cần gurus. Chúng ta cần những người chuyên nghiệp sẵn lòng giúp đỡ những người chuyên nghiệp khác phát triển trong lĩnh vực của họ. Đó là nơi cho tất cả chúng ta.

Phần 37: Chăm chỉ chưa chắc thành công

Là một lập trình viên, làm việc chăm chỉ thường không được đền đáp. Bạn có thể tự lừa dối bản thân và đồng nghiệp rằng bạn đang đóng góp rất nhiều cho một dự án bằng cách dành nhiều giờ ở văn phòng. Nhưng sự thật là chỉ cần làm ít, bạn có thể đạt

được nhiều- thậm chí nhiều hơn nữa. Nếu bạn đang cố gắng tập trung và “làm việc hiệu quả” hơn 30 giờ một tuần thì có lẽ bạn đang làm việc quá sức. Bạn nên xem xét giảm khối lượng công việc để làm việc được hiệu quả hơn và hoàn thành nhiều việc hơn.

Kết luận này nghe có vẻ trái nghịch và thậm chí còn gây tranh cãi, nhưng nó là hệ quả trực tiếp của thực tế là việc lập trình và phát triển phần mềm nói chung là cả một quá trình học tập không ngừng nghỉ. Khi bạn triển khai một dự án, bạn sẽ hiểu vấn đề hơn và tìm ra những cách hiệu quả hơn để đạt được mục tiêu. Để tránh những việc thừa thãi, bạn phải dành thời gian quan sát ảnh hưởng của những gì bạn đang làm- phản ánh qua những gì bạn nhìn thấy và thay đổi hành vi của bạn cho phù hợp.

Lập trình chuyên nghiệp thường không giống như chạy vài km, với mục tiêu nằm ở ngay cuối con đường. Hầu hết các dự án phần mềm giống như một cuộc đua marathon trong- bóng- tối, với chỉ một tấm bản đồ sơ sài. Nếu bạn chỉ hướng về một hướng và chạy nhanh nhất có thể, bạn có thể gây ấn tượng với một số người, nhưng bạn không có khả năng thành công. Bạn cần duy trì một tốc độ bền vững và bạn cần điều chỉnh ít nhiều khi tìm hiểu về vị trí của bản thân cũng như vạch đích mà bạn đang hướng tới.

Ngoài ra, bạn luôn cần tìm hiểu thêm về phát triển phần mềm nói chung và kỹ thuật lập trình nói riêng. Bạn có thể đọc sách, tham dự hội nghị, giao tiếp với các chuyên gia, thử nghiệm các kỹ thuật mới và tìm hiểu về các công cụ giúp đơn giản hóa công việc của bạn. Là một lập trình viên chuyên nghiệp, bạn phải luôn tự cập nhật lĩnh vực chuyên môn của mình - giống như các bác sĩ phẫu thuật và phi công luôn phải cập nhật lĩnh vực chuyên môn riêng của họ. Bạn cần dành buổi tối, cuối tuần và ngày lễ để tự học, do đó khoảng thời gian này không thể dành để làm thêm giờ cho dự án hiện tại của bạn. Liệu bạn có mong các bác sĩ phẫu thuật dành 60 giờ một tuần trong phòng mổ, hay phi công bay đến 60 giờ một tuần? Tất nhiên là không, vì sự chuẩn bị và giáo dục là một phần thiết yếu trong nghề nghiệp của họ.

Hãy tập trung vào dự án, đóng góp nhiều nhất có thể bằng cách tìm những giải pháp thông minh, cải thiện kỹ năng của bạn, thể hiện những gì bạn làm và điều chỉnh hành vi của bạn. Tránh tự bôi xấu bản thân bằng cách cư xử như một con chuột đồng trong

một cái lồng quay. Là một lập trình viên chuyên nghiệp, bạn nên biết rằng cố gắng tập trung và “làm việc hiệu quả” 60 giờ một tuần không phải điều hợp lý. Hãy hành động như một chuyên gia: chuẩn bị, thực hiện, quan sát, phản ánh và thay đổi.

Phần 38: Làm thế nào để săn bug?

Dù cho bạn có gọi là lỗi, dị tật, hay kể cả tác dụng phụ thiết kế, thì bạn chả có một cách nào thoát khỏi chúng. Hiểu cách để nộp một bản báo lỗi tốt và biết nên tìm kiếm gì trong đó, là một trong những kỹ năng then chốt giúp project của bạn phát triển một cách trơn tru.

Một bản báo cáo lỗi tốt gồm có ba điều:

- Nguyên nhân gây ra lỗi, càng chi tiết càng tốt, và tần suất xuất hiện của chúng.
- Chúng ta nên thực hiện điều gì, ít nhất là ý kiến của bạn.
- Điều gì đã xảy ra trong thực tế, hoặc ít nhất là những thông tin mà bạn đã ghi nhận được.

Số lượng và chất lượng của toàn bộ thông tin được báo cáo về lỗi đấy không chỉ giúp ta hiểu về bug mà còn giúp ta hiểu hơn về người phát triển. Sự tức giận, chửi bugs(“hàm này như hạch”) cho những nhà phát triển rằng bạn đang có một khoảng thời gian tồi tệ, và đó là tất cả. Một bug với phạm vi rộng khiến nó dễ dàng nhân lên và rồi nhận được sự dè chừng của mọi người ngay cả khi chúng dừng hoạt động.

Các lỗi ấy giống như những cuộc hội thoại, với tất lịch sử ngay đó trước mặt mọi người. Đừng đổ lỗi cho người khác và phủ định sự tồn tại của nó. Thay vào đó hãy hỏi để có thêm thông tin hay tiểu hiểu xem mình đã bỏ lỡ điều gì.

Đầu tiên, hãy thay đổi trạng thái của bugs, v.v… chuyển chúng từ hoạt động sang kết thúc, đó là mệnh đề bạn đặt ra khi nghĩ về bug. Hãy dành thời gian để giải thích lý do để ngăn chặn nó sẽ giúp bạn tiết kiệm hàng giờ tẻ nhạt ngồi chỉnh sửa để rồi làm cho khách hàng và quản lý thất vọng. Thay đổi sự ưu tiên của một bug cũng chính là câu

hỏi chung, và chỉ vì nó tầm thường với bạn không có nghĩa là nó không ngăn cản người khác sử dụng sản phẩm.

Thứ hai, đừng khiến việc tìm hiểu về mảng trục trặc bị quá tải bởi lý do cá nhân của bạn. Thay vào đó bạn hãy thử thêm từ “quan trọng” vào trước một chủ đề về một mảng của lỗi có thể giúp bạn dễ dàng thực hiện việc sắp xếp kết quả đến từ một vài bản báo cáo, nhưng đột nhiên chính việc ấy lại trở thành bản copy của các bản báo cáo khác và chắc chắn rằng đó không phải là điều mà bạn mong muốn, hoặc là bạn có thể xoá bớt 1 số mảng ấy để nó phù hợp hơn với mục đích sử dụng ở các bản báo cáo khác. Thay vào đó chúng ta hãy sử dụng một giá trị khác hay một mảng khác để đánh giá lỗi, và tìm hiểu về mục đích của mảng đó để những người khác không phải tự thực hiện lại công việc này.

Chúng ta phải chắc chắn rằng bất kỳ ai cũng có thể dễ dàng phát hiện được lỗi mà cả team chúng ta đang làm việc với nó. Việc này thường có thể được hoàn thành nhờ vào sử dụng một query cụ thể chung. Đồng thời, việc đảm bảo mọi người dùng chung query vô cùng quan trọng và không được cập nhật nó nếu chưa có sự đồng ý đến từ tất cả mọi người trong nhóm.

Cuối cùng hãy luôn ghi nhớ rằng bugs không phải là đơn vị chuẩn của công việc hay của từng dòng code mà chính bugs là đơn vị đo lường chính xác sự nỗ lực tuyệt vời của bạn.

Phần 39: Cải Thiện Code Bằng Cách Loại Bỏ Chúng

Sống tối giản để tận hưởng nhiều hơn. Đó là một câu châm ngôn nhỏ,nhưng nó thật sự đúng trong một số trường hợp.

Một trong những cách cải thiện codebase mà tôi đã làm trong vài tuần qua đó là loại bỏ một số phần của nó.

Chúng tôi đã viết phần mềm dựa trên những nguyên lí XP, bao gồm YAGNI (có nghĩa là, You Aren't Gonna Need It). Bản chất của con người là như vậy, chúng ta chắc chắn đã không đạt được mục tiêu được yêu cầu trong vài nhiệm vụ.

Tôi đã quan sát và nhận ra rằng sản phẩm đã mất quá nhiều thời gian trong việc hoàn thành các nhiệm vụ nhất định - những công việc đơn giản đáng lẽ phải được thực hiện một cách gần như tức thời. Điều này là do chúng đã được thực hiện quá mức; được trang hoàng thêm những chiếc chuông và còi không cần thiết, những thứ mà tại thời điểm đó có vẻ như là một ý tưởng hay.

Thế nên tôi đã đơn giản hóa mã code, cải thiện sự thi hành của sản phẩm, và giảm bớt mức độ phức tạp của toàn bộ code một cách đơn giản hơn bằng cách loại bỏ những tính chất khó chịu từ codebase. Nó thực sự hữu ích, các đơn vị kiểm tra đã nói với tôi rằng tôi chẳng làm hỏng bất cứ thứ gì trong suốt quá trình hoạt động.

Một trải nghiệm đơn giản và hoàn toàn hài lòng.

Vậy thì tại sao những dòng code không cần thiết đó lại xuất hiện ngay từ đầu? Tại sao một người lập trình nào đó lại cảm thấy những dòng code viết thêm ấy hữu dụng, và làm thế nào nó vượt qua được sự đánh giá trước đó hoặc quá trình kết nối? Chắc chắn có một cái gì đó như là:

- Những thứ được thêm có một chút thú vị, và người lập trình muốn viết nó. (Nhắc nhở: Viết code bởi vì nó làm tăng giá trị, chứ không phải vì chúng giải trí).
- Có người nghĩ rằng mã code đó sẽ có ích cho sau này, nên tốt hơn hết là viết chúng ngay bây giờ. (Nhắc nhở: Đó không phải là YAGNI. Đừng viết chúng nếu bạn không sử dụng ngay lập tức).
- Chúng không xuất hiện để đoạn code trở nên "đặc biệt", vì vậy sẽ dễ dàng thực hiện nó hơn là đến gặp khách hàng để xem liệu chúng có thực sự hữu dụng hay không. (Nhắc nhở: Sẽ luôn luôn mất nhiều thời gian để viết và duy trì những dòng code phụ. Và khách hàng thì khá dễ để tiếp cận. Một vài dòng code phụ

theo thời gian sẽ làm quá trình hoạt động tăng trưởng nhanh và trở thành một mảng công việc cần được bảo trì.)

- Những lập trình viên đặt ra những yêu cầu không được ghi chép hay thảo luận để bào chữa cho tính năng bổ sung. Thực ra những nhu cầu đó không hề có thật. (Nhắc nhở: Những lập trình viên không đặt ra những yêu cầu hệ thống mà là khách hàng.)

Ngay bây giờ bạn đang làm việc gì vậy? Có phải tất cả chúng đều cần thiết không?

Phần 40: Hãy cài đặt phần mềm này

Tôi không phải là không có hứng thú về chương trình của bạn

Tôi đang ngập mặt trong một đống công việc với một to-do list dài cả thước. Lý do duy nhất tôi truy cập vào website của bạn là vì tôi nghe nói, từ một nguồn cũng chẳng đáng tin cậy, rằng phần mềm của bạn sẽ giải quyết hết mọi vấn đề của tôi. Vậy nên hãy thứ lỗi nếu tôi có hơi kỹ tính một chút.

Theo cách đọc thông thường, tôi thường đọc tiêu đề và đi tìm dòng chữ được đánh dấu bằng một đường gạch chân màu xanh với nội dung “download now”. Thêm nữa, nếu tôi truy cập trang này bằng trình duyệt Linux và IP đến từ Anh Quốc, tôi hẳn nhiên muốn thấy các sản phẩm với phiên bản dành cho Linux và được cung cấp từ một máy chủ ở Châu Âu. Ngay lúc đó nếu có một hộp thoại tập tin mở ra, tôi chắc chắn sẽ nhấn lệnh tải tập tin xuống trước, rồi sau đó mới đọc các nội dung còn lại.

Chúng ta vẫn cứ hay trình bày dài dòng về lý do, mục đích của mọi thứ mà chúng ta đã thực hiện. Trong khi chỉ cần project của bạn khiến tôi không vừa ý dù chỉ một giây, tôi sẵn sàng vứt nó đi để tìm cái khác. Sự đáp ứng nhu cầu ngay lập tức là quan trọng nhất.

Thử thách đầu tiên là cài đặt. Chớ nghĩ rằng nó không thành vấn đề. Hãy thử đến thư mục tải về xem. Nó có phải toàn là file zip và tar? Bạn đã giải nén bao nhiêu trong số

chúng rồi? Và Có bao nhiêu file thực sự hữu dụng? Nếu bạn cũng nghĩ như tôi, có lẽ chỉ 1/3 chúng thực sự làm được gì đó có ích hơn là chỉ chiếm bộ nhớ.

Mặc dù tôi rất thích sự tiện lợi, nhưng tôi cũng không cho phép ai khác vào nhà tôi mà chưa được mời. Trước khi khi kích hoạt cài đặt tôi muốn biết chính xác nơi bạn sẽ đặt ứng dụng của bạn trên máy tính của tôi. Và tôi cũng muốn nó được cài đặt tinh gọn nhất có thể. Hơn nữa, tôi cũng sẽ không bao giờ cho phép nó được cài đặt nếu như tôi cảm thấy nó không thể bị gỡ bỏ ngay khi tôi thất vọng về nó. Máy tính tôi đang chạy rất ổn, và tôi muốn nó luôn ổn như vậy.

Nếu chương trình của bạn là ứng dụng giao diện người dùng, thì tôi chỉ muốn thấy những thao tác đơn giản và kết quả của chúng. Đừng nói về wizards, tôi chẳng hiểu biết gì về cách nó hoạt động. Điều tôi muốn đơn giản là đọc hoặc ghi file. Tôi không thích phải tạo project mới, import thư viện hay phải khai báo email. Và sau cùng, đừng quên viết hướng dẫn sử dụng.

Nếu chương trình của bạn là một thư viện, tôi sẽ cần tìm trong web page của bạn mục quick start guide. Tôi muốn website của bạn phải trình bày kỹ thuật in ra cụm từ “Hello World” ngắn gọn trong vòng 5 dòng, và phải tường minh, dễ hiểu. Chỉ cần một đoạn script đơn giản thôi, chứ đừng bắt tôi phải điền vào một tệp XML hoặc template nào quá lớn. Hãy nhớ rằng tôi có thể chọn tải về một framework đối thủ của bạn. Và bạn cũng hiểu rằng luôn có một ai đó ngoài kia đang cố gắng trở nên xuất sắc hơn bạn ngoài cộng đồng. Và sau cùng, tất nhiên, đừng quên viết hướng dẫn sử dụng.

Và sẽ luôn có một hướng dẫn sử dụng, đúng chứ?! Cái thứ có chức năng chỉ dẫn tôi bằng ngôn ngữ tôi có thể hiểu.

Nếu mục hướng dẫn sử dụng của bạn giải quyết đúng vấn đề của tôi, tôi sẽ phấn khích vô cùng. Tôi sẽ đọc cách làm thế nào để khởi động các chức năng của nó trong một tâm trạng vui vẻ. Tôi có thể thư giãn và làm một tách trà – vâng, vì tôi người Anh mà!! – sau đó thử các ví dụ và học cách dùng sản phẩm của bạn. Nếu vấn đề của tôi được giải quyết, tôi sẽ gửi bạn lời cảm ơn, tôi còn có thể sẽ báo cáo bạn một số lỗi hay đề nghị những tính năng mới. Tôi thậm chí sẽ khoe với bạn bè tôi rằng phần mềm của bạn

tuyệt với cỡ nào, và chẳng bao giờ bận tâm đến những phần mềm khác của đối thủ. Tất cả là vì bạn đã hoàn toàn chinh phục trải nghiệm đầu tiên của tôi. Và tôi có lý do gì để hồ nghi bạn được?

Phần 41: Giao tiếp giữa các tiến trình(*) ảnh hưởng đến thời gian phản hồi của ứng dụng

Thời gian phản hồi là một yếu tố quan trọng ảnh hưởng đến khả năng sử dụng phần mềm. Một điều gây khó chịu trong quá trình sử dụng là chờ đợi phản hồi từ hệ thống, đặc biệt là khi sự tương tác với phần mềm liên quan cần lặp đi lặp lại. Chúng tôi cảm thấy phần mềm đang tốn nhiều thời gian và nó gây ảnh hưởng đến năng suất. Tuy vậy, nguyên nhân của thời gian phản hồi kém là không có đủ sự quan tâm từ nhà phát triển, đặc biệt là trong các ứng dụng hiện đại. Nhiều tài liệu về quản lý hiệu suất vẫn tập trung vào cấu trúc dữ liệu và thuật toán, các vấn đề này có thể khác đi trong một số trường hợp nhưng ít có khả năng chi phối hiệu suất trong các ứng dụng doanh nghiệp đa tầng hiện đại (multi-tier enterprise applications).

Khi hiệu suất là một vấn đề trong nhiều ứng dụng, kinh nghiệm của tôi là việc kiểm tra các cấu trúc dữ liệu và thuật toán không thích hợp để tìm kiếm giải pháp. Thời gian phản hồi phụ thuộc nhiều vào số lượng các giao tiếp từ xa giữa các tiền trình (IPC) để hồi đáp lại một yêu cầu. Mỗi giao tiếp từ xa giữa các tiến trình ảnh hưởng một phần nhỏ vào tổng thời gian phản hồi nhưng sẽ trở nên nghiêm trọng khi các quá trình này phát sinh liên tục.

Một ví dụ điển hình là Ripple Loading trong phần mềm có sử dụng object-relational mapping (kỹ thuật chuyển đổi dữ liệu giữa các hệ thống). Ripple Loading mô tả việc thực hiện liên tục các lệnh gọi cơ sở dữ liệu để chọn các dữ liệu cần thiết qua đó xây dựng một biểu đồ cho các đối tượng (như Lazy Load trong cuốn sách “Martin Fowler's Patterns of Enterprise Application Architecture”). Khi cơ sở dữ liệu của khách hàng là một máy chủ ứng dụng trung cấp để hiển thị một trang web, các yêu cầu từ cơ sở dữ

liệu được thực hiện liên tục trong luồng đơn. Độ trễ của mỗi quá trình đơn lẻ sẽ ảnh hưởng đến thời gian phản hồi tổng thể. Ngay cả khi các câu lệnh từ cơ sở dữ liệu chỉ mất 10ms, một trang yêu cầu 1000 câu lệnh (điều này không thật sự phổ biến) sẽ hiển thị mất khoảng 10 giây. Một số ví dụ khác như dịch vụ web, yêu cầu HTTP từ trình duyệt web, dẫn các đối tượng tương phản (distributed object invocation), yêu cầu trả lời tin nhắn và tương tác bằng lưới dữ liệu (data-grid interaction) qua các giao thức mạng tùy chỉnh. Cần càng nhiều IPC từ xa để trả lời một yêu cầu, thời gian phản hồi sẽ càng nhiều.

Có một số cách hay để giảm số lượng IPC từ xa với mỗi yêu cầu. Một cách trong số đó là áp dụng phân tích cú pháp, tối ưu hóa giao diện giúp thay đổi dữ liệu chính xác cho mục đích hạn chế tối đa các giao tiếp. Một cách khác là song song hóa (parallelize) các IPC ở bất cứ vị trí nào có thể, nhờ đó thời gian phản hồi tổng thể chủ yếu được tác động bởi IPC có tốc độ xử lí yêu cầu nhanh nhất. Cách thứ ba là lưu trữ kết quả của các IPC trước đó, để có thể tránh được sự ảnh đến cache cục bộ qua tính năng của các IPC.

Khi bạn đang thiết kế một ứng dụng, hãy chú ý đến số lượng IPC trong mỗi yêu cầu. Khi phân tích các ứng dụng có hiệu suất kém, tôi thường nhận ra rằng tỉ lệ giữa các IPC và những yêu cầu là 1/1000. Việc giảm tỉ lệ này, cho dù bằng cách lưu vào cache hoặc song song hóa hay một kỹ thuật nào khác, đều sẽ mang lại nhiều lợi ích hơn so với việc thay đổi cấu trúc dữ liệu hoặc điều chỉnh thuật toán sắp xếp.
(*) Inter-Process Communication (IPC)

Phần 42: Hãy giữ cho thiết kế thật sạch sẽ

Bạn đã từng bao giờ nhìn vào một bản danh sách của người biên soạn về việc cảnh báo độ dài của một đoạn code không tốt và nghĩ rằng: “Mình thật sự nên làm gì đó cho việc này... Nhưng mà bây giờ mình lại không có thời gian?” Mặt khác, bạn đã bao giờ nhìn thấy một cảnh báo xuất hiện trong trình biên dịch và sẵn sàng sửa nó?

Khi tôi bắt đầu một project mới lại từ đầu, sẽ không có bất kì cảnh báo, cản trở, vấn đề nào xuất hiện. Nhưng với tư cách là một người phát triển code base, nếu không để ý, những cản trở, cảnh báo, và những vấn đề có thể sẽ xuất hiện và dễ mất kiểm soát. Khi mà có quá nhiều vấn đề cần giải quyết, thì để tìm được cảnh báo cần thiết trong hàng trăm cảnh báo khác nhau mà thậm chí mình không quan tâm là một điều thực sự khó khăn với chúng ta.

Để làm cho những cảnh báo hữu dụng trở lại, tôi đã thử một chính sách "không khoan nhượng" cho các cảnh báo trong thiết kế. Kể cả cảnh báo đó không quá nghiêm trọng, tôi vẫn thực hiện cách làm đó. Nếu cảnh báo đó không quan trọng, nhưng có sức ảnh hưởng thì tôi sẽ sửa nó. Khi người biên soạn cảnh báo về khả năng bị phản đối, tôi sẽ khắc phục nguyên nhân, thậm chí tôi biết rằng vấn đề sẽ chẳng bao giờ xuất hiện trong sản phẩm cả. Nếu như tài liệu đã được nhúng (Javadoc hoặc Similar) đề cập đến các thông số đã bị xóa hoặc đổi tên, thì tôi sẽ xóa bỏ những tài liệu đó.

Nếu như có thứ gì đó tôi không thật sự quan tâm và nó cũng không quan trọng lắm với mình, tôi sẽ yêu cầu team của mình thay đổi chính sách cảnh báo của chúng tôi. Ví dụ, đưa ra tài liệu về các thông số và đưa giá trị của hệ thống trở lại, trong nhiều trường hợp, không làm tăng thêm bất cứ giá trị nào cả. Vì vậy nó không thể là cảnh báo nếu nó đang khiếm khuyết. Hoặc là nâng cấp lên một phiên bản mới của ngôn ngữ lập trình có thể sẽ làm cảnh báo được đưa ra một cách tốt hơn. Ví dụ như khi Java 5 giới thiệu các chủng loại, tất cả loại code đã cũ không ghi rõ các loại thông số sẽ bị cảnh báo. Đây là một phương thức cảnh báo mà tôi không hề muốn đối diện. Một loạt các cảnh báo không phù hợp với thực tế sẽ không giúp được bất cứ ai.

Bằng cách hãy chắc chắn rằng thiết kế luôn được sạch sẽ, tôi sẽ không phải quyết định rằng một cảnh báo là không thích hợp mỗi lúc gặp phải nó. Phớt lờ đi mọi chuyện là một công việc cần phải suy nghĩ nhiều, và tôi cần thoát khỏi tất cả những công việc không cần thiết đó. Một thiết kế sạch sẽ cũng giúp cho những người đảm nhận công việc của tôi làm việc một cách dễ dàng hơn. Nếu tôi để mặc những cảnh báo, người khác sẽ phải vất vả để biết cái gì thích hợp và cái gì không. Hoặc nhiều khả năng, họ sẽ phớt lờ tất cả cảnh báo, kể cả những cái quan trọng.

Các cảnh báo từ thiết kế của bạn thực sự rất hữu ích, bạn chỉ cần đừng quan tâm đến những thứ không cần thiết để chú ý đến chúng. Đừng chờ đợi một lần đại trùng tu. Khi một thứ gì đó mà bạn không muốn thấy xuất hiện, hãy đối phó với nó lập tức. Hoặc hãy sửa lại từ nguồn gốc của các cảnh báo, chặn các cảnh báo, sửa lại cách cảnh báo từ công cụ của bạn. Đảm bảo thiết kế được sạch sẽ không chỉ là giữ nó không bị ràng buộc bởi các lỗi biên dịch hay lỗi kiểm tra, các cảnh báo cũng rất quan trọng và các phần code quyết định cũng thế.

Phần 43: Biết cách sử dụng các công cụ dòng lệnh

Ngày nay, nhiều công cụ phát triển phần mềm được đóng gói dưới dạng Môi trường phát triển tích hợp (IDE).Visual Studio của Microsoft và phần mềm mã nguồn mở Eclipse là hai ví dụ phổ biến mặc dù bên cạnh đó có rất nhiều phần mềm khác.Có rất nhiều điều để thích về IDE. Không chỉ dễ sử dụng, chúng còn giúp lập trình viên suy nghĩ về rất nhiều chi tiết nhỏ liên quan đến quá trình xây dựng.

Dễ sử dụng, tuy nhiên, có nhược điểm của nó. Thông thường, khi một công cụ dễ sử dụng, đó là vì công cụ đang đưa ra quyết định cho bạn và tự động thực hiện nhiều việc, đằng sau hậu trường. Do đó, nếu IDE là môi trường lập trình duy nhất mà bạn từng sử dụng, bạn có thể không bao giờ hiểu đầy đủ những gì công cụ của bạn đang thực sự làm. Bạn bấm vào một nút, một số điều kì diệu xảy ra và một tệp thực thi xuất hiện trong thư mục dự án.

Bằng cách làm việc với các công cụ xây dựng dòng lệnh, bạn sẽ học được thêm rất nhiều về những gì các công cụ đang làm khi dự án của bạn đang được xây dựng. Viết các tập tin của riêng bạn sẽ giúp bạn hiểu tất cả các bước (biên dịch, lắp ráp, liên kết, v.v.) mà đi vào việc xây dựng một tập tin thực thi.Thử nghiệm với nhiều tùy chọn dòng lệnh cho các công cụ này cũng là một kinh nghiệm giáo dục có giá trị. Để bắt đầu với việc sử dụng các công cụ xây dựng dòng lệnh, bạn có thể sử dụng các công cụ dòng lệnh nguồn mở như GCC hoặc bạn có thể sử dụng các công cụ được cung cấp với IDE

bản quyền của bạn. Xét cho cùng, một IDE được thiết kế tốt chỉ là một giao diện đồ họa cho một bộ công cụ dòng lệnh.

Ngoài việc cải thiện hiểu biết của bạn về quá trình xây dựng, có một số tác vụ có thể được thực hiện dễ dàng hoặc hiệu quả hơn với các công cụ dòng lệnh so với IDE.Ví dụ, các khả năng tìm kiếm và thay thế được cung cấp bởi các tiện ích grep và sed thường mạnh hơn các khả năng tìm thấy trong IDE. Các công cụ dòng lệnh vốn đã hỗ trợ kịch bản, cho phép tự động hóa các tác vụ như sản xuất các bản dựng hàng ngày theo lịch trình, tạo nhiều phiên bản của dự án và chạy các bộ thử nghiệm. Trong IDE, loại tự động hóa này có thể khó thực hiện hơn (nếu không thể thực hiện được) vì các tùy chọn xây dựng thường được chỉ định bằng hộp thoại GUI và quá trình xây dựng được gọi bằng một cú click chuột. Nếu bạn không bao giờ rời khỏi IDE, bạn thậm chí có thể không nhận ra rằng các loại tác vụ tự động này có thể thực hiện được.
Nhưng chờ đã. IDE không tồn tại để làm cho việc phát triển dễ dàng hơn và để cải thiện năng suất của lập trình viên ư? Vâng, chuẩn luôn. Gợi ý được trình bày ở đây không phải là bạn nên ngừng sử dụng IDE. Gợi ý là bạn nên "nhìn dưới mui xe" và hiểu IDE của bạn đang làm gì cho bạn. Cách tốt nhất để làm điều đó là học cách sử dụng các công cụ dòng lệnh. Sau đó, khi bạn quay lại sử dụng IDE của mình, bạn sẽ hiểu rõ hơn nhiều về những gì nó đang làm cho bạn và cách bạn có thể kiểm soát quá trình xây dựng. Mặt khác, khi bạn thành thạo việc sử dụng các công cụ dòng lệnh và trải nghiệm sức mạnh và tính linh hoạt mà chúng cung cấp, bạn có thể thấy rằng bạn thích dòng lệnh hơn IDE.

Phần 44: Biết rõ nhiều hơn hai ngôn ngữ lập trình

Suy nghĩ của những người lập trình lâu năm rằng từ lâu họ đã biết chuyên môn lập trình có liên quan trực tiếp đến số lượng các cơ chế lập trình khác nhau mà họ cảm thấy thỏa mái khi làm việc với nó. Nó không chỉ là biết hay chỉ là biết một ít mà thực sự có thể làm việc với nó.

Mỗi lập trình viên đều bắt đầu với một ngôn ngữ lập trình. Nó có ảnh hưởng lớn đến cách mà lập trình viên tư duy về làm phần mềm. Bất kể có bao nhiêu năm kinh nghiệm lập trình với ngôn ngữ lập trình đấy, nếu như họ chỉ dừng lại ở việc học mỗi một ngôn ngữ họ sẽ chỉ
biết có mỗi mình nó. Một lập trình viên mà chỉ biết một ngôn ngữ lập trình duy nhất họ sẽ bị giới hạn bởi tư duy của ngôn ngữ lập trình đấy.

Một lập trình viên mà học hai ngôn ngữ lập trình sẽ thách thức hơn, đặc biệt là nếu nó có cơ chế khác so với ngôn ngữ đã học. C, Pascal, Fortran, tất cả đều là ngôn ngữ lập trình cơ bản. Chuyển từ Fortran sang C thì giới thiệu được một ít căn bản và không nhiều thử thách. Ta từ C hay Fortan chuyển qua học C++ thì sẽ có những thách thức căn bản hơn trong cách xử lí với các chương trình. Chuyển từ C++ sang Haskell là một bước thay đổi đáng kể do đó nó có một thách thức thấy rõ hơn. Chuyển từ C sang Prolog là một thử thách rất khó khăn.

Chúng ta có thể liệt kê một số cơ chế lập trình: procedural, object-oriented, functional, logic, dataflow,..
Chuyển từ mô hình này sang mô hình khác tạo nên một thử thách lớn nhất.

Tại sao những thử thách này lại tốt? Nó là cách để làm chúng ta suy nghĩ về việc triển khai các thuật toán và các cách thức và chuẩn của mô hình mà chúng ta áp dụng. Đặc biệt, Trộn lẫn các kiến thức học được với nhau là cốt lõi của thành công. Cách thức tìm ra giải pháp cho một vấn đề trong một ngôn ngữ có thể không thể thực hiện trong một ngôn ngữ khác. Cố gắng giải quyết vấn để bằng cách thức của ngôn ngữ này chuyển sang giải quyết với ngôn ngữ khác dạy cho chúng ta về một vấn đề mà cả hai ngôn ngữ đang giải quyết.

Trộn lẫn các kiến thức lập trình của nhiều ngôn ngữ lập trình với nhau có tác dụng rất lớn. Có lẽ rõ ràng nhất là việc sử dụng ngày càng nhiều các phương thức biểu đạt khai báo ttrong các hệ thống bằng các ngôn ngữ mệnh lệnh. Bất kì ai thành thạo lập trình hướng thủ tục đều có thể dễ dàng áp dụng phương pháp khai báo ngay cả khi sử dụng ngôn ngữ như C. Sử dụng phương pháp khai báo thường dẫn đến các chương trình

ngắn hơn và dễ hiểu hơn. C++ là một ví dụ cho sự hỗ trợ hết mình cho lập trình tổng quát, cái mà hầu như luôn cần một phương thức khai báo.

Hậu quả của tất cả những điều này là nó khiến mọi lập trình viên phải có kỹ năng lập trình tốt trong ít nhất hai mô hình cơ chế khác nhau, và hợp lí nhất là thực hiện ít nhất năm đề cập ở trên. Lập trình viên nên luôn luôn hứng thú với việc học ngôn ngữ lập trình mới, tốt nhất là từ một mô hình xa lạ. Ngay cả khi công việc thường ngày luôn sử dụng một ngôn ngữ, sự thành thạo của việc sử dụng một ngôn ngữ đó là khi một người có thể áp dụng các cơ chế của ngôn ngữ lập trình khác và không đánh giá thấp nó. Nhà tuyển dụng nên chú ý và cho phép sử dụng ngân sách đào tạo nhân viên của họ học các ngôn ngữ lập trình hiện không sử dụng làm chính trong công việc mục đích để tăng độ tư duy trong công việc và sử dụng các ngôn ngữ lập trình.

Mặc dù đó là một sự khởi đầu, Khóa đào tạo một tuần không đủ để học một ngôn ngữ mới: nó thường mất tầm một vài tháng để sử dụng được tốt, ngay cả khi ngoài giờ, để có được kiến thức làm việc được với một ngôn ngữ. Nó là cách thức sử dụng, không chỉ là cú pháp và các cơ chế mô hình tính toán, đó là những yếu tố quan trọng.

Phần 45: Thành thạo IDE của bạn

Vào thập niên 80, khi mà những môi trường lập trình thường không có gì sánh bằng những trình biên soạn text. Làm nổi bật lên các cú pháp, là việc mà chúng ta công nhận bấy giờ là một điều xa xỉ mà không phải ai cũng có thể làm được. Những chiếc máy in để định dạng code một cách đẹp đẽ thường là công cụ bên ngoài được sử dụng để sửa khoảng cách. Debuggers cũng là những chương trình rời rạc được chạy để bỏ qua code của chúng ta, nhưng lại với một loạt các thao tác phím bí mật.

Trong suốt những năm thuộc thập niên 90, các công ty bắt đầu nhận ra được tiềm năng kinh tế mà họ có thể chuyển hóa từ việc trang bị thêm các lập trình viên có công cụ tốt hơn và hữu ích hơn. Môi Trường Thiết Kế Hợp Nhất (IDE) đã tập hợp các tính năng được chỉnh sửa trước đó với một bộ biên dịch, một debugger, một máy in tốt và

các công cụ khác. Trong khoảng thời gian đó, các menu và chuột máy tính cũng trở nên nổi tiếng, điều đó có nghĩa rằng sẽ chẳng bao lâu nữa các nhà phát triển sẽ cần nghiên cứu về các chìa khóa bí mật để sử dụng chương trình biên soạn của họ. Họ có thể dễ dàng lựa chọn lệnh từ menu.

Đến thế kỉ 21, các IDE đã thực sự trở nên phổ biến đến mức chúng được tặng miễn phí bởi các công ty muốn giành lấy thị phần trong các lĩnh vực khác. IDE hiện đại đã được trang bị một hệ thống các tính năng tuyệt vời. Tính năng mà tôi rất yêu thích là tự động tái cấu trúc, đặc biệt là Extract Method, đó là nơi mà tôi có thể lựa chọn và biến đổi một mảng code theo một thứ tự nhất định. Công cụ tái cấu trúc sẽ biến tất cả thông số cần được duyệt thành một hệ thống thứ tự, là thứ sẽ làm cho hoạt động sửa đổi code trở nên cực kì dễ dàng. IDE của tôi còn phát hiện các mảng code khác có thể được thay thể bởi hệ thống này và cũng đưa ra câu hỏi liệu rằng tôi có muốn thay thế chúng hay không.

Một tính năng ngạc nhiên nữa của những IDE hiện đại đó chính là khả năng ép buộc các kiểu quy tắc trong phạm vi một công ty. Ví dụ, trong ngôn ngữ Java, một vài lập trình viên đã làm tất cả các tham số cuối cùng (đó là việc mà tôi cho là khá tốn thời gian). Tuy nhiên, kể từ khi họ có một kiểu quy tắc riêng, tất cả tôi cần để theo dõi nó đó chính là cài nó vào IDE của tôi: Tôi sẽ nhận được một cảnh báo cho bất cứ tham số nào chưa đến cuối cùng. Các kiểu quy tắc này cũng được sử dụng để tìm kiếm các bugs có khả năng xảy ra, ví dụ như là so sánh đối tượng được autoboxed để lấy đẳng thức tham chiếu, ví dụ như sử dụng == trên các giá trị gốc đã được autoboxed vào các đối tượng tham chiếu.

Không may là các IDE hiện đại không đòi hỏi chúng ta nỗ lực đầu tư để học hỏi cách sử dụng chúng. Khi tôi lập trình C lần đầu tiên trên hệ điều hành Unix, tôi đã phải dành khá nhiều thời gian để học về cách thức chương trình vi hoạt động, vì “đường cong học tập” của nó khá dốc. Khoảng thời gian này đã được trả lại hết sau nhiều năm. Tôi thậm chí vẫn đang gõ bản nháp của chủ đề này với vi. IDE hiện đại có một “ đường cong học tập” rất chậm rãi, là thứ có tác động chúng rằng ta sẽ không bao giờ tiến bộ vượt quá mức sử dụng cơ bản của các công cụ.

Bước đầu tiên của tôi trong việc nghiên cứu về IDE là học thuộc lòng các phím tắt. Khi đặt tay lên bàn phím để gõ code, tôi sẽ nhấn tổ hợp phím Ctrl+Shift+I để inline các bản lưu có thể thay đổi khỏi việc ngắt dòng, trong khi chuyển sang điều khiển menu thông qua chuột để ngắt dòng. Những sự gián đoạn dòng đó dẫn đến các sự chuyển đổi ngữ cảnh không cần thiết và sẽ làm giảm bớt năng suất của tôi nếu tôi cố gắng để làm mọi thứ theo cách chậm chạp. Quy định giống nhau cũng áp dụng cho các kĩ thuật gõ phím. Hãy học hỏi để đạt đến sự mẫu mực, bạn sẽ không phải hối tiếc về khoảng thời gian đã bỏ ra cho việc đó.
Cuối cùng, với tư cách là một lập trình viên, chúng ta có thời gian chứng tỏ rằng công cụ phát trực tuyến của Unix có thể giúp ta vận dụng cho code. Ví dụ trong lúc xem xét lại các đoạn code, tôi đã để ý rằng các lập trình viên đã đặt tên rất nhiều class giống nhau, và ta có thể tìm được chúng một cách dễ dàng thông qua các công cụ như find, sed, sort, uniq và grep, giống như mô tả dưới đây:

find . -name ".java" | sed 's/.*\///' | sort | uniq -c | grep -v "^ *1 " | sort –r*

Chúng ta đều biết rằng một người thợ sửa ống nước đến nhà có thể sử dụng thành thạo đèn hàn của anh ấy. Hãy dành ra một ít thời gian để học cách trở nên thành thạo với IDE của mỗi chúng ta.

Phần 46: Nhận thức giới hạn của bản thân

“Làm người phải biết giới hạn của bản thân”_ Dirty Harry

Mọi nguồn lực của bạn đều hữu hạn. Bạn chỉ có thật nhiều thời gian và tiền bạc để bạn thực hiện công việc của bạn, kể cả thời gian và tiền bạc giúp bạn giữ cho kiến thức, kỹ năng và dụng cụ lúc nào cũng nhạy bén. Bạn chỉ có thể làm việc chăm chỉ, thật nhanh, thật thông minh, và thật bền bỉ. Dụng cụ mà bạn có chính là những trợ thủ đắc lực

nhất. Và chính mục tiêu của bạn cũng mạnh mẽ không kém. Chính vì vậy bạn phải tôn trọng giới hạn của những nguồn tài nguyên mà bạn có.

Vậy chúng ta thực hiện nó bằng cách nào? Đó là hiểu bản thân, hiểu đồng nghiệp, hiểu ngân sách của bạn và cuối cùng chính là thấu hiểu dụng cụ của bạn. Điển hình như cách mà một kỹ sư phần mềm hiểu biết và sự độ phức tạp không gian và thời gian của thuật toán và cấu trúc dữ liệu, cũng như cấu trúc và hiệu năng riêng biệt của hệ thống. Công việc của bạn chính là tạo nên sự gắn bó chặt chẽ tối ưu giữa phần mềm và hệ thống.

Sự phức tạp về không gian và thời gian thực hiện của một thuật toán được xác định bởi hàm O(f(n)) (với n là kích thước đầu vào của chương trình) là sự dự đoán về thời gian hay không gian lưu trữ của thuật toán khi n tiến đến vô cực. Một số lớp quan trọng của hàm f(n) bao gồm: ln(n), n, n ln(n), n^e và cuối cùng là e^n. Khi tổng hợp kết quả từ việc thử nghiệm hàm này và biểu diễn bằng đồ thị chúng ta sẽ nhận thấy sự khác biết rõ ràng, khi n ngày càng lớn, O(ln(n)) sẽ cho kết quả vô cùng nhỏ khi so sánh với O(n) hay O(n ln(n)), và còn nhỏ hơn rất nhiều lần so với O(n^e) và O(e^n). Khi Sean Parent thử với mọi n lớn nhất có thể đạt được thì tất cả các lớp trả về một kết quả gần như hằng số hay gần tuyến tính hay gần như vô cùng lớn.

	Access time	Capacity
Register	<1ns	64b
Cache line		64B
L1 cache	1ns	64KB
L2 cache	4ns	8MB
RAM	20ns	32GB
Disk	10ms	10TB
LAN	20ms	>1PB
Internet	100ms	>1ZB

Phân tích độ phức tạp chính là một trong những phần của máy tính trừu tượng, nhưng phần mềm lại chạy trên những cỗ máy thật. Những hệ thống máy tính hiện đại được chế tạo dựa trên lý thuyết vật lý cùng với những hệ thống trừu tượng, bao gồm cả thời

gian xử lý ngôn ngữ, CPUs, bộ nhớ cache, RAM, ổ cứng, và mạng máy tính. Bảng trên cho thấy giới hạn của thời gian truy cập nhiên và khả năng lưu trữ của một server máy tính tiêu biểu.

Chúng ta nhận thấy tốc độ truy cập tỉ lệ thuận với khả năng lưu trữ, khi khả năng lưu trữ càng lớn thời gian truy cập của chúng ngày chậm. Bộ nhớ cache và toàn bộ phần trước nó đều được tận dụng một cách triệt để trong hệ thống của chúng ta và giảm thiểu tối đa các phương tiện còn lại tuy nhiên điều này chỉ có thể thực hiện được khi những lượt truy cập đấy có thể dự đoán được. Và khi thiếu cache hệ thống thường sẽ dừng lại. Lấy ví dụ đơn giản, nếu chúng ta kiểm tra ngẫu nhiên từng byte trên một ở cứng có thể mất đến 32 năm, kể cả kiểm tra ngẫu nhiên mọi byte trong RAM có thể tốn đến 11 phút. Lượng truy cập ngẫu nhiên là không thể đoán trước được. Chính vì thế chúng ta có một phương pháp thay thế khác đó chính là truy cập lại các phần tử đã được sử dụng và truy cập chúng một cách tuần tự.

Và thuật toán và cấu trúc dữ liệu cho thấy chúng hiệu quả như thế nào trong việc tận dụng cache. Lấy ví dụ:

- Tìm kiếm tuyến tính có công dụng tốt trong việc tìm kiếm một giá trị nào đó nhưng lại cần đến O(n) phép so sánh.
- Tìm kiếm nhị phân trong một mảng phần tử đã được sắp xếp chỉ cần O(log(n)) phép so sánh.
- Cây van Emde Boas tìm kiếm và thuật toán Cache-Oblivious chỉ cần O(log(log(n))).

Số phần tử	Thời gian tìm kiếm (ns)		
	tuyến tính	nhị phân	vEB
8	50	90	40
64	180	150	70
512	1200	230	100
4096	17000	320	160

Vậy chúng ta nên sử dụng thuật toán nào? Dựa trên số liệu thống kê và đo đạc được bảng trên thể hiện thời gian cần thiết tìm kiếm trong một mảng số nguyên 64 bit bằng ba phương pháp kể trên. Cá nhân tôi cho rằng:

- Tìm kiếm tuyến tính có thể áp dụng cho mảng nhỏ, nhưng nó dần trở nên rất lâu với những mảng lớn hơn.
- Còn cây van Emde Boas chiến thắng tất cả, nhờ vào sự dự đoán trước lượt truy cập.

“Bạn phải trả giá cho lựa chọn của bản thân mình” _ Punch

Phần 47: Nắm rõ cam kết tiếp theo của bản thân

Tôi đã từng thử hỏi các lập trình viên rằng họ đang làm gì vào lúc đó để xem họ trả lời như thế nào. Người đầu tiên nói: “ Tôi đang tái cấu trúc lại những phương thức ”. “ Tôi đang thêm vào một vài thông số cho hoạt động của trang web ”, người tiếp theo trả lời. Người thứ ba bộc bạch: “ Tôi đang làm việc trên lịch sử người dùng ”.

Trông có vẻ như hai người đầu tiên đã quá tập trung vào các chi tiết trong công việc của họ, trong khi người thứ ba đã nhìn ra được vấn đề lớn hơn, hắn đã để ý rất tốt. Tuy nhiên, khi tôi đặt ra câu hỏi họ sẽ cam kết khi nào và những gì, mọi chuyện thay đổi một cách đột ngột. Hai người đầu tiên đã “clear” những files liên quan và hoàn thành chúng trong vòng hơn một tiếng đồng hồ. Nhưng lập trình viên thứ ba đã nói như sau: “Ồ, tôi đoán là tôi sẽ sẵn sàng trong vòng vài ngày, có thể tôi sẽ thêm một vài class và thay đổi các dịch vụ theo cách nào đó. ”

Rõ ràng là hai người đầu tiên có tầm nhìn mục tiêu tổng quát, họ đã lựa chọn những nhiệm vụ mà họ nghĩ rằng nó sẽ hiệu quả hơn, và hoàn toàn có thể hoàn thành trong vòng chưa đến hai tiếng đồng hồ. Một khi họ đã hoàn tất những công việc đó, họ sẽ có thể lựa chọn một tính năng mới mẻ hoặc tái cấu trúc để công việc được tiếp tục. Tất cả

code được viết đã được hoàn thành như thế bởi một một mục đích rất rõ ràng và một mục tiêu có giới hạn, khả thi trong tâm trí người lập trình.

Còn lập trình viên thứ ba không thể phân tích được vấn đề và đã làm việc trên mọi khía cạnh cùng một lúc. Anh ấy không hề biết rằng mình sẽ làm gì, cơ bản như thực hiện lập trình speculative hay hy vọng rằng đến một lúc nào đó anh ta có thể thực hiện cam kết. Gần như chắc chắn rằng những đoạn code được viết vào thời điểm bắt đầu của đợt lâu dài này sẽ không phù hợp với solution được xuất ra cuối cùng.

Liệu hai lập trình viên đầu tiên sẽ làm như thế nào nếu những nhiệm vụ của họ tiêu tốn nhiều hơn hai tiếng đồng hồ? Sau khi nhận ra mình đã dành ra quá nhiều, có lẽ họ sẽ vứt bỏ đi những thay đổi, xác định rõ những nhiệm vụ nhỏ hơn, và bắt đầu lại. Thay vì tiếp tục làm việc một cách thiếu tập trung và dẫn đến việc những đoạn code đầu cơ xâm nhập vào kho lưu trữ, thì những thay đổi sẽ bị loại bỏ, nhưng những hiểu biết sâu sắc vẫn sẽ được giữ lại và phát huy.

Người lập trình viên thứ ba có thể sẽ tiếp tục “phỏng đoán” và cố gắng hết sức để biến những thay đổi của anh ta thành một thứ có thể cam kết được. Rốt cuộc, chúng ta vẫn không thể vứt bỏ các thay đổi code mà mình đã thực hiện, điều đó thực sự rất lãng phí. Nhưng không may là việc giữ lại code có thể sẽ khiến cho những code lạ và không rõ ràng dễ dàng xâm nhập vào kho lưu trữ.

Tại một vài thời điểm, kể cả những lập trình viên “cam kết tập trung” vẫn không thể tìm thấy thứ gì hữu ích có thể giúp họ hoàn thành công việc sau hai giờ đồng hồ. Sau đó, họ sẽ chuyển sang chế độ speculative, viết một vài đoạn code và vứt bỏ những thay đổi bất cứ khi nào những insight đưa họ về đúng hướng. Ngay cả những pha hack trông như chả có cấu trúc gì cũng có mục đích của nó: Nghiên cứu về code để có thể xác định nhiệm vụ của mình – đó thực sự là một bước đi vô cùng hiệu quả.

Phải biết rõ được cam kết tiếp theo của bạn là gì, nếu không làm được, hãy vứt bỏ hết những thay đổi, sau đó xác định một nhiệm vụ mới mà các bạn tin tưởng. Hãy thực hiện thực nghiệm speculative bất cứ khi nào cần, nhưng đừng để bị rơi vào mode đó

một cách thiếu nhận biết và đừng để các cam kết “phỏng đoán” “rơi” vào kho lưu trữ của mình!

Phần 48: Dữ liệu liên kết lớn thuộc về cơ sở dữ liệu

Nếu chương trình ứng dụng của bạn chuẩn bị xử lý một loạt các tài nguyên dữ liệu rộng lớn và liên kết, đừng chần chừ mà hãy đặt nó vào trong cơ sở dữ liệu quan hệ. Trong quá khứ RDBMS(1) (Hệ Quản trị Cơ Sở Dữ Liệu Quan Hệ) từng được coi như một con “linh thú” vừa đắt tiền, khan hiếm, phức tạp và rất khó sử dụng. Nhưng rồi điều đó cũng không còn quan trọng nữa. Hiện nay rất dễ dàng để tìm kiếm các hệ thống RDBMS, giống như là mọi hệ thống bạn đang sử dụng luôn sẵn có một hoặc hai RDBMS đã được cài đặt trong đó vậy. Một vài RDBMS có tiềm năng, như MySQL(2) và PostgreSQL(3) là các phần mềm mã nguồn mở, vì thế nên vấn đề về bản quyền không thành vấn đề. Hơn cả vậy, cái được gọi là Hệ Thống Nhúng Cơ Sở Dữ Liệu có thể được liên kết bằng các library trực tiếp vào chương trình ứng dụng của bạn và hầu như không đòi hỏi phải thiết lập hay quản lý – hai hệ thống nguồn mở đáng chú ý là SQLite(4) và HSQLDB(5), những hệ thống đó thực sự rất có hiệu quả.

Nếu dữ liệu các ứng dụng của bạn lớn hơn dung lượng RAM của hệ thống, một bảng mục của RDBMS sẽ đưa ra các mệnh lệnh quan trọng nhanh hơn tập hợp library map của bạn, đồng thời giải phóng bộ nhớ ảo. Dịch vụ cơ sở dữ liệu hiện đại có thể dễ dàng phát triển với nhiều nhu cầu. Với sự chăm sóc chu đáo, bạn có thể phát triển quy mô một Hệ Thống Nhúng Cơ Sở Dữ Liệu thành một Hệ Thống Cơ Sở Dữ Liệu lớn hơn khi cần. Sau này bạn có thể chuyển đổi từ một nguồn mở miễn phí sang hệ thống độc quyền được hỗ trợ tốt và mạnh mẽ hơn.

Một khi bạn hiểu rõ về SQL, viết ứng dụng cơ sở dữ liệu trung tâm sẽ là một việc rất thú vị. Sau khi đã đặt dữ liệu được chuẩn hóa chính xác của mình và cơ sở dữ liệu, thật dễ dàng để trích xuất thông tin một cách hiệu quả với truy vấn SQL mà không cần

viết bất cứ đoạn code phức tạp nào. Tương tự vậy, một lệnh đơn SQL có thể đưa ra các thay đổi dữ liệu phức tạp. Với những sự sửa đổi một lần, nó thay đổi cách bạn sắp xếp dữ liệu liên tục, thậm chí bạn không cần phải viết code, chỉ cần kích hoạt trực tiếp giao diện SQL của cơ sở dữ liệu. Sự giống nhau về giao diện cũng cho phép bạn thực hiện với các truy vấn và vượt qua trình chỉnh sửa biên dịch ngôn ngữ lập trình thông thường.

Một lợi ích khác của việc dựa vào code trong một RDBMS sẽ liên quan đến việc xử lý các mối quan hệ giữa các thành phần dữ liệu của bạn. Có thể miêu tả các ràng buộc nhất quán trên dữ liệu của mình bằng cách khai báo, tránh hiện tượng Dangling Pointers (6) trong trường hợp lỡ quên cập nhật dữ liệu của mình. Ví dụ, ta hiểu rằng khi một người dùng bị “ban”, các tin nhắn được gửi bởi người dùng đó cũng sẽ bị loại bỏ.

Bạn cũng có thể tạo các đường dẫn hiệu quả giữa các đối tượng được lưu trữ trong cơ sở dữ liệu bất cứ khi nào mình muốn bằng cách tạo một bản mục lục. Việc đưa ra các lần tái cấu trúc đắt đỏ và mông lung cho vùng “class fields” là thật sự không cần thiết. Ngoài ra, viết code cho cơ sở dữ liệu giúp kết nối rất nhiều các ứng dụng đến dữ liệu của bạn một cách an toàn, điều này làm cho việc nâng cấp ứng dụng để sử dụng cùng lúc và đồng thời cũng để việc viết code mỗi phần cho hầu hết các ứng dụng sử dụng ngôn ngữ và nền tảng thích hợp được dễ dàng hơn. Giả sử bạn có thể viết back – end(7) cho một ứng dụng chạy trên nền web bằng Java(8), một vài bản kiểm tra bằng ngôn ngữ Ruby(9) và một giao diện đẹp mắt bằng Processing(10).
Cuối cùng, hãy nhớ rằng RDBMS sẽ hoạt động tốt để tối ưu hóa các lệnh SQL của bạn, cho phép bạn tập trung vào chức năng của các ứng dụng hơn là vào điều chỉnh thuật toán. Các hệ thống cơ sở dữ liệu cao cấp sẽ tận dụng lợi thế của bộ xử lý đa lõi, và khi công nghệ được cải thiện, hiệu suất ứng dụng của bạn cũng vậy.
Chú thích:
(1): Relational Database Management System.
(2) (3): Là các hệ thống quản trị cơ sở dữ liệu quan hệ.

Phần 49: Học Ngoại Ngữ

Lập trình viên cũng cần phải giao tiếp nhiều.

Cuộc đời của một lập trình viên dường như đa số là làm việc với máy tính. Cụ thể là với các chương trình chạy trên máy. Cuộc giao tiếp này cho việc thể hiện ý tưởng theo cách có thể đọc bằng máy. Đây là một triển vọng hứng khởi: Các chương trình đều trở nên hiện thực hóa từ những ý tưởng mà không có sự tham gia của chất vật lí nào.

Lập trình viên cần phải lưu loát trong ngôn ngữ của máy móc, dù là thật hay ảo, và các ý tưởng chung liên quan đến ngôn ngữ ấy đều thông qua các công cụ phát triển. Học thêm nhiều khái niệm trừu tượng là điều quan trọng, nếu không thì một vài ý tưởng sẽ trở nên rất khó để thể hiện. Một người lập trình giỏi cần có khả năng ra khỏi thói quen hằng ngày, nhận thức được các ngôn ngữ khác nhau sẽ thể hiện các mục đích khác nhau. Khi bạn làm được, thời cơ sẽ đến.

Ngoài giao tiếp với máy móc thì người lập trình cần phải giao tiếp với đồng nghiệp của họ. Ngày nay các dự án lớn là thành quả của sự nỗ lực tập thể hơn chỉ đơn giản là một ứng dụng nghệ thuật của lập trình. Quan trọng là phải hiểu và diễn đạt nhiều hơn những các khái niệm trừu tượng có thể đọc bằng máy. Những người lập trình viên giỏi nhất tôi từng biết hầu như đều thông thạo tiếng mẹ đẻ của họ, cũng như các ngôn ngữ thông dụng khác. Đây không chỉ là về giao tiếp với người khác: nói được một ngôn ngữ thành thạo sẽ mang lại sự suy nghĩ rõ ràng không thể bỏ qua khi trừu tượng hóa một vấn đề. Và đây cũng là vấn đề của việc lập trình.

Bên cạnh giao tiếp với máy móc, bản thân và đồng nghiệp, một dự án vẫn còn có những người hỗ trợ khác, đa số là có nền tảng kĩ thuật khác nhau hoặc không có. Họ sống với việc thử nghiệm, chất lượng và triển khai, với tiếp thị và bán hàng, họ là những người dùng cuối cùng ở vài công sở (hoặc cửa hàng hoặc nhà ở). Bạn phải hiểu họ và các mối quan tâm của họ. Điều này gần như không thể nếu bạn không thể nói

ngôn ngữ của họ - ngôn ngữ trong thế giới, lĩnh vực của họ. Trong khi bạn nghĩ rằng mình đã có một cuộc trao đổi rất ổn với họ, họ chắc chắn không nghĩ thế.

Nếu bạn nói chuyện với nhân viên kế toán, bạn cần có kiến thức cơ bản về cost-center, về vốn bất động, vốn làm việc và những thứ khác. Nếu như bạn nói chuyện với tiếp thị hoặc luật sư, một vài thuật ngữ và ngôn ngữ (và suy nghĩ của họ) sẽ trở nên quen thuộc với bạn. Những ngôn ngữ chuyên ngành đó cần được thông thạo lưu loát bởi một người trong dự án - lập trình viên là những người lí tưởng. Lập trình viên là những người chịu trách nhiệm cuối cùng cho việc đưa các ý tưởng từ máy tính đến đời sống thực tiễn.

Và, tất nhiên, cuộc sống là những gì nhiều hơn máy tính phần mềm có. Theo ghi chép của Charlemagne, biết được một ngôn ngữ đồng nghĩa với việc hiểu được một tâm hồn. Đối với những mối liên hệ của bạn ngoài ngành công nghiệp phần mềm, việc biết ngoại ngữ sẽ được đánh giá cao. Để biết khi nào nên lắng nghe hơn là nói. Để biết được hầu hết các ngôn ngữ đều không dùng đến lời nói.

Whereof one cannot speak, thereof one must be silent. - Ludwig Wittgenstein
(Cái gì người ta không thể nói về, người ta phải im lặng về nó.)

Phần 50: Học Cách Ước Tính

Là một lập trình viên, bạn phải có khả năng đưa ra các ước tính cho quản lí, đồng nghiệp, và người dùng về các công việc được giao cho bạn, như thế họ sẽ có cách kế hoạch cụ thể, hợp lí về thời gian, giá cả, công nghệ, và các nguồn cung cấp khác cần cho công việc của họ.

Để có năng lực ước tính thì chắc chắn bạn phải học vài kĩ thuật ước lượng. Điều cơ bản đầu tiên là bạn phải học được ước lượng là gì, và chúng được sử dụng với mục đích gì - nó có vẻ khá kì quặc khi nhiều nhà phát triển và người điều hành vẫn không thực sự biết điều này.

Sau đây là một cuộc trao đổi điển hình giữa nhà quản lý dự án và lập trình viên:

Nhà quản lý dự án: Anh có thể cho tôi biết thời gian cụ thể để phát triển tính năng xyz hay không ?
Lập trình viên: Một tháng.
Nhà quản lý dự án: Một tháng là quá lâu! Chúng ta chỉ có một tuần thôi.
Lập trình viên: Tôi cần ít nhất ba tuần.
Nhà quản lý dự án: Tôi có thể cho anh tối đa hai tuần.
Lập trình viên: Quyết định vậy nhé.

Cuối cùng, lập trình viên đưa ra những "ước tính" phù hợp với nhà quản lí. Nhưng bởi vì đó là "ước tính" của người lập trình, nên họ phải chịu trách nhiệm cho chúng. Để hiểu được cuộc đối thoại trên có vấn đề gì, chúng ta phải tìm hiểu ba định nghĩa - estimate. target, và commitment:

- Estimate là sự tính toán hoặc đánh giá gần đúng về giá trị, số lượng, định lượng, hoặc mức độ của một thứ gì đó. Việc này có nghĩa là đưa ra những đánh giá thực tế dựa trên thông tin có được và kinh nghiệm trước đây - bạn nên bỏ qua những hy vọng và mong muốn trong khi tính toán. Đồng thời, việc ước tính này không mang tính chính xác, ví dụ, một nhiệm vụ phát triển không thể được ước tính kéo dài trong 234.14 ngày.
- Targer là mục tiêu kinh doanh mong muốn, ví dụ, "Hệ thống phải hỗ trợ đồng thời ít nhất 400 người dùng."
- Commitment là một lời hứa cung cấp chức năng cụ thể đến một mức độ chất lượng nhất định vào một ngày hoặc sự kiện rõ ràng. Một ví dụ có thể là "Chức năng tìm kiếm sẽ có thể sử dụng sau lần ra mắt sản phẩm tới."

Estimates, targets, và commitments hoàn toàn độc lập với nhau, nhưng targets và commitments phải được dựa trên estimates. Theo ghi chép của Steve McConnell, "Lí do cơ bản của việc ước tính phần mềm không phải là dự đoán kết quả của dự án; nó được dùng để xác định rằng liệu mục tiêu của dự án có đủ tính thiết thực để điều khiển dự án đạt đến kết quả đó hay không." Do đó, mục đích của việc ước tính là làm cho sự

quản lí dự án đạt tiêu chuẩn và việc lên kế hoạch trở nên khả thi, cho phép các bên liên quan đến dự án (stakeholders) thực hiện việc cam kết dựa trên mục tiêu thực tiễn.

Những gì mà nhà quản lí thực sự yêu cầu lập trình viên trong cuộc đối thoại phía trên là đưa ra lời cam kết dựa trên mục tiêu bất thành văn của nhà quản lí, chứ không phải là để được cung cấp sự ước tính. Sau này khi bạn được yêu cầu để đưa ra ước tính, hãy chắc chắn rằng những người có liên quan họ biết mình đang nói về vấn đề gì, và dự án của bạn sẽ có cơ hội thành công cao hơn. Nào, đã đến lúc để học một vài kĩ thuật…

Phần 51: Học cách nói "hello, world"

Paul Lee, biệt danh, hay được biết đến với cái tên Hoppy, nổi tiếng là một chuyên gia về các vấn đề lập trình. Tôi thật sự cần sự giúp đỡ. Chính vì thế tôi đã tìm đến nơi làm việc của Hoppy để nhờ vả, liệu anh ta có thể kiểm tra một số đoạn code của tôi không?

Chắc chắn rồi Hoppy đáp. Mời anh ngồi. Tôi cẩn thận để không làm rơi những lon Coca rỗng được xếp ngay ngắn thành một cái tháp ở sau anh ta.

- Đoạn code nào?
- Nó ở trong file trong một cái function. Tôi nói.
- Nào chúng ta hãy cùng nhau xem đoạn chương trình này nào. Anh ta tạo một bản copy của file K&R và đẩy cái bàn phím về phía tôi.

Cái IDE của anh ở đâu? Hoppy không có một cái IDE nào cả mà thay vào đó là vài cái editor cái mà tôi không thể làm việc cùng. Anh ta lấy lại cái bàn phím. Chỉ sau một vài cú nhấp phím và chúng tôi đã mở được file – nó là một file khá lớn đấy – và tìm kiếm cái function cần thiết – Đây hẳn là một function lớn. Anh ta nhanh chóng cuộn xuống tới khối lệnh điều kiện mà tôi muốn hỏi anh ta.

Cụ thể thì khoảng lệnh này thực sự thực hiện điều gì khi biến X nhận giá trị âm? Tôi hỏi. Tất nhiên là nó sai.

Tôi đã dành cả buổi sáng để tìm cách ép X nhận giá trị số nguyên âm, nhưng rắc rối chính là đây là một chức năng phức tạp trong một file lớn của một dự án lớn, việc phải thực hiện chu trình recompile và rerunning đã khiến tôi vô cùng nản chí. Liệu một chuyên gia như Hoppy có thể giúp tôi tìm được câu trả lời?

Anh ấy thừa nhận rằng anh ta cũng không chắc chắn. Thật bất ngờ anh ta không hề làm việc trên file K&R đó. Thay vào đó anh ta copy đoạn ấy vào một editor khác, tỉnh chỉnh nó, và biến nó thành một function hoàn chỉnh. Chỉ một lúc sau, anh ta đã hoàn thành một hàm điều khiển chạy mãi mãi dùng để nhập dữ liệu từ của màn hình, sau đó truyền dữ liệu cho hàm cần kiểm tra, và cuối cùng là xuất ra màn hình kết quả trả về. Anh ta lưu file vừa tạo với tên “tryit.c”. Tất cả công việc này tôi đều có thể tự làm đấy nhưng có thể là không nhanh như anh ta.Tuy nhiên bước tiếp theo của anh ta đơn giản đến lạ kỳ và có chút khác lạ so với cách làm việc của tôi:

$ cc tryit.c && ./a.out

Nhìn xem! Chương trình của anh ta thật sự vừa được tạo ra chỉ mới đây thôi đã và đang vận hành. Chúng tôi cùng nhau thử nhập một vài giá trị và kiểm chứng sự hoài nghi của tôi (vậy tôi đã đúng về một số thứ) và rồi anh ta nhanh chóng kiểm tra những vùng lệnh lân cận trong file K&R. Tôi cảm ơn Hoppy và rời đi, một lần nữa cẩn thận để không làm đổ cái tháp lon Coca của anh ta.

Về đến phòng làm việc của mình, tôi tắt IDE. Tôi đã trở nên quá quen với làm việc với các dự án lớn cùng với với một sản phẩm lớn và tôi bắt đầu nghĩ rằng đó chính là những gì mà tôi nên thực hiện. Với một mục đích rõ ràng thì máy tính cũng có thể giải quyết những tác vụ nhỏ. Sau đó, tôi khởi động một editor và bắt đầu gõ.

```
#include <stdio.h>
int main()
{
        printf(“Hello, World\n”);
        return 0;
```

```
}
```

Phần 52: Hãy để dự án của bạn tự lên tiếng

Dự án của bạn có thể có một hệ thống kiểm soát thích hợp. Có thể nó được kết nối với một máy chủ tích hợp liên tục xác minh tính chính xác bằng hàng loạt các kiểm tra tự động. Đó là một điều tốt.

Bạn có thể đưa các công cụ phân tích code tĩnh vào máy chủ của mình để thu thập dữ liệu. Các dữ liệu này cung cấp phản hồi về các khía cạnh cụ thể trong code của bạn, cũng như sự phát triển của chúng theo thời gian. Khi bạn cài đặt dữ liệu, sẽ luôn có một ranh giới mà bạn không muốn vượt qua. Ví dụ khi bạn bắt đầu với phạm vi 20%, bạn không bao giờ muốn tụt xuống mức 15%. Tích hợp liên tục giúp bạn theo dõi những con số này, nhưng bạn vẫn cần kiểm tra thường xuyên. Hãy tưởng tượng bạn ủy thác nhiệm vụ này cho chính dự án và khi mọi thứ trở nên tồi tệ hơn, bạn cũng dựa vào nó để báo cáo.

Dự án của bạn cần có tiếng nói riêng. Điều này có thể thực hiện qua email hay tin nhắn thông báo cho các nhà phát triển về sự suy giảm hoặc cải thiện mới nhất của số liệu. Nhưng nó thậm chí còn hiệu quả hơn khi thể hiện dự án của bạn bằng cách sử dụng một thiết bị phản hồi cực độ (XFD).

Ý tưởng của XFD là vận hành một thiết bị như đèn, vòi nước di động, robot đồ chơi hay thậm chí là máy phóng tên lửa USB, dựa trên kết quả phân tích tự động. Bất cứ khi nào giới hạn của bạn bị phá vỡ, thiết bị sẽ thay đổi trạng thái của nó. Trong trường hợp của bóng đèn, nó sẽ sáng lên rõ ràng. Bạn sẽ không bỏ lỡ thông điệp ngay cả khi bạn đang vội vã ra về.

Tùy thuộc vào loại thiết bị phản hồi mà bạn có thể nghe thấy tiếng ngắt, nhìn thấy các tín hiệu cảnh báo đỏ hay thậm chí “ngửi” thấy “mùi code”. Các thiết bị có thể được nhân rộng tại các địa điểm khác nhau nếu bạn làm việc trong một nhóm phân phối. Bạn có thể đặt đèn giao thông trong văn phòng của bạn, biểu thị trạng thái tổng thể của dự án. Quản lý của bạn sẽ đánh giá cao điều đó.

Hãy để óc sáng tạo dẫn dắt trong việc lựa chọn một thiết bị phù hợp. Nếu phong cách của bạn có chút lập dị, bạn có thể trang bị cho linh vật của đội mình những món đồ điều khiển bằng radio. Nếu bạn muốn một phong cách chuyên nghiệp hơn, hãy đầu tư vào những chiếc đèn thiết kế bóng bẩy. Tìm kiếm trên Internet để có thêm cảm hứng. Bất cứ thứ gì có phích cắm hay điều khiển từ xa đều có tiềm năng sử dụng như một thiết bị phản hồi cực độ.

Thiết bị phản hồi cực độ hoạt động như hộp thoại trong dự án của bạn. Dự án ấy hiện đang được các nhà phát triển phàn nàn hoặc khen ngợi theo các tiêu chuẩn mà họ đã đề ra. Bạn có thể thúc đẩy việc nhân cách hóa này bằng cách áp dụng phần mềm tổng hợp giọng nói và một cặp loa. Bây giờ thì dự án của bạn đã thực sự lên tiếng.

Phần 53: Linker(Trình liên kết) không phải là một chương trình ma thuật gì cả đâu

Thường xuyên chán nản (nó lại xảy ra với tôi ngay trước khi tôi viết bài này), quan điểm của nhiều lập trình viên về quá trình chuyển từ mã nguồn sang thực thi được liên kết tĩnh trong một ngôn ngữ được biên dịch là:

1. Chỉnh sửa mã nguồn
- Biên dịch mã nguồn thành các tệp đối tượng
- Một cái gì đó kỳ diệu xảy ra
- Chạy thực thi

Bước 3 đương nhiên là bước liên kết rồi. Tại sao tôi lại nói một điều thái quá lên như vậy? Tôi đã làm hỗ trợ công nghệ trong nhiều thập kỷ và tôi nhận được các câu hỏi sau đây:

1. Trình liên kết nói def được định nghĩa nhiều lần..
- Trình liên kết nói abc là một biểu tượng chưa được giải quyết.
- Tại sao khả năng thực thi của tôi rất lớn?

Theo cuốn sách "What do I do now?" (tạm dịch là "Tôi phải làm gì bây giờ?), thông thường với các cụm từ "seems to" (dường như) và "somehow" (bằng cách nào đó) trộn lẫn vào, và sẽ có một sự khó khăn được sinh ra.Đó là "dường như" và "bằng cách nào đó" chỉ ra rằng quá trình liên kết được xem như là một quá trình kỳ diệu, và có lẽ chỉ có các pháp sư hay phù thủy mới có thể hiểu được. Quá trình biên dịch không gợi ra các loại cụm từ này, có thể ngầm đình rằng các lập trình viên thường hiểu cách trình biên dịch làm việc hoặc ít nhất là những gì họ làm.

Một linker là một chương trình rất ngu ngốc, dành cho người đi bộ, đơn giản. Tất cả những gì nó làm là nối các phần mã và dữ liệu của các tệp đối tượng lại với nhau, kết nối các tham chiếu đến các biểu tượng với định nghĩa của chúng, kéo các biểu tượng chưa được giải quyết ra khỏi thư viện và viết ra một tệp thực thi. Là nó đó.. Không có sự mê hoặc! Không có phép thuật. Sự nhàm chán trong việc viết một trình liên kết thường là tất cả về giải mã và tạo ra các định dạng tệp thường quá phức tạp, nhưng điều đó không thay đổi bản chất cơ bản của một trình liên kết.

Vì vậy, hãy nói rằng trình liên kết đang nói def được định nghĩa nhiều lần. Nhiều ngôn ngữ lập trình, như C, C ++ và D, có cả khai báo và định nghĩa. Các khai báo thường đi vào các tệp tiêu đề, như:

extern int iii;

tạo ra một tham chiếu bên ngoài đến ký hiệu iii. Mặt khác, một định nghĩa, thực sự đặt lưu trữ cho biểu tượng, thường xuất hiện trong tệp thực hiện và trông như thế này:

int iii = 3;

Có bao nhiêu định nghĩa có thể có cho mỗi biểu tượng? Như trong phim Highlander, chỉ có duy nhất một. Vậy, điều gì sẽ xảy ra nếu một định nghĩa về iii xuất hiện trong nhiều tệp thực hiện?

```
// File a.c
int iii = 3;

// File b.c
double iii(int x) { return 3.7; }
```

Trình liên kết sẽ phàn nàn về iii được nhân lên.

Không chỉ có thể có một, mà phải có một. Nếu iii chỉ xuất hiện dưới dạng tuyên bố, nhưng không bao giờ là định nghĩa, trình liên kết sẽ phàn nàn về iii là một biểu tượng chưa được giải quyết.

Để xác định lý do tại sao một thực thi là kích thước của nó, hãy xem tập tin bản đồ mà các trình liên kết tùy ý tạo ra.Một tệp bản đồ không có gì khác hơn là một danh sách tất cả các biểu tượng trong tệp thực thi cùng với địa chỉ của chúng.Điều này cho bạn biết các mô-đun đã được liên kết từ thư viện và kích thước của mỗi mô-đun. Bây giờ bạn có thể thấy nơi phình ra từ đâu.Thường thì sẽ có các mô-đun thư viện mà bạn không biết tại sao lại được liên kết bên trong. Để tìm ra nó, tạm thời xóa mô-đun đáng ngờ khỏi thư viện và xem lại. Lỗi biểu tượng không xác định sau đó được tạo sẽ cho biết ai đang tham chiếu mô-đun đó.

Mặc dù không phải lúc nào cũng rõ ràng ngay lập tức tại sao bạn nhận được một thông điệp liên kết cụ thể, nhưng không có gì kỳ diệu về các trình liên kết. Các cơ chế là đơn giản; đó là chi tiết bạn phải tìm ra trong từng trường hợp.

Phần 54: “Tuổi thọ” của các giải pháp tạm thời.

Tại sao chúng ta tạo ra các giải pháp tạm thời?

Thông thường nếu có một vấn đề, chúng ta sẽ giải quyết nó ngay. Điều này có thể đến từ nội bộ của nhóm phát triển, một số công cụ được thêm vào trong một chuỗi công cụ. Nó có thể là từ bên ngoài, hiển thị cho người dùng cuối, chẳng hạn như thêm một chức năng bị thiếu.

Trong hầu hết các nhóm và hệ thống, bạn sẽ tìm thấy một số phần mềm được tách riêng ra, nó được coi là chương trình nhất thời để thay thế khi không tuân theo các tiêu chuẩn và phần còn lại của code. Chắc chắn bạn sẽ nghe các nhà phát triển phàn nàn về điều này. Lý do cho sự “sáng tạo” của họ rất nhiều và đa dạng, nhưng điều mấu chốt để sử dụng các giải pháp tạm thời: Vì nó hữu ích.

Tuy nhiên, các giải pháp tạm thời có được “quán tính” (hoặc động lượng, tùy thuộc vào quan điểm của bạn). Bởi vì nó vẫn hoạt động âm thầm, cuối cùng mới phát huy lợi ích và được chấp nhận rộng rãi, không cần phải làm gì ngay lập tức. Bất cứ khi nào một bên liên quan phải quyết định xem hành động nào hiệu quả nhất, họ sẽ phải xếp trên việc tích hợp một giải pháp tạm thời. Lý do vì sao? Bởi vì nó hoạt động âm thầm cho đến khi được chấp nhận. Nhược điểm duy nhất là nó không tuân theo các tiêu chuẩn và các hướng dẫn đã chọn – ngoại trừ một vài thị trường ngách (niche markets), đây không được coi là một lực lượng đáng kể.

Vì vậy, giải pháp tạm thời vẫn còn ở đó. Mãi mãi là như vậy.

Và nền có lỗi phát sinh với giải pháp tạm thời đó, sẽ khó có một bản cập nhật phù hợp với chất lượng sản xuất. Phải làm sao để một bản cập nhật tạm thời nhanh chóng phát

huy tác dụng và được chấp nhận. Nó thể hiện những điểm mạng tương tự như giải pháp đầu tiên nhưng nó chỉ cập nhật mới hơn.

Đây có phải là vấn đề?

Câu trả lời phụ thuộc vào dự án của bạn và số vốn cá nhân trong các tiêu chuẩn sản xuất code. Khi hệ thống chứa quá nhiều giải pháp tạm thời, độ phức tạp và tính lộn xộn sẽ tăng lên làm cho khả năng bảo trì sẽ giảm đi. Tuy nhiên, có lẽ việc đặt câu hỏi là sai. Hãy nhớ rằng chúng ta đang nói về một giải pháp. Nó có thể không phải là giải pháp ưu tiên của bạn nhưng động lực để thực hiện lại giải pháp này là rất thấp.

Vậy chúng ta có thể làm gì nếu thấy có lỗi?

- Tránh tạo ra một giải pháp tạm thời ngay từ đầu.
- Thay đổi các yếu tố ảnh hưởng đến quyết định của người quản lý dự án.
- Hãy để yên nó như vậy.
- Hãy xem xét các tùy chọn này kỹ càng hơn:

Tránh việc chức năng không hoạt động ở hầu hết các trường hợp. Có một vấn đề thực sự cần giải quyết và các tiêu chuẩn đề ra lại quá hạn chế. Bạn có thể dành một chút sức lực của mình để cố gắng thay đổi các tiêu chuẩn đó. Một nỗ lực đáng trân trọng dù là tẻ nhạt và hiệu quả … không đến ngay lập tức để giải quyết vấn đề của bạn.

Sự ảnh hưởng bắt nguồn từ văn hóa dự án, trong đó có sự ngăn cản việc thay đổi. Nó có thể sẽ thành công trong một dự án nhỏ chỉ có bạn và bạn chỉ tình cờ “dọn dẹp” mớ hỗn độn mà không cần thông qua ai. Nó cũng có thể thành công trong một dự án có nhiều vấn đề đến mức đình trệ và buộc phải dành nhiều thời gian tìm ra cách giải quyết.

Tính năng động sẽ được áp dụng nếu tùy chọn trước đó không có.

Bạn sẽ tạo ra nhiều giải pháp, một trong số chúng sẽ chỉ là tạm thời nhưng hầu hết lại hữu ích. Cách tốt nhất để bảo qua các giải pháp tạm thời là làm cho chúng trở nên thừa thãi, qua đó cung cấp một giải pháp hiệu quả hơn. Tôi mong rằng bạn có được sự

thoải mái để chấp nhận những điều bạn không thể thay đổi, can đảm thay đổi những điều bạn có thể và có khả năng để biết được sự khác biệt.

Phần 55: Làm cho giao diện dễ sử dụng hơn

Một trong hầu hết các nhiệm vụ phổ biến trong việc phát triển phần mềm đó là kỹ thuật làm giao diện. Các giao diện đạt mức độ cao nhất của sự trừu tượng (những người sử dụng giao diện), ở cấp độ thấp nhất (chức năng các giao diện) và ở tầm trung (các lớp giao diện, thư viện các giao diện, vân vân,...). Bất kể việc bạn có làm việc với những người dùng cuối cùng hay không để chỉ rõ cách thức họ tương tác bằng một một hệ thống, hay kết hợp với các nhà phát triển để chỉ ra một API(1), hoặc là trình bày các chức năng riêng biệt cho một class, thiết kế giao diện vẫn là một phần cực kì quan trọng trong công việc của bạn. Nếu hoàn thành tốt, các giao diện của bạn sẽ được đưa vào sử dụng và đồng thời thúc đẩy năng suất các công việc. Nếu làm không tốt, có thể giao diện sẽ gây thất vọng hoặc tệ hơn là nguồn gốc phát sinh các lỗi.

Một giao diện tốt sẽ có các đặc điểm sau:

- Dễ dàng sử dụng chính xác: Những người đang sử dụng một giao diện được thiết kế tốt hầu hết luôn sử dụng chúng một cách chính xác, bởi vì giao diện đó thật sự rất dễ hiểu và sử dụng. Trong một giao diện GUI(2), người ta thường click vào các icon, nút bấm hoặc cổng vào menu, tại vì nó rõ ràng và dễ thực hiện. Trong giao diện API, người ta hầu hết thường duyệt các thông số chính xác bằng các giá trị chính xác, vì đó là điều hiển nhiên nhất. Với các giao diện dễ dàng để sử dụng một cách chính xác, công việc sẽ hoạt động trơn tru.
- Khó để sử dụng sai: Một giao diện tốt sẽ đoán trước được những lỗi sai mà con người có thể mắc phải và làm chúng trở nên khó xảy ra hơn vào lần tiếp theo. Một giao diện GUI cũng có thể vô hiệu hóa hoặc hủy các lệnh không có ý nghĩa trong bối cảnh hiện tại. Ví dụ, một giao diện API có thể loại bỏ các sự cố xung đột thứ tự bằng cách cho phép các thông số được thông qua theo bất kì thứ tự nào

Một cách hiệu quả để thiết kế các giao diện sao cho dễ sử dụng đó là dùng thử trước khi đem vào thực tiễn. Hãy giả lập một giao diện GUI trên Whiteboard(3), sử dụng Index card(4) và làm việc với nó trước khi bắt tay vào viết bất kì dòng code nào. Hoặc là viết các hàm call cho một API trước khi các function được khai báo. Hãy bỏ qua các trường hợp sử dụng phổ biến và chỉ ra: bạn muốn giao diện hoạt động như thế nào, muốn click vào những gì và muốn được thông qua những gì? Một giao diện dễ sử dụng sẽ trông rất tự nhiên, bởi vì những nhà thiết kế sẽ để người dùng làm những gì họ muốn. Bạn hoàn toàn có thể làm được các giao diện tương tự như vậy, nếu bạn phát triển chúng theo quan điểm của người sử dụng. (Đó cũng là một trong những điều rất quan trọng thuộc giai đoạn kiểm thử chương trình).

Để làm cho các giao diện trở nên khó sử dụng sai đòi hỏi hai việc. Thứ nhất, bạn phải đoán trước được các lỗi mà người dùng thường mắc phải và tìm ra cách để ngăn chặn chúng. Thứ hai, bạn phải chú ý xem nguyên nhân một giao diện bị sử dụng sai trong quà trình giải phóng sớm và sửa đổi giao diện đó. Đúng vậy! Sửa đổi giao diện đó để ngăn chặn các lỗi có thể xảy ra. Cách tốt nhất để không sử dụng sai đó là ngăn chặn việc sử dụng như vậy. Nếu người dùng vẫn muốn xóa bỏ các hoạt động không thể hủy bỏ, hãy thử làm theo ý họ. Nếu như họ vẫn tiếp tục thông qua sai giá trị cho API, hãy cố gắng sửa đổi API để nhận lấy các giá trị mà người dùng muốn chuyển đổi.

Trên tất cả, hãy nhớ rằng giao diện được tạo ra để phục vụ cho nhu cầu tiện nghi của các người dùng, không phải cho người thực hiện.

*Chú thích:
(1): API là viết tắt của Application Programming Interface (giao diện lập trình ứng dụng) phương thức kết nối với các thư viện và ứng dụng khác.
(2): GUI là viết tắt của Graphical User Interface (giao diện đồ họa người dùng), nơi mà bạn tương tác với máy tính bằng hình ảnh chứ không phải là văn bản.
(3): Whiteboard: là một khung vẽ kỹ thuật số miễn phí, là một ứng dụng cho phép người dùng phác thảo, lập kế hoạch và cộng tác với nội dung và ý tưởng của họ.
(4): Index card: (Thẻ chỉ mục) Bao gồm kho thẻ được cắt theo kích thước tiêu chuẩn, được sử dụng để ghi và lưu trữ một lượng nhỏ dữ liệu rời rạc.

Phần 56: Khiến những điều vô hình trở nên rõ ràng!

Những khía cạnh của vấn đề “vô hình” được đề cập nhiều là việc duy trì các nguyên tắc của phần mềm. Thuật ngữ của chúng tôi rất giàu tính ẩn dụ vô hình – tính minh bạch của cơ chế và ẩn đi thông tin, tôi chỉ liệt kê ra hai trong số các thuật ngữ này. Phần mềm và quá trình phát triển nó được giải thích bởi Douglas Adams, hầu hết là không được để ý tới.

Source code không tự nhiên sinh ra, không có bản năng và cũng không tuân theo một định luật vật lý nào. Điều này sẽ được cụ thể khi bạn tải code vào một trình biên dịch, nhưng khi bạn đóng chương trình lại và tất cả biến mất. Việc suy nghĩ về điều này trong một thời gian dài, như một cành cây rụng xuống và không ai để ý đến, bạn bắt đầu tự hỏi liệu nó có tồn tại hay không.
Một ứng dụng đang chạy với giao diện và những tính năng bình thường nhưng lại không tiết lộ thông tin gì về source code của nó. Trang chủ của Google nhìn trông rất đơn giản nhưng những thứ đằng sau đó chắc chắn rất phức tạp.

Nếu bạn đã hoàn thành 90% công việc và bị mắc kẹt mãi mãi khi cố gắng gỡ 10% còn lại đang bị lỗi thì suy cho cùng, bạn cũng không hoàn thành 90% đó phải không? Gỡ lỗi không phải là tiến bộ. Bạn không được trả tiền cho quá trình lãng phí như debug. Tốt hơn là bạn nên tìm hiểu rõ ràng các lỗi để bạn có thể hiểu nó là gì và bắt đầu nghĩ về việc cố gắng không tạo ra nó ngay từ đầu.

Nếu dự án của bạn đang đi đúng hướng rồi một tuần sau đó, bạn gặp lỗi và trễ mất sáu tháng. Vấn đề đáng lo nhất không phải là trễ sáu tháng, mà là những điều vô hình quá lớn đã che đi việc chậm trễ sáu tháng. Thiếu sự rõ ràng đồng nghĩa với việc thiếu đi sự tiến bộ.

Những vấn đề vô hình có thể rất nguy hiểm. Bạn có thể hình dung cụ thể hơn khi có một cái gì đó kết nối các suy nghĩ của bạn. Bạn có thể quản lý mọi thứ tốt hơn khi bạn có thể nhìn thấy chúng và thấy chúng thay đổi liên tục:

Viết ra các đơn vị kiểm thử (unit tests) sẽ có thêm cơ sở để dễ dàng hơn trong kiểm tra đơn vị code. Điều này giúp biết được liệu có hay không các yếu tố chất lượng trong quá trình phát triển mà bạn muốn code của mình có, các yếu tố chất lượng này như các khớp nối giúp gắn kết code của bạn.

Chạy các đơn vị kiểm thử cung cấp thông tin về những tính chất của code. Nó giúp tiết lộ sự có mặt hay không của thời gian chạy các yếu tố trong ứng dụng mà bạn mong muốn, những yếu tố như là sự chặt chẽ và chính xác.

Sử dụng bảng thông báo và thẻ là cho quá trình dễ dàng nhìn thấy và cụ thể hơn. Các tác vụ có thể được đánh giá là Chưa bắt đầu, Đang trong tiến trình hoặc Đã hoàn thành mà không cần tham khảo công cụ quản lý dự án ẩn và không cần thông qua lập trình viên để báo cáo tình trạng.

Thực hiện đều các quá trình phát triển làm tăng sự rõ ràng trong tiến trình bằng cách làm tăng tần suất của cơ sở phát triển. Hoàn thành phần mềm đáng tin cậy cho thấy thực tế khác đi so với tưởng tượng.

Tốt nhất là phát triển phần mềm với cơ sở rõ ràng một cách thường xuyên. Tầm nhìn mang lại sự tự tin rằng sự tiến bộ là có thật và không phải là tưởng tượng, có sự chú ý và không bất cẩn, lặp lại và không vô tình.

Phần 57: Mẹo giúp cải thiện hiệu quả của các hệ thống xử lý song song

Các lập trình viên ngay từ đầu đã được dạy rằng việc lập trình đồng thời là một vấn đề rất khó, đặc biệt đối với việc xử lý song song, rằng chỉ có những lập trình viên xuất sắc

nhất mới có thể vận dụng nó một cách đúng đắn, và ngay cả những người xuất sắc đó cũng đã mắc sai lầm. Có vô số các lý thuyết về luồng, các cột mốc, các tiến trình, và rất khó để có thể truy cập vào các for programming biến trong luồng một cách an toàn.

Thật vậy, có rất nhiều vấn đề nan giải. Nhưng gốc rễ của chúng là gì? Chia sẻ bộ nhớ. Hầu hết các vấn đề của lập trình đồng thời mà mọi người gặp phải đều liên quan việc chia sẻ một bộ nhớ có thể bị biến đổi (do quá trình xử lý luồng): race conditions, deadlock, livelock, v.v…. Câu trả lời dường như rất rõ ràng: hoặc là từ bỏ đồng thời hoặc tránh chia sẻ bộ nhớ.

Gửi đồng thời gần như chắc chắn không phải là một lựa chọn. Máy tính càng ngày càng có nhiều lõi hơn, do đó khai thác song song ngày càng trở nên quan trọng. Chúng ta không thể tăng tốc vi xử lý để cải thiện hiệu suất ứng dụng. Chỉ bằng cách khai thác song song thì hiệu suất của các ứng dụng sẽ được cải thiện. Rõ ràng, việc không cải thiện hiệu suất là một lựa chọn, nhưng khó có thể được người dùng chấp nhận.

Vậy, chúng ta có thể tránh chia sẻ bộ nhớ như thế nào? Chắc chắn rồi.

Thay vì sử dụng các luồng và chia sẻ bộ nhớ như mô hình lập trình của chúng ta, ta có thể sử dụng các tiến trình và thông báo truyền qua. Tiến trình này có nghĩa là một trạng thái độc lập được bảo vệ với mã thực thi, không nhất thiết phải là một tiến trình hệ điều hành. Các ngôn ngữ như Erlang(và trước đó là Occam) đã chỉ ra rằng các tiến trình là một cơ chế rất thành công để lập trình các hệ thống đồng thời và song song. Các hệ thống như vậy không có tất cả các ứng suất đồng bộ hóa mà bộ nhớ chia sẻ, các hệ thống đa luồng có. Hơn nữa, có một mô hình chính thức – Mô hình giao tiếp tuần tự (Communicating Sequential Processes - CSP) có thể được áp dụng như một phần kỹ thuật của các hệ thống đó.

Chúng ta có thể tiến xa hơn và giới thiệu các hệ thống dataflow như một cách tính toán. Trong một hệ thống dataflow, không có luồng điều khiển được lập trình rõ ràng. Thay vào đó, một biểu đồ hướng của các toán tử, được kết nối bằng đường dẫn dữ liệu, được thiết lập và sau đó dữ liệu được đưa vào hệ thống. Đánh giá được kiểm soát bởi sự sẵn sàng của các dữ liệu trong hệ thống. Chắc chắn không có vấn đề đồng bộ.

Điều đó nói rằng, các ngôn ngữ như C, C ++, Java, Python và Groovy là ngôn ngữ chính của phát triển hệ thống và tất cả các ngôn ngữ này được trình bày cho lập trình viên là ngôn ngữ để phát triển bộ nhớ chia sẻ, hệ thống đa luồng. Vậy thì cái gì có thể làm được? Câu trả lời là sử dụng, hoặc nếu chúng không tồn tại, hãy tạo ra các thư viện và các framework cung cấp các mô hình tiến trình và thông điệp, tránh tất cả việc sử dụng bộ nhớ có thể thay đổi được chia sẻ.

Nói chung, không phải lập trình với bộ nhớ chia sẻ, mà thay vào đó là sử dụng thông điệp truyền đi, có thể là cách thành công nhất để thực hiện các hệ thống điều khiển song song hiện hữu trong phần cứng máy tính. Có lẽ kỳ lạ, mặc dù các tiến trình có trước các luồng như là một đơn vị đồng thời, trong tương lai dường như sẽ sử dụng các luồng để triển khai thực hiện các tiến trình.

Phần 58: Thông điệp cho tương lai

Có lẽ bởi hầu hết là những người thông minh, trong tất cả những năm tôi dạy và làm việc cùng với các lập trình viên, có vẻ như phần đông nghĩ rằng vì những vấn đề họ gặp phải khá khó khăn nên các giải pháp cũng cần phải phức tạp theo với mọi đối tượng (thậm chí có thể là chính họ một vài tháng sau khi hoàn thành code).

Tôi nhớ một câu chuyện với Joe, một sinh viên lớp cấu trúc dữ liệu của tôi, đã đến để cho tôi xem những gì anh ấy đã viết. Anh ấy nói rằng "Betcha không thể đoán nó làm gì!"

"Bạn nói đúng", tôi đồng ý ngay mà không cần dành quá nhiều thời gian đọc ví dụ của anh ấy và tự hỏi làm thế nào để giải thích dễ hiểu một vấn đề quan trọng. "Tôi chắc chắn bạn đã làm việc chăm chỉ. Tuy nhiên, tôi tự hỏi, liệu bạn có quên mất một điều quan trọng. Nói đi, Joe, bạn có cậu em trai nào không?"

"Có chứ. Chắc chắn rồi! Phil! Em ấy đang theo học lớp khởi đầu của bạn. Nó cũng đang học lập trình!" Joe tự hào tuyên bố.

"Thật tuyệt," tôi trả lời. "Tôi tự hỏi liệu cậu ấy có thể đọc hiểu đoạn code này."

"Không đời nào!" Joe nói. "Đây là một đoạn code khó!"

"Giả sử," tôi nói, "đoạn code này thực sự đi vào hoạt động vài năm sau và Phil được thuê để thực hiện cập nhật. Và xem bạn đã làm gì này?" Joe nhìn tôi chớp mắt. "Chúng ta đều biết rằng Phil thực sự thông minh, phải không?" Joe gật đầu. "Và dù ghét phải nói điều này, nhưng tôi cũng khá thông minh!" Joe cười toe toét. "Vì vậy, nếu tôi không thể hiểu những gì bạn đã làm và cậu em thông minh của bạn có vẻ cũng sẽ chật vật với nó, thì những gì bạn đã viết còn nghĩa lý gì nữa đây?" Joe có vẻ đã có cái nhìn khác với đoạn code của anh ấy. "Thế này thì sao," tôi gợi ý, "Hãy coi mỗi dòng code bạn viết như thể một tin nhắn gửi đến ai đó trong tương lai- có thể là em trai của bạn. Hãy giả vờ bạn đang giải thích cho người ấy cách để giải quyết vấn đề khó nhằn này."

"Đây có phải là những gì bạn muốn không? Rằng trong tương lai, một lập trình viên nào đó sẽ thấy đoạn code này của bạn và thốt lên, "Wow! Thật tuyệt vời! Tôi hoàn toàn hiểu được những gì được thực hiện ở đây và tôi ngạc nhiên về sự thanh lịch- không-sự đẹp đẽ của đoạn code này. Tôi sẽ cho những người khác trong nhóm xem. Quả là một kiệt tác!"

"Joe, bạn nghĩ liệu bạn có thể viết code giải quyết vấn đề khó khăn này đẹp đẽ như một lời ca không? Chính xác nó sẽ giống như một giai điệu bắt tai. Tôi nghĩ rằng bất cứ ai có khả năng đưa ra giải pháp phức tạp như cái bạn có ở đây cũng có thể viết một cái gì đó đẹp đẽ. Tôi tự hỏi liệu tôi có nên bắt đầu chấm điểm cả cái đẹp không? Bạn nghĩ sao, Joe?"

Joe nhận lại đoạn code của anh ấy và nhìn tôi với một nụ cười thấp thoáng trên môi. "Tôi hiểu rồi, thưa giáo sư, tôi sẽ khiến thế giới tốt đẹp hơn cho Phil. Cảm ơn."

Phần 59: Thiếu cơ hội cho Polymorphism

Polymorphism là một trong những ý tưởng lớn làm nền tảng cho OO. Từ này được lấy từ tiếng Hy Lạp, có nghĩa là nhiều (poly) dạng (morph). Trong lập trình, Polymorphism đề cập đến nhiều dạng của một class đối tượng hoặc phương thức cụ thể. Nhưng Polymorphism không chỉ đơn thuần là về việc triển khai thay thế. Nếu được sử dụng cẩn thận, Polymorphism sẽ tạo ra các context thực thi cục bộ nhỏ cho phép chúng ta làm việc mà không cần phải sử dụng câu lệnh if-then-else dài dòng. Ở bên trong context cho phép chúng ta làm việc trực tiếp, còn ở bên ngoài context, chúng ta buộc phải xây dựng lại nó rồi mới có thể làm việc. Với việc sử dụng cẩn thận các triển khai thay thế, chúng ta có thể nắm bắt context, qua đó giúp sản xuất ít mã hơn nhưng đem lại hiệu quả cao hơn. Điều này được thể hiện tốt nhất thông qua một số mã, chẳng hạn như giỏ hàng giả định đơn giản sau đây:

```
public class ShoppingCart {
    private ArrayList<Item> cart = new ArrayList<Item>();
    public void add(Item item) { cart.add(item); }
    public Item takeNext() { return cart.remove(0);  }
    public boolean isEmpty() { return cart.isEmpty(); }
}
```

Giả sử webshop của chúng tôi cung cấp các mặt hàng có thể tải xuống và các mặt hàng cần vận chuyển. Hãy xây dựng một đối tượng hỗ trợ hoạt động này:

```
public class Shipping {
    public boolean ship(Item item, SurfaceAddress address) { ... }
    public boolean ship(Item item, EMailAddress address { ... }
}
```

Khi một khách hàng đã thanh toán xong, chúng tôi cần vận chuyển hàng hóa:

```
while (!cart.isEmpty()) {
    shipping.ship(cart.takeNext(), ???);
}
```

Tham số ??? không phải là một số toán tử elvis mới. Nó hỏi tôi nên gửi email hay gửi thư cho mục này? Context cần thiết để trả lời câu hỏi này không còn tồn tại. Chúng tôi có thể nắm bắt được phương thức giao hàng trong boolean hoặc enum và sau đó sử dụng if-then-else để điền vào tham số bị thiếu. Một giải pháp khác sẽ là tạo hai class đều mở rộng Item. Hãy gọi đó là DownloadableItem và SurfaceItem. Bây giờ hãy viết một số code. Tôi sẽ quảng cáo Item như một giao diện hỗ trợ phương thức duy nhất-xuất xưởng. Để gửi giỏ hàng, chúng tôi sẽ gọi item.ship(shipper). Các class DownloadableItem và SurfaceItem sẽ cùng thực hiện việc giao hàng.

```
public class DownloadableItem implements Item {
    public boolean ship(Shipping shipper) {
        shipper.ship(this, customer.getEmailAddress());
    }
}

public class SurfaceItem implements Item {
    public boolean ship(Shipping shipper) {
        shipper.ship(this, customer.getSurfaceAddress());
    }
}
```

Trong ví dụ này, chúng tôi đã ủy thác trách nhiệm làm việc với Shipping cho từng Item. Vì mỗi mặt hàng đều được vận chuyển tốt, nên sự sắp xếp này cho phép chúng tôi tiếp tục mà không cần if-then-else. Code này cũng minh hoạ cho hai mẫu thường được kết hợp với nhau: Command và Double Dispatch. Việc sử dụng hiệu quả các mẫu này phụ thuộc vào việc sử dụng Polymorphism một cách cẩn thận. Khi điều đó xảy ra, số lượng câu lệnh if-then-else trong code sẽ giảm thiểu đáng kể.

Mặc dù có những trường hợp sẽ hiệu quả hơn nhiều khi sử dụng if-then-else thay vì Polymorphism, nhưng thông thường, Polymorphism coding sẽ mang lại một cơ sở code nhỏ hơn, dễ đọc hơn và ít mong manh hơn. Số cơ hội bị bỏ lỡ là số lượng các câu lệnh if-then-else trong code.

Phần 60: Tester là bạn của bạn

Mặc dù họ tự gọi mình là “Người đảm bảo chất lượng” hay “Kẻ kiểm soát chất lượng”, nhiều lập trình viên coi họ là “Đồ rắc rối”. Dựa trên quan sát của bản thân, các lập trình viên thường có mối quan hệ không mấy tốt đẹp với những người phụ trách khâu kiểm tra phần mềm của họ. “Họ quá kén chọn” hay “Họ muốn mọi thứ phải hoàn hảo” là những lời phàn nàn phổ biến. Nghe quen đúng không?

Tôi không chắc tại sao, nhưng tôi luôn có một cái nhìn khác về các tester. Có lẽ là vì tester trong công việc đầu tiên của tôi chính là thư ký của công ty. Margaret là một người phụ nữ tốt, giúp duy trì hoạt động của văn phòng, và cố gắng dạy cho một vài lập trình viên trẻ cách cư xử chuyên nghiệp trước mặt khách hàng. Cô cũng chuẩn bị một phần quà nhỏ cho bất kỳ ai tìm ra bug, bất kể như thế nào.

Hồi đó tôi đang làm với một chương trình được viết bởi một kế toán viên nghĩ rằng mình có khả năng lập trình. Và đương nhiên, nó đã có một số vấn đề nghiêm trọng. Mỗi khi tôi nảy ra một ý tưởng mới và Margaret thử áp dụng, nó thường thất bại chỉ sau một vài cú nhấp phím. Đó là khoảng thời gian bực bội và xấu hổ, nhưng cô ấy là một người dễ tính đến nỗi tôi không bao giờ nghĩ sẽ đổ lỗi cho cô ấy. Cuối cùng, Margaret đã có thể khởi động chương trình một cách trơn tru, nhập hóa đơn, in ra và tắt nó đi. Tuyệt vời hơn nữa là khi chúng tôi cài đặt nó vào máy của khách hàng, mọi thứ đều hoạt động tốt. Họ không gặp bất kỳ vấn đề nào vì Margaret đã giúp tôi tìm kiếm và sửa chữa.

Đó là lý do tại sao tôi nói tester là bạn của bạn. Bạn có thể cho rằng các tester khiến bạn trông thật ngốc nghếch bằng cách báo cáo các vấn đề tầm thường. Nhưng khi

khách hàng cảm thấy hài lòng vì không bị làm phiền bởi những “điều nhỏ nhặt” mà QC đã bắt bạn sửa lại trước đó, thì bạn sẽ trở nên tuyệt vời. Hiểu ý tôi chứ?

Hãy thử tưởng tượng: Bạn đang dùng thử một tiện ích sử dụng “thuật toán đột phá trí tuệ nhân tạo” để tìm và khắc phục các sự cố. Bạn kích hoạt nó và ngay lập tức nhận thấy họ viết sai từ “trí thông minh”. Một chút không may, nhưng đó chỉ là một lỗi đánh máy, phải không? Sau đó, bạn nhận thấy cấu hình sử dụng các checkboxes mà đáng ra phải có radio button và một số phím tắt không hoạt động. Hiện tại thì không cái nào trong số này thực sự là vấn đề, nhưng khi kết hợp lại, bạn bắt đầu nghi ngờ các lập trình viên. Nếu họ không thể làm đúng từ những điều đơn giản nhất thì có bao nhiêu khả năng AI của họ có thể thực sự tìm thấy và khắc phục điều gì đó khó khăn như các vấn đề tương tranh?

Họ có thể là những thiên tài, tập trung hết sức vào việc khiến AI trở nên tuyệt vời đến mức họ không nhận thấy những điều tầm thường đó. Và nếu không có tester kén- chọn ở đó để chỉ ra các vấn đề, họ sẽ không thể tự tìm ra chúng. Và bây giờ bạn sẽ hoài nghi về năng lực của các lập trình viên.

Dù nghe có vẻ lạ lùng, nhưng những tester quyết tâm vạch trần mọi lỗi nhỏ trong code của bạn thực sự là một người bạn tốt.

Phần 61: One Binary

Tôi đã từng xem qua một vài project mà đội ngũ thiết kế phải viết lại một vài phần của source code để tạo ra một chương trình dành riêng cho từng môi trường nhất định. Điều này khiến mọi việc trở nên phức tạp hơn, và có khả năng khiến team gặp khó khăn trong xử lý các phiên bản trên từng cài đặt. Chí ít thì nó liên quan đến việc xây dựng nhiều bản copy gần giống hệt nhau của phần mềm, và mỗi bản phải được chuyển đến đúng nơi mà nó thuộc về. Điều này có nghĩa là chúng ta phải thực hiện công tác di chuyển nhiều hơn bình thường, song điều này khiến cho khả năng xảy ra sai sót cao hơn.

Tôi từng làm việc chung với team mà mỗi sự thay đổi tính năng đều phải được kiểm tra thật kỹ lưỡng trước khi bước vào chu kỳ xây dựng chính. Chính vì thế mà tester được giữ lại chờ đợi bất cứ khi nào họ cần chỉnh sửa một vài thay đổi nhỏ(Tôi đã từng nhắc đến việc xây dựng toàn bộ sẽ mất thời gian quá lâu chưa?). Tôi cũng làm việc với team sẵn sàng xây dựng lại mọi thứ từ đầu cho quá trình sản xuất(sử dụng cách thức cũ như chúng tôi đã làm). Điều này có nghĩa là chúng ta chả có bằng chứng nào về việc phiên bản trong quá trình sản xuất đã được kiểm tra kỹ lưỡng. Và nhiều thứ nữa.

Quy tắc rất đơn giản: hãy xây dựng một chương trình duy nhất mà bạn có thể xác định và thúc đẩy chúng thông qua tất cả các giai đoạn của quá trình phát hành. Hãy giữ chi tiết tối ưu cho từng môi trường trong môi trường của chúng. Ví dụ như là giữ chúng trong nơi chứa thành phần nhất định trong một file đã biết hoặc đường dẫn nào đó.

Nếu team bạn có một bản code chứa tất cả mọi thứ hoặc lưu trữ nhưng cài đặt quan trọng trong code điều này cho thấy không một ai đã suy nghĩ kĩ càng về việc thiết kế để chia các tính năng mà quan trọng đối với ứng dụng, cái nào dành cho các môi trường nhất định. Hoặc tệ hơn, team đó biết phải làm gì nhưng lại không thể quan trọng hóa nó để thực hiện các thay đổi.
Tất nhiên là sẽ có những trường hợp ngoại lệ. Bạn có thể phát triển cho đối tượng mà có sự khác biệt rõ rệt trong ràng buộc tài nguyên, nhưng nó không ảnh hưởng lớn đến những người mà viết ứng dụng truy xuất database liên tục. Hoặc là bạn đang ăn nằm với một vài mốt rắc rối đã cũ mà chúng quá khó để sửa chữa ngay bây giờ. Trong những trường hợp như thế này, bạn phải từng bước tiến dần giải quyết chúng nhưng hãy bắt đầu càng sớm càng tốt.

Và một điều nữa: hãy giữ cho những thông tin về môi trường bạn cần luôn được cập nhật.

Không việc gì có thể tệ hơn việc phá huỷ các thiết lập của môi trường và không thể biết được rằng chúng đã thay đổi những gì. Những thông tin về chúng luôn phải được cập nhật tách biệt khỏi code, bởi chính thay đổi ở những khoản khác nhau và thay đổi bởi nhiều lý do khác nhau. Chính vì thế các team nên dùng những hệ thống version control

như là bazzar hay Git, bởi chúng khiến cho việc thay đổi môi trường sản phẩm trở nên dễ dàng hơn rất nhiều, và cũng đừng quên lưu lại chúng.

Phần 62: Chỉ Có Code Mới Nói Lên Sự Thật

Các dòng code hoạt động chính là ý nghĩa cuối cùng của một chương trình. Sẽ rất khó đọc khi chúng chỉ ở dạng nhị phân. Nếu đây là chương trình của bạn, của bất kì nhà phát triển phần mềm thương mại điển hình hay dự án nguồn mở, hoặc là viết code bằng ngôn ngữ thông dịch linh hoạt thì nên cần có mã nguồn. Khi nhìn vào mã nguồn,ý nghĩa của chương trình sẽ được thể hiện rõ ràng. Nội dung trong mã nguồn là toàn bộ những gì bạn có thể chắc chắn về công dụng của chương trình. Thậm chí hầu hết các tài liệu yêu cầu tính chính xác cũng không nói lên tất cả: nó không chứa nội dung chi tiết về những gì mà chương trình thật sự làm, nó chỉ thể hiện những ý định tổng quát của nhà phân tích requirements. Tài liệu thiết kế có thể thu nạp bản thiết kế theo kế hoạch, nhưng sẽ thiếu sót các chi tiết thực hiện cần thiết. Những tài liệu ấy có thể sẽ không đồng bộ hóa được với công việc hiện tại… hoặc có thể bị mất, hoặc không thể được tìm thấy ở nguồn đầu tiên. Khi đó, mã nguồn chính là nơi lưu trữ cuối cùng.

Với ý tưởng này, hãy tự hỏi rằng mã code của bạn có thể thể hiện cụ thể với bạn hay những lập trình viên khác vai trò của nó hay không ?

Có thể bạn sẽ nói rằng, “Oh, những chú thích của tôi sẽ cho bạn biết tất cả những thứ cần thiết.” Nhưng hãy nhớ rằng những chú thích ấy chẳng làm cho code của bạn hoạt động. Chúng có thể mắc phải sai lầm như các dạng tài liệu khác. Có một truyền thống cho rằng việc có thêm các chú thích đó sẽ tốt hơn, vì thế không ngạc nhiên lắm khi các lập trình viên viết chúng ngày càng nhiều, thậm chí họ còn trình bày và giải thích lại những thứ hiển nhiên trong mã code. Điều này thật không đúng để làm rõ code của bạn. Nếu code của bạn cần những chú thích đó, thì hãy tái cấu trúc để nó không cần nữa. Những chú thích dài dòng sẽ làm cho không gian màn hình bị xáo trộn và có thể IDE của bạn sẽ tự động ẩn chúng đi. Nếu bạn muốn giải thích về một sự thay đổi nào

đó, hãy làm điều này trong thông báo đăng kí hệ thống kiểm soát phiên bản thay vì trong code.

Bạn làm gì để có thể khiến cho code của bạn tự nói lên sự thật càng cụ thể càng tốt? Hãy đặt tên đơn giản, dễ nhớ và sắp xếp cấu trúc code của bạn để chúng thể hiện được chức năng của mình. Hãy tách rời các đoạn code để được giao diện rõ ràng. Viết những bài kiểm tra tự động để giải thích các hành vi dự định và kiểm tra các giao diện. Hãy mạnh dạn sửa lại cấu trúc khi bạn học cách viết code đơn giản với những phương pháp tốt hơn. Hãy làm cho code của bạn càng đơn giản càng tốt để dễ đọc và dễ hiểu.

Hãy chăm chút cho code của bạn như thể đang biên soạn một bài thơ, bài diễn văn, một bài blog, hoặc là một tin email quan trọng. Hãy thể hiện sự khéo léo của bạn, nó sẽ thể hiện công dụng của nó và nói lên chính xác những gì nó đang làm, như thế nó vẫn biểu hiện được mục didschm dự định của bạn ngay cả khi bạn không ở bên cạnh. Hãy nhớ rằng những mã code hữu dụng đều được sử dụng lâu dài hơn dự định. Những lập trình viên bảo trì sẽ cảm ơn bạn. Và, nếu bạn là một lập trình viên bảo trì và code của bạn không dễ dàng thể hiện những gì bạn muốn, hãy chủ động áp dụng những hướng dẫn ở phía trên. Hãy giữ sự tỉnh táo trong khi thiết lập code của bạn.

Phần 63: Làm chủ và tái cấu trúc trình biên dịch

Không có gì là lạ khi các team có kỉ luật tốt trong việc viết code lại ít chú ý đến việc biên dịch các scripts(1), do họ nghĩ rằng chúng chỉ là một chi tiết không quan trọng hoặc do sợ rằng chúng phức tạp và đòi hỏi các kĩ thuật cao trong quá trình hoạt động. Mặt khác việc biên dịch scripts không thể duy trì cùng với sự trùng lặp và các lỗi gây ra những vấn đề tương tự như các dòng code không đủ mạnh.

Lý do về việc tại sao những nhà phát triển có tính kỉ luật và có kĩ năng xem biên dịch như là một công việc phụ của họ, đó khi biên dịch scripts ta thường viết bằng một ngôn

ngữ khác với mã nguồn. Một lí do nữa đó là build không hẳn là “code”. Những lập luận này xuất hiện trước một thực tế rằng hầu hết các nhà phát triển phần mềm đều ưa thích học các ngôn ngữ mới, và trình biên dịch là thứ tạo ra các sản phẩm để các nhà phát triển và người dùng chạy thử nghiệm. Code sẽ trở nên vô dụng nếu không được xây dựng, và trình biên dịch là thứ sẽ xác định thành phần cấu trúc của một ứng dụng. Nó cũng là một phần không thể thiếu trong quá trình phát triển, ra quyết định cho quá trình xây dựng có thể làm cho code và quá trình viết code đơn giản hơn nhiều.

Sử dụng các đặc ngữ trong quá trình viết scripts sẽ làm cho việc duy trì và quan trọng hơn là cải thiện chúng trở nên khó khăn hơn. Sẽ thật hợp lí nếu ta dành chút thời gian để hiểu đúng cách và tạo ra sự thay đổi. Các lỗi có thể xuất hiện khi ứng dụng được xây dựng sai cách hoặc khi cấu hình biên dịch không đúng.

Hoạt động kiểm thử truyền thống đã luôn được giao cho đội QA (Quality Asurance)(2). Đến bây giờ ta mới nhận ra được nhìn nhận việc kiểm thử như là viết code là điều thực sự cần thiết để có thể mang lại các giá trị theo một cách dễ hiểu hơn. Gần giống như vậy, quá trình biên dịch cần được sở hữu bởi các nhóm có sự phát triển.

Hiểu rõ được quá trình biên dịch có thể đơn giản hóa việc kéo dài tuổi thọ và giảm giá thành cho nó. Trình biên dịch đơn giản cho phép một một nhà phát triển mới bắt đầu một cách nhanh chóng và dễ dàng. Cấu hình tự động hóa trong biên dịch sẽ giúp bạn làm việc một cách hợp lí khi mà rất nhiều người làm việc trên cùng một project, tránh việc xung đột trong công việc. Nhiều công cụ biên dịch có khả năng hỗ trợ bạn chạy các báo cáo trên các dòng code chất lượng tốt và cho phép bạn cảm nhận các vấn đề tiềm năng một cách sớm nhất. Bằng cách dành ra thời gian nghiên cứu về cách biến trình biên dịch thành của mình, bạn có thể giúp đỡ cho mọi người trong nhóm và chính cả bản thân bạn nữa. Từ đó ta có thể tập trung vào việc viêt code và lợi ích hóa cho các bên liên quan, đồng thời làm cho công việc này thêm phần thú vị.

Hãy am hiểu quá trình biên dịch của bạn để biết được khi nào và làm thế nào để thực hiện những sự thay đổi hợp lí. Biên dịch scripts chính là viết code. Chúng thật sự quá quan trọng để giao phó cho một người khác, nếu không vì lí do nào khác ngoài việc ứng dụng chưa được hoàn thành cho đến khi nó được biên dịch. Công việc của

chương trình sẽ chưa được hoàn thiện cho đến khi ta cung cấp phần mềm làm việc chính xác cho nó.

(*) Chú thích:
(1) Scripts: Script hay Scripting Language (hay Ngôn ngữ Script) chính là Ngôn ngữ kịch bản. Một ngôn ngữ kịch bản là một ngôn ngữ mà không đòi hỏi một bước biên dịch. Ngôn ngữ kịch bản thường thông dịch (Interpreted) thay vì biên dịch.
(2) Quality Asurance: QA (Quality Assurance) có nhiệm vụ giám sát, quản lý và đảm bảo chất lượng của việc xây dựng hệ thống, quy trình sản xuất của công ty theo một chuẩn mực chất lượng. Quản lý chặt chẽ các tiêu chuẩn chất lượng trong tất cả các giai đoạn từ khâu nghiên cứu thị trường, thiết kế … cho đến khâu sản xuất ra sản phẩm cuối cùng và bán hàng, tiêu thụ trên thị trường, chăm sóc khách hàng.

Phần 64: Ghép chương trình và cảm nhận dòng chảy

Hãy tưởng tượng bạn đang hoàn toàn chìm đắm trong công việc của bản thân- tập trung, tận lực và đặt hết tâm trí vào đó. Bạn không để ý thời gian trôi qua. Bạn cảm thấy hạnh phúc. Bạn đang cảm nhận dòng chảy. Thật khó để vừa đạt được vừa duy trì dòng chảy cho cả nhóm vì có quá nhiều sự gián đoạn, tương tác và phiền nhiễu khác có thể dễ dàng phá vỡ nó.

Nếu bạn từng thực hành lập trình cặp, có lẽ bạn đã quen với việc ghép đôi đóng góp như nào vào dòng chảy. Còn nếu bạn chưa thử, chúng tôi muốn sử dụng kinh nghiệm của mình để thúc đẩy bạn bắt đầu ngay bây giờ! Để thành công với lập trình cặp, từng thành viên và cả nhóm đều phải nỗ lực.

Là một thành viên trong nhóm, hãy kiên nhẫn với các nhà phát triển ít kinh nghiệm hơn; hãy đối mặt với nỗi sợ về việc bị đe dọa bởi các nhà phát triển lành nghề hơn. Nhận thức được rằng mỗi người mỗi khác và trân trọng điều đó. Tự nhận thức điểm mạnh và

điểm yếu của riêng mình, cũng như của các thành viên khác trong nhóm. Bạn có thể ngạc nhiên với những gì bạn học được từ đồng nghiệp của mình.

Là một nhóm, lập trình cặp có thể thúc đẩy phân phối các kỹ năng và kiến thức trong suốt dự án. Bạn nên giải quyết nhiệm vụ của mình theo cặp và luân phiên các cặp cũng như nhiệm vụ thường xuyên. Thỏa thuận một quy tắc xoay vòng. Đặt quy tắc sang một bên hoặc điều chỉnh khi cần thiết. Theo kinh nghiệm của chúng tôi, bạn không nhất thiết phải hoàn thành một nhiệm vụ trước khi chuyển nó sang một cặp khác. Gián đoạn một nhiệm vụ để chuyển nó sang một cặp khác nghe có vẻ phản trực giác, nhưng chúng tôi đã thấy nó hoạt động.

Có rất nhiều tình huống mà dòng chảy có thể bị phá vỡ, nhưng lập trình cặp sẽ giúp bạn duy trì nó:

- Giảm "yếu tố xe tải": Đó là một thử nghiệm hơi quái dị, nhưng có bao nhiêu thành viên trong nhóm sẽ bị xe tải đâm trước khi nhóm không thể hoàn thành đơn hàng cuối cùng? Nói cách khác, sự phân phối của bạn phụ thuộc thế nào vào các thành viên trong nhóm? Kiến thức là độc quyền hay để chia sẻ? Nếu bạn luân phiên nhiệm vụ giữa các cặp, sẽ luôn có người có đủ kiến thức để hoàn thành công việc. Dòng chảy của nhóm sẽ không bị ảnh hưởng bởi "yếu tố xe tải".
- Giải quyết vấn đề hiệu quả: Nếu bạn đang lập trình cặp và bạn gặp phải một vấn đề khó, bạn luôn có người để cùng thảo luận. Trao đổi như vậy có nhiều khả năng đưa ra giải pháp hơn là mắc kẹt với chính mình. Khi công việc xoay vòng, giải pháp của bạn sẽ được xem xét lại bởi các cặp tiếp theo; như vậy, nếu ban đầu bạn không đưa ra được giải pháp tối ưu thì cũng không sao.
- Tích hợp trơn tru: Nếu nhiệm vụ hiện tại của bạn liên quan đến một đoạn code khác, bạn cần đảm bảo tên của các phương thức, tài liệu và thử nghiệm đủ để bạn nắm bắt chúng. Nếu không, việc kết hợp với một nhà phát triển có liên quan đến việc viết code đó sẽ giúp bạn có cái nhìn tổng quát hơn và tích hợp nhanh hơn vào code của riêng bạn. Ngoài ra, bạn có thể coi cuộc thảo luận ấy như một cơ hội để cải thiện việc đặt tên, tài liệu và thử nghiệm.
- Giảm thiểu gián đoạn: Nếu ai đó đến hỏi bạn một câu hỏi, bạn có điện thoại hay bạn phải trả lời một email khẩn cấp hoặc tham dự một cuộc họp, cộng sự của

bạn có thể tiếp tục viết code. Khi bạn trở về, cộng sự của bạn vẫn đang trong dòng chảy và bạn có thể nhanh chóng bắt kịp và tham gia lại.

- Thành viên mới tiến bộ nhanh chóng: Với lập trình cặp và xoay vòng các cặp cũng như nhiệm vụ phù hợp, người mới sẽ nhanh chóng làm quen được với code và các thành viên khác trong nhóm.

Dòng chảy sẽ giúp bạn nâng cao năng suất. Nhưng nó cũng dễ bị phá huỷ. Làm những gì bạn có thể để có được nó, và duy trì khi bạn đã sỡ hữu nó!

Phần 65: Kiểu Miền Chuyên Biệt Được Ưa Chuộng Hơn Là Kiểu Nguyên Thủy

Vào ngày 23 tháng 9 năm 1997, chiếc tàu quỹ đạo Khí hậu Sao Hỏa đã bị thất lạc khi đang tiến vào quỹ đạo Sao Hỏa vì một lỗi phần mềm khi trở về Trái Đất. Sau này, lỗi ấy được gọi là metric mix-up. Trạm phần mềm tại mặt đất đã phải làm việc rất cật lực trong khi con tàu vũ trụ thì lại cần nỗ lực nhiều hơn thế, và trạm phần mềm mặt đất đã đánh giá thấp sức mạnh của các máy đẩy trong tàu vũ trụ với hệ số 4.45.

Đây là một trong các ví dụ về sự thất bại phần mềm có thể được ngăn chặn nếu được áp dụng nhiều và chắc các kiểu gõ miền chuyên biệt (domain-specific). Đây cũng là một ví dụ cho lí luận về các tính năng của ngôn ngữ Ada, một trong các mục tiêu thiết kế chính của những tính năng đó là hoàn thiện Quan trọng an toàn phần mềm nhúng (embedded safety-critical software). Ada có tính vững mạnh trong việc kiểm tra kiểu tĩnh cho cả hai kiểu nguyên thủy và kiểu do người dùng định (user-defined):

type Velocity_In_Knots is new Float range 0.0 .. 500.00;
type Distance_In_Nautical_Miles is new Float range 0.0 .. 3000.00;
Velocity: Velocity_In_Knots;
Distance: Distance_In_Nautical_Miles;
Some_Number: Float;
Some_Number:= Distance + Velocity; -- Will be caught by the compiler as a type error.

Các nhà phát triển ở lĩnh vực ít đòi hỏi hơn cũng có lợi từ việc sử dụng nhiều hơn kiểu gõ domain-specific, nơi họ có thể sử dụng kiểu dữ liệu nguyên thủy được cung cấp bởi ngôn ngữ và thư viện của nó, như là strings và floats. Trong ngôn ngữ Java, C++, Python và các ngôn ngữ hiện đại khác, kiểu dữ liệu trừu tượng được xem như là lớp (class). Sử dụng các lớp như là Velocity_In_Knots và Distance_In_Nautical_Miles sẽ tăng thêm giá trị nối liền với chất lượng code.

- Code trở nên dễ đọc vì nó thể hiện nhiều khái niệm tuyệt đối của một lĩnh vực chứ không chỉ là String hay Float.
- Code trở nên có thể kiểm tra vì nó đóng gói (encapsulates) các hành vi có thể dễ dàng kiểm tra.
- Code tạo điệu kiện để tái sử dụng các ứng dụng và hệ thống.

Sử dụng ngôn ngữ đánh máy tĩnh hay động đều mang đến cho người dùng giá trị như nhau. Điểm khác nhau duy nhất là các nhà phát triển sử dụng ngôn ngữ đánh máy tĩnh cần sự giúp đỡ của trình biên dịch trong khi những người sử dụng ngôn ngữ đánh máy động lại có xu hướng dựa vào các bài kiểm tra đơn vị của họ. Phong cách kiểm tra có thể khác nhau, nhưng mục đích và cách thể hiện thì không.

Và lời khuyên là hãy bắt đầu khám phá các kiểu domain-specific vì mục đích phát triển phần mềm chất lượng.

Phần 66: Ngăn ngừa lỗi

Thông báo lỗi là tương tác then chốt giữa người dùng và hệ thống. Chúng xảy ra khi giao tiếp giữa người dùng và hệ thống lâm vào bế tắc.

Thật dễ dàng khi cho rằng lỗi là do người dùng nhập input sai. Nhưng lỗi sai của con người có hệ thống và khá dễ đoán. Vì vậy, có thể sửa lỗi giao tiếp giữa người dùng và hệ thống theo cách bạn làm giữa các thành phần khác nhau trong hệ thống.

Chẳng hạn bạn muốn người dùng nhập một ngày trong một phạm vi nhất định. Thay vì cho phép người dùng nhập một ngày bất kỳ, hãy cung cấp một danh sách hay lịch hiển thị các ngày trong phạm vi đó. Điều này giúp loại bỏ nguy cơ người dùng nhập một ngày ngoài phạm vi đã cho.

Lỗi định dạng là một vấn đề phổ biến khác. Chẳng hạn, nếu người dùng được cung cấp văn bản Ngày và nhập một ngày tuỳ ý, ví dụ "ngày 29 tháng 7 năm 2012" thì sẽ khá khó hiểu nếu họ từ chối chỉ vì nó không ở định dạng ưa thích (chẳng hạn như "DD/ MM/ YYYY"). Tệ hơn nữa là từ chối "29 / 07 / 2012" chỉ vì nó chứa thêm khoảng cách- loại vấn đề này đặc biệt khó hiểu đối với người dùng vì ngày dường như luôn ở định dạng họ mong muốn.

Lỗi này xảy ra vì việc từ chối ngày dễ dàng hơn so với phân tích ba hoặc bốn định dạng ngày phổ biến nhất. Những lỗi lặt vặt này dẫn đến sự thất vọng của người dùng, kéo theo nhiều lỗi khi họ mất tập trung. Thay vào đó, hãy tôn trọng sở thích của người dùng để nhập thông tin.

Một cách khác để tránh lỗi định dạng là cung cấp tín hiệu- ví dụ một nhãn hiển thị định dạng mong muốn ("DD/ MM/ YYYY"). Một gợi ý khác là có thể chia phạm vi đã cho thành ba hộp văn bản gồm hai, hai và bốn ký tự.

Tín hiệu khác với hướng dẫn: Tín hiệu thường có xu hướng gợi ý; trong khi hướng dẫn thường khá dài dòng. Tín hiệu xảy ra tại thời điểm tương tác; hướng dẫn xuất hiện trước thời điểm tương tác. Tín hiệu cung cấp bối cảnh; hướng dẫn thì sử dụng chúng.

Nói chung, hướng dẫn không hiệu quả trong việc ngăn ngừa lỗi. Người dùng có xu hướng cho rằng các giao diện sẽ hoạt động phù hợp với trải nghiệm trước đây của họ (Hẳn mọi người đều biết "ngày 29 tháng 7 năm 2012" nghĩa là gì). Vì vậy, hướng dẫn thường bị bỏ qua. Trong khi đó tín hiệu giúp người dùng tránh xa các lỗi.

Một cách khác để tránh lỗi là cung cấp mặc định. Người dùng thường nhập các giá trị tương ứng với hôm nay, ngày mai, sinh nhật của tôi, deadline hoặc ngày tôi nhập lần

trước khi tôi sử dụng biểu mẫu này. Tùy thuộc vào ngữ cảnh, một trong số này có thể là một lựa chọn tốt như một mặc định thông minh.

Dù nguyên nhân là gì, các hệ thống nên tiếp nhận được lỗi. Bạn có thể làm điều này bằng cách cung cấp nhiều cấp độ hoàn tác cho tất cả hành động, đặc biệt với những hành động có khả năng phá hủy hoặc sửa đổi dữ liệu của người dùng.

Ghi chép và phân tích các hành động hoàn tác cũng có thể làm nổi bật giao diện đang khiến người dùng mắc lỗi trong vô thức, chẳng hạn như nhấp liên tục vào nút sai. Những lỗi này thường được gây ra bởi tín hiệu sai lệch hoặc do trình tự tương tác mà bạn hoàn toàn có thể thiết kế lại để tránh những lỗi về sau.

Bất kể cách tiếp cận nào bạn thực hiện, hầu hết các lỗi đều theo một hệ thống nhất định. Đó là hệ quả của sự hiểu lầm giữa người dùng và phần mềm. Thấu hiểu suy nghĩ của người dùng, giải thích thông tin, đưa ra quyết định và dữ liệu đầu vào sẽ giúp bạn sửa các lỗi tương tác giữa phần mềm và người dùng.

Phần 67: Một Lập Trình Viên Chuyên Nghiệp

Như thế nào là một lập trình viên chuyên nghiệp?

Đặc điểm quan trọng nhất của một người lập trình viên chuyên nghiệp là có trách nhiệm cá nhân. Họ chịu trách nhiệm cho sự nghiệp, các ước tính, các lời cam kết, những sai lầm và tay nghề của họ. Một người lập trình viên chuyên nghiệp sẽ không bao giờ đùn đẩy trách nhiệm của họ cho người khác.

- Nếu bạn là một người chuyên nghiệp, bạn sẽ chịu trách nhiệm cho sự nghiệp của mình. Bạn có trách nhiệm học và đọc. Bạn phải luôn giữ mình cập nhật những thứ về công nghiệp và công nghệ. Đã có quá nhiều người lập trình nghĩ

rằng cấp trên phải làm những việc đó để đào tạo họ. Đáng tiếc, đây là một sai lầm chết người. Bạn có nghĩ rằng bác sĩ, luật sư sẽ nghĩ như vậy không? Không, họ tự đào tạo bản thân bằng thời gian và sức lực của mình. Họ dành ra nhiều thời gian ngoài giờ để đọc biên bản và các quyết định. Họ giữ cho mình luôn ở trạng thái cập nhật mọi thứ. Chúng ta cũng nên như vậy. Mối qua hệ giữa bạn và cấp trên đã được nêu ra rõ ràng trong hợp đồng của công ty. Nói một cách ngắn gọn: Họ cam kết sẽ trả lương cho bạn, và bạn cam kết hoàn thành tốt công việc.

- Những người chuyên nghiệp chịu trách nhiệm cho các mã code họ viết. Họ không phát hành các mã code trừ khi họ chắc rằng nó hoạt động. Hãy nghĩ về điều đó một chút. Làm sao bạn có thể tự cho rằng mình là một người chuyên nghiệp khi bạn thực sự muốn đưa ra mã code mà bạn không tự tin về nó ? Lập trình viên chuyên nghiệp mong đợi QA (Quality Assurance) sẽ không tìm thấy bất kì lỗi nào bởi vì họ không ra mắt mã code không được thử nghiệm kỹ lưỡng. Tất nhiên QA vẫn tìm được một vài lỗi, bởi vì không có gì là tuyệt hảo. Nhưng là một người chuyên nghiệp, ta phải có thái độ làm việc như vậy thì QA sẽ chẳng tìm được lỗi nào của ta.

- Những người chuyên nghiệp là một đội người chơi. Họ sẽ chịu trách nhiệm cho kết quả của cả đội, chứ không riêng gì công việc của họ. Họ giúp đỡ, học hỏi, truyền đạt kinh nghiệm lẫn nhau, và thậm chí bảo vệ, an ủi lẫn nhau khi cần thiết. Mỗi khi một người gặp khó khăn, những người khác sẽ giúp đỡ vì họ biết rằng sẽ có một lúc nào đó họ sẽ là người cần được che chở.

- Người chuyên nghiệp sẽ không chấp nhận có một danh sách bug lớn. Có được danh sách bug lớn như vậy thể hiện tính cẩu thả. Những hệ thống chứa hàng ngàn các vấn đề trong cơ sỡ dữ liệu theo dõi vấn đề là một chuỗi bi kịch của sự bất cẩn. Sự thật là, ở đa số các dự án, nhu cầu hệ thống theo dõi các vấn đề (issue tracking system) là một dấu hiệu cho sự bất cẩn. Chỉ có những hệ thống lớn nhất mới có những danh sách lỗi dài đến mức cần tự động hóa để có thể xử lí chúng.

- Những người chuyên nghiệp không tạo ra mớ hỗn độn. Họ tự hào về tay nghề của họ. Họ xây dựng mã code của họ theo cách tốt nhất, gọn gàng và dễ đọc. Họ tuân theo những tiêu chuẩn và sự luyện tập tốt nhất. Họ không bao giờ vội vàng. Hãy tưởng tượng rằng bạn có một trải nghiệm được ra khỏi cơ thể của mình và một xem bác sĩ thực hiện phẫu thuật tim ngay trên người bạn. Vị bác sĩ này đang có thời hạn phẫu thuật (deadline). Anh ta phải hoàn thành trước khi máy tim phổi nhân tạo tổn hại quá nhiều tế bào máu của bạn. Bạn muốn anh ta phải thực hiện như thế nào? Bạn có muốn anh ta hành động như các nhà phát triển phần mềm điển hình, vội vàng và tạo nên mớ hỗn độn? Bạn có muốn anh ta sẽ nói rằng: “Tôi sẽ quay lại và sửa nó sau?” Hay bạn muốn anh ta sẽ theo khuôn phép, bình tĩnh, tự tin rằng cách làm của anh ấy là cách tốt nhất có thể thực hiện. Bạn muốn sự bừa bộn, hay là tính chuyên nghiệp?

Những người trách nhiệm luôn đáng tin cậy. Họ chịu trách nhiệm cho sự nghiệp của mình. Họ chịu trách nhiệm về việc bảo đảm các mã code của họ chắc chắn sẽ hoạt động tốt. Họ chịu trách nhiệm cho chất lượng tay nghề của họ. Họ sẽ không bỏ qua các nguyên tắc dù các kì hạn (deadlines) đang đến gần. Trên thực tế, khi áp lực dần xuất hiện, người chuyên nghiệp sẽ bám sát vào các nguyên tắc mà họ cho là đúng.

Phần 68: Lưu giữ mọi thứ bằng version control

Hãy lưu tất cả các dự án của bạn bằng version control. Mọi tài nguyên của bạn đều đã có sẵn, như là Subversion, Git, Mercurial, và CVS; không gian lưu trữ lớn, server mạnh mẽ và hoàn toàn miễn phí, mạng lưới toàn cầu, và gồm cả dịch vụ project-hosting. Sau khi bạn đã hoàn tất cài đặt version control, tất cả mọi thứ bạn cần làm để lưu trữ công việc vào kho riêng là thực hiện một vài dòng lệnh cơ bản tại nơi chứa code của bạn. Và chúng chỉ có hai thao tác cơ bản bạn cần phải học là: xác nhận những sự tinh chỉnh trong code với kho lưu trữ và bạn phải cập nhật phiên bản cũ của dự án của bạn trong kho.

Một khi bạn đã lưu trữ dự án bằng version control, bạn có thể xem lịch sử của nó một cách trực quan, xem rằng ai đã viết đoạn code đó, và tham khảo các phiên bản của tệp tin hoặc dự án bằng một định danh duy nhất. Quang trọng hơn nữa, bạn có thể mạnh dạn thực hiện những thay đổi mà không cần comment trừ khi bạn thực sự cần nó trong tương lai, bởi vì mọi phiên bản chỉnh sửa sẽ được lưu trữ một cách vô cùng an toàn. Bạn còn có thể(nên) gán một bản chương trình hoàn thiện bằng một cái tên gợi nhớ để trong tương lai bạn có thể truy xuất chính xác phiên bản mà khách hàng của bạn sử dụng. Hơn nữa bạn còn có thể tạo ra thêm các nhánh chương trình song song để phát triển. Hầu hết các project đều có một nhánh phát triển chính cùng với nhiều nhánh là các phiên bản được phát hành và hỗ trợ.

Version control giảm thiểu tối đa sự thiếu tương tác giữa những nhà phát triển. Khi các lập trình viên làm việc trên các mảng độc lập với nhau và tất cả những mảng ấy được kết hợp lại như được thực hiện bởi “ma thuật”. Khi họ thực hiện thao tác cùng một lúc, hệ thống sẽ nhận ra và cho phép họ sắp xếp những sự mâu thuẫn ấy. Đồng thời khi chúng ta thực hiện thêm một vài thiết lập đặc biệt, hệ thống sẽ có thể nhận biết tất cả nhà phát triển với từng sự thay đổi, giúp ta thiết lập sự theo dõi chung về tiến độ của dự án.

Khi chuẩn bị cho dự án của bạn, đừng trở nên keo kiệt: hãy đặt tất cả chúng dưới sự bảo quản của version control. Tách biệt khỏi source code, kể cả tài liệu, dụng cụ, bản kế hoạch, test case, nghệ thuật, và cả thư viện. Cùng với sự bảo mật chặt chẽ, do thường xuyên backup, kho lưu trữ của bạn khiến cho khả năng bị mất dữ liệu trở nên không đáng kể. Chính vì thế mà việc thiết lập một môi trường phát triển mới trên một thiết bị mới vô cùng tiện lợi và đơn giản như kiểm tra dự án từ kho lưu trữ. Điều này giúp tối giản sự phân phối, xây dựng, và kiểm tra code trên các nền tảng khác nhau. Và trên mỗi thiết bị chỉ với một lệnh cập nhật đơn giản sẽ đảm bảo cho dự án luôn ở phiên bản mới nhất.

Một khi bạn đã thấy được sự tiện khi làm việc với version control, chỉ cần thực hiện theo một số quy tắc cơ bản sau đây sẽ giúp cho bạn và team làm việc hiệu quả hơn:

- Xác nhận mọi sự thay đổi hợp lý trong một hoạt động riêng biệt. Kết hợp nhiều sự thay đổi cùng lúc chỉ với một lệnh xác nhận sẽ khiến chúng trở nên khó để

tách rời khi cần thiết trong tương lai. Điều này cực kỳ quan trọng khi bạn tiến hành tải bản hay thay đổi kiểu cách của toàn bộ code, điều mà rất dễ gặp khó khăn trong quá trình thay đổi.

- Kèm theo mọi sự xác nhận thay đổi là một tin nhắn giải thích lý do. Chí ít là nội chính của sự thay đổi ấy nhưng nếu bạn muốn ghi chép lý do của sự thay đổi ấy thì đây chính là nơi hoàn hảo.
- Cuối cùng, tránh xác nhận sự thay đổi code mà khiến cho cả dự án của bạn sụp đổ, hoặc bạn sẽ bị xa lánh bởi các nhà phát triển khác.

Cuộc sống khi có version control là quá tốt để bị phá huỷ bởi các lỗi có thể tránh được.

Phần 69: Chia tay chuột và bàn phím

Đã bao giờ bạn dành hàng giờ tập trung giải quyết một vấn đề khó khăn tuy nhiên bạn lại trở nên cạn kiệt ý tưởng. Vì lý do ấy, bạn hãy đứng dậy duỗi thẳng gân cốt và tìm đến một máy bán nước tự động, và khi bạn trở về câu trả lời đột nhiên xuất hiện trong đầu của bạn.

Bạn cảm thấy kịch bản này có vẻ quen thuộc? Và băn khoăn tại sao nó lại xảy ra? Một mẹo nhỏ ở đây chính là trong khi bạn đang viết code, phần bán cầu não phụ trách tư duy logic của bạn được active trong khi phần sáng tạo thì ngược lại. Nó không thể nảy ra một ý tưởng nào cho đến khi phần logic tạm thời nghỉ ngơi.

Và đây chính là một ví dụ thực tế. Trong lúc tôi đang dọn dẹp mớ code cũ của bản thân thì tình cờ tìm thấy một method thứ vị. Nó được thiết kế để kiểm tra một chuỗi chứa thông tin về thời gian có được định dạng theo chuẩn hh:mm:ss xx, với hh đại diện cho giờ, mm đại diện cho phút, ss đại diện cho giây, và xx đại cho AM hoặc PM.

Cái method ấy sử dụng đoạn code dưới đây để chuyển hai ký tự (đại diện cho giờ) thành một số và kiểm tra rằng chúng nằng trong một phạm vi thích hợp:

```
try {
        Integer.parseInt(time.substring(0,2));
}catch (Exception x){
        Return false;
```

```
}
if (Integer.parseInt(time.substring(0,2)) > 12) {
        Return false;
}
```

Và đoạn code như thế xuất hiện thêm hai lần nữa cùng với sự tinh chỉnh thích hợp cho từng loại và giới hạn thích hợp để kiểm tra phút và giây. Method nảy kết thúc cùng với những dòng này để kiểm tra AM hay PM:

```
if (!time.substring(9, 11).equals("AM") & !time.substring(9, 11).equals("PM")){
        return false;
}
```

Nếu không phép so sánh nào trong chuỗi so sánh trên thất bại, trả về false, và method trả về true.

Nếu bạn cảm thấy đoạn code phía trên cứng nhắc và khó có thể theo kịp thì đừng lo lắng. Bản thân tôi cũng thấy thế nên tôi đã tìm thấy thứ mà mình cần phải dọn dẹp. Tôi tiến hành tái bản nó đồng thời tạo một số bộ test cho nó chỉ để đảm bảo nó vẫn hoạt động tốt.

Khi hoàn tất mọi việc, tôi cảm thấy vô cùng mãn nguyện với thành quả của mình. Phiên bản mới đã dễ đọc hơn, chỉ bằng một nửa kích thước của bản cũ, và còn chính xác hơn bởi ở bản gốc bởi nó chỉ kiểm tra giới hạn trên cho giờ, phút, giây.

Sau đó, trong khi tôi đang chuẩn bị cho công việc của ngày mai, một ý tưởng nảy lên trong đầu tôi. Tại mình lại không dùng regular expression ngay từ đầu để duyệt chuỗi nhỉ? Chỉ sau một vài phút gõ phím, tôi đã hoàn thành một bản cài đặt mới hoạt động hoàn toàn bình thường chỉ trong một dòng code. Và nó đây:

```
Public static boolean validateTime(String time) {
        return time.matches("(0[1-9]|1[0-2]):[0-5][0-9]:[0-5][0-9] ([AP]M)");
}
```

Mấu chốt của câu chuyện không phải là việc tôi thay thế tất cả ba mươi dòng code chỉ bằng một dòng duy nhất mà mấu chốt ở đây chính là cho tới khi tôi tạm chia tay cái máy tính thì tôi mới nghĩ ra giải pháp tối ưu cho bài toán của tôi.

Chốt lại vấn đề, lần sau nếu bạn gặp phải một vấn đề khó khăn, hãy cho bản thân mình một ân huệ. Một khi bạn đã thật sự hiểu vấn đề hãy đi làm gì đó kích thích phần sáng tạo ở não của bạn. Hãy mô tả vấn đề của bạn vào giấy, thêm vào một chút nhạc, và đưa bản thân dạo một vòng. Đôi khi điều tốt nhất bạn có thể làm để giải quyết một vấn đề đó chính là tạm chia tay chú chuột và cái bàn phím thân thương của các bạn.

Phần 70: Đọc Code

Lập trình viên chúng ta là những con người kì lạ. Chúng ta yêu thích viết code. Nhưng chúng ta thường e ngại khi phải đọc chúng. Sau cùng thì, việc viết code lúc nào cũng mang lại sự hứng thú hơn so với đọc code, và rất khó để đọc được chúng - đôi lúc việc này là không thể, đặc biệt là đọc code của một người nào đó. Không nhất thiết là do code của họ tệ, có thể họ có tư duy và cách giải quyết vấn đề khác hoàn toàn so với bạn thôi. Nhưng bạn có bao giờ tự nghĩ rằng việc đọc code của người khác sẽ giúp bạn hoàn thiện khả năng của mình hơn không ?

Khi bạn đọc code, hãy vừa đọc vừa ngẫm nghĩ một chút. Có phải code rất khó hiểu không ? Nếu vậy thì tại sao ? Có phải do cách định dạng tệ không ? Là do cách đặt tên không thống nhất và phi logic ? Hay là những điều quan trọng bạn quan tâm đều nằm cùng trong một đoạn mã ? Hoặc có lẽ là do việc lựa chọn ngôn ngữ đã khiến bạn không thể đọc được chúng. Hãy thử học hỏi từ lỗi sai của người khác, như thế bạn sẽ không mắc phải những lỗi ấy trong code của bạn. Có thể bạn sẽ bất ngờ. Ví dụ, các kĩ thuật dependency-breaking có thể có ích cho low coupling, nhưng đôi khi chúng sẽ làm cho việc đọc code trở nên khó khăn. Và đó cũng là điều mà vài người gọi là elegant code, số khác gọi là unreadable.

Nếu bạn nghĩ rằng việc đọc code là dễ dàng, hãy dừng lại để tìm kiếm những thứ bạn cho là có ích để học từ nó. Có thể đó là một mẫu mã thiết kế đang được sử dụng mà

bạn không biết, hoặc là đã từng được thêm vào để bổ sung. Có lẽ những phương pháp đó sẽ ngắn gọn hơn và tên của chúng thì có nghĩa hơn của bạn. Một vài dự án nguồn mở có rất nhiều ví dụ hữu dụng về cách viết code thông minh, dễ đọc - trong khi các dự án khác lại hoàn toàn có vai trò ngược lại! Bạn hãy xem qua vài đoạn code của họ.

Đọc những bài code cũ của bạn, từ dự án mà bây giờ bạn không thực hiện, cũng là một cách để bạn hoàn thiện và rút kinh nghiệm cho sau này. Hãy bắt đầu với những bài code cũ nhất của bạn và làm việc theo cách của bạn bây giờ. Chắc chắn bạn sẽ nhận ra rằng thật không dễ chút nào để đọc như khi bạn viết nó. Những bài code đầu tiên hẳn sẽ khiến bạn có chút vui vẻ ngượng ngùng, cũng giống như việc bạn bị gợi nhớ về những câu nói của bạn trong lúc say mèm trong quán rượu tối qua. Nhìn xem, kĩ năng của bạn đã được cải thiện qua nhiều năm - và điều này thực sự là một nguồn động lực thúc đẩy bạn. Hãy xem xét những dòng code khó đọc, và suy nghĩ xem liệu bây giờ bạn đã thay đổi cách viết code của mình hay chưa.

Vậy là giờ bạn đã biết bạn cần gì để phát triển kĩ năng lập trình của mình, đừng đọc quyển sách khác. Hãy đọc code.

Phần 71: Đọc vị nhân loại

Trong tất cả những dự án từ nhỏ nhất con người làm việc với con người. Trong cả những lĩnh vực nghiên cứu trừu tượng nhất, con người viết phần mềm để hỗ trợ con người hoàn thành mục tiêu. Con người viết phần mềm với người khác dành cho con người. Đó là chuyện giữa người với người. Nhưng thật không may, những gì các lập trình viên được dạy trang bị cho họ rất ít cách để đối phó với những người họ làm việc cùng. May mắn thay, có hẳn một lĩnh vực nghiên cứu hỗ trợ điều này.

Ludwig Wittgenstein đã đưa ra một giả thuyết rất hay trong “Điều tra triết học” rằng bất kỳ ngôn ngữ nào chúng ta sử dụng đều không thể chuyển tiếp ý nghĩ, ý tưởng hay hình ảnh từ người này sang người khác. Chúng ta cần cẩn thận tránh hiểu lầm khi chúng ta “thu thập yêu cầu”. Wittgenstein cũng cho thấy rằng sự am hiểu lẫn nhau không xuất phát từ các định nghĩa được chia sẻ, mà từ nền tảng kinh nghiệm chung. Đây có thể là

một lý do tại sao các lập trình viên thành thạo lĩnh vực của bản thân thường làm tốt hơn những người không thuộc lĩnh vực đó.

Lakoff và Johnson cho chúng tôi xem một danh mục các phép ẩn dụ mà chúng ta thường sử dụng, cho thấy rằng ngôn ngữ chủ yếu là các phép ẩn dụ và những ẩn dụ này cung cấp một cái nhìn sâu sắc về việc chúng ta hiểu thế giới như nào. Ngay cả những thuật ngữ cụ thể như “dòng tiền” mà chúng ta sử dụng khi nói về một hệ thống tài chính, cũng được coi là một phép ẩn dụ “tiền là một dạng chất lỏng”. Phép ẩn dụ đó có ảnh hưởng đến cách chúng ta nghĩ về hệ thống xử lý tiền như thế nào? Hoặc chúng ta có thể nói về các lớp trong một chồng biên bản, với một số mức cao và một số mức thấp. Điều này mang tính ẩn dụ mạnh mẽ: người dùng “lên” và công nghệ “xuống”. Điều này phơi bày suy nghĩ của chúng tôi về cấu trúc các hệ thống chúng tôi xây dựng. Nó cũng có thể đánh dấu một thói quen lười biếng rằng đôi khi chúng ta có thể hưởng lợi từ việc phá luật.

Martin Heidegger đã nghiên cứu kỹ cách mọi người trải nghiệm các công cụ. Các lập trình viên xây dựng và sử dụng công cụ, chúng tôi nghĩ, tạo nên, sửa đổi và tái tạo công cụ. Công cụ là đối tượng chúng tôi quan tâm. Nhưng đối với người dùng, như Heidegger thể hiện trong “Bản thể và Thời gian”, một công cụ là một thứ vô hình chỉ được tìm hiểu khi sử dụng. Đối với người dùng, công cụ chỉ được quan tâm khi chúng không hoạt động. Sự khác biệt này đáng được nhấn mạnh bất cứ khi nào khả năng sử dụng được bàn luận đến.

Eleanor Rosch đã đảo ngược mô hình Aristoteles mà chúng tôi dùng để sắp xếp sự hiểu biết của chúng tôi về thế giới. Khi lập trình viên hỏi người dùng về hệ thống mà họ mong muốn, chúng ta có xu hướng yêu cầu các định nghĩa được xây dựng từ các vị từ. Điều này rất thuận tiện cho chúng tôi. Các thuật ngữ trong vị từ có thể dễ dàng trở thành các thuộc tính trong class hoặc các cột trong bảng. Thật không may, như Rosch đã thể hiện trong “Thể loại tự nhiên” và những nghiên cứu sau đó, đó không phải là cách nhân loại nói chung hiểu về thế giới. Họ hiểu nó dựa trên các ví dụ. Một số ví dụ (được gọi là nguyên mẫu) tốt hơn so với các nguyên mẫu khác, cho ra kết quả chồng chéo với cấu trúc phức tạp. Chừng nào chúng tôi còn nhấn mạnh vào câu trả lời của

Aristoteles, thì chúng tôi không thể hỏi người dùng đúng câu hỏi về thế giới của họ và sẽ rất khó khăn để đạt được sự thấu hiểu chung mà chúng tôi cần.

Phần 72: Đôi khi hãy tái phát minh bánh xe

“Hãy dùng thứ gì đó có sẵn – tái phát minh bánh xe là một điều ngớ ngẩn”

Bạn có bao giờ nghe điều này hay những câu đại loại như thế? Chắc chắn bạn đã từng nghe qua. Tại sao lại như thế? Tại sao việc tái phát minh bánh xe lại khiến người ta trở nên khó chịu? Nguyên nhận là thông thường các đoạn code có sẵn là những đoạn code hoạt động được. Nó chắc chắn đã trải qua một vài phương thức kiểm định chất lượng nghiêm ngặt, và hoạt động cùng thành công. Hơn nữa đầu tư thời gian vào việc tái phát minh bánh xe hầu như không thể sánh với việc sử dụng sản phẩm hoặc codebase có sẵn. Bạn có nên tái phát minh bánh xe không? Lý do và nguyên nhân của việc này?

Có lẽ bạn đã từng xem qua những mô hình chung trong phát triển ứng dụng, hay trong những cuốn sách nói về thiết kế ứng dụng. Những cuốn sách là những kẻ ru ngủ bất kể nội dung của chúng có tuyệt vời thế nào đi chăn nữa. Điều này cũng giống như việc coi một bộ phim về chèo thuyền hoàn toàn khác so với đi thuyền ngoài thực tế. Chính vì thế đừng bao giờ so sánh những đoạn code có sẵn với đoạn code mà chính tay bạn xây dựng nên từ ban đầu, kiểm tra nó, làm hỏng nó, sửa chữa nó, và phát triển nó trên đường phát triển sản phẩm.

Tái phát minh bánh xe không chỉ là bài tập giúp bạn nâng cao khả năng xây dựng cấu trúc của code mà chúng còn giúp cho bạn có được những kiến thức quý giá về cách hoạt động của nhiều thành phần khác nhau đã tồn tại. Bạn có biết làm cách nào để bộ quản lý bộ nhớ hoạt động? phân trang ảo? Bạn có thể tự mình cài đặt chúng không? Hay là danh sách liên kết đôi? Những đối tượng mảng động? ODCD client? Bạn có thể viết một giao diện đồ hoạ hoạt động như cái mà bạn đang sử dụng? Bạn có thể tự

mình tạo ra những tiện ứng cho trình duyệt web của bạn? Bạn có biết khi nào nên viết hệ thống multiplex hay đa luồng? Làm sao để quyết định database sử dụng file hay bộ nhớ? Hầu hết các lập trình viên chưa bao giờ tự tạo ra các thiết lập cốt lõi của phần mềm này do đó họ không có một lượng kiến thức nhất định về cách thức hoạt động của chúng. Hậu quả dẫn đến những phần mềm được tạo ra theo cách này trông giống như những hộp đen bí ẩn đang hoạt động. Chỉ hiểu phần nổi của tảng băng trôi là không đủ để biết được những gì thực sự nguy hiểm bên dưới nó. Không thực sự thấu hiểu cặn kẽ vấn đề trong phát triển phần mềm sẽ giới hạn khả năng tạo ra những sản phẩm tuyệt vời nhất.

Tái phát minh bánh xe và đối với những sai lầm giá trị hơn việc sử dụng cái có sẵn ngay từ đầu. Chúng sẽ cho bạn những bài học từ những lần thử nghiệm, lỗi, và những cảm xúc nhất định đối với thành phần đó, đây chính là điều mà đọc một quyển sách không thể cho bạn được.
Những sự thật và kiến thức từ trong sách thật sự rất quan trọng, nhưng để trở thành một lập trình viên vĩ đại yêu cầu thật nhiều kinh nghiệm như việc sưu tầm những sự thật. Tái phát minh bánh xe đóng vai trò vô cùng quan trọng trong quá trình học tập của lập trình viên như việc cử tạ đối với vận động viên thể hình.

Phần 73: Chống lại sự cám dỗ của Singleton Pattern(*)

Singleton Pattern giải quyết được rất nhiều vấn đề. Bạn biết bạn chỉ cần một instance(**) duy nhất. Bạn có thể đảm bảo rằng instance này được khởi tạo trước khi nó được sử dụng. Nó giữ cho thiết kế của bạn đơn giản bằng một điểm truy cập toàn cầu. Tất cả đều tốt. Vậy điều gì khiến thiết kế pattern cổ điển này trở nên không được yêu thích?

Nhiều trường hợp, một kết quả bất ngờ sẽ đến. Nó có thể rất hấp dẫn, nhưng kinh nghiệm chỉ ra rằng, hầu hết các Singleton Pattern này hại nhiều hơn lợi.Chúng cản trở

khả năng kiểm tra và bảo trì. Thật không may, sự bổ sung khôn ngoan không được sử dụng phổ biến và các lập trình viên không thể bỏ qua việc sử dụng các Singleton.

Nhưng điều này rất đáng để dừng lại:

Các yêu cầu single-instance thường được tính đến. Trong nhiều trường hợp, các suy đoán đơn giản rằng những instance sẽ cần thêm vào trong tương lai. Nêu ra các tính năng trên lý thuyết về thiết kế của một ứng dụng sẽ ảnh hưởng tiêu cực tại một số vị trí. Yêu cầu sẽ cần thay đổi. Thiết kế tốt bao gồm điều này nhưng các Singleton thì không.

Singleton cũng âm thầm gây ra sự phụ thuộc giữa các đơn vị code riêng lẻ. Đây là vấn đề cả bởi vì chúng bị ẩn đi và vì chúng liên kết các các đơn vị không cần thiết. Code của bạn sẽ trở nên “khó nuốt” khi bạn cố gắng viết cái đơn vị kiểm tra, điều này phụ thuộc vào các kết nối lỏng lẻo và khả năng thay thế có chọn lọc các đối tượng mô phỏng trên môi trường thực tế.

Singleton ngăn chặn các đối tượng mô phỏng này.

Singleton cũng kéo dài ngầm trong hệ thống, điều này một lần nữa cản trở các đơn vị kiểm tra, thứ phụ thuộc vào thành phần độc lập, do đó việc kiểm tra có thể được thực hiện theo bất kỳ thứ tự nào và chương trình có thể được đặt vào trạng thái đã biết trước khi thực hiện kiểm thử. Một khi bạn thêm Singleton với trạng thái có thể biến đổi, điều này khó có thể thực hiện. Ngoài ra, trạng thái lâu dài của khả năng truy cập toàn cầu làm cho việc suy luận về code trở nên khó khăn hơn đặc biệt là trong môi trường đa luồng.

Môi trường đa luồng đưa ra những sự “cám dỗ” với Singleton Pattern. Các lớp bảo mật đơn giản khi truy cập không thực sự hiệu quả, do đó bảo mật hai lớp trở (DCLP) nên phổ biến. Thật không may, đây có thể là một hình thức gây thiệt hại lớn. Nó chỉ ra rằng, trong nhiều ngôn ngữ, DCLP không an toàn, ngay cả khi được thiết lập thì vẫn có khả năng vượt qua.

Việc dọn dẹp Singleton cũng có thêm một thách thức nữa:

Việc loại bỏ các Singleton không được hỗ trợ, đây có thể là một vấn đề nghiêm trọng trong một vài trường hợp. Ví dụ, trong cấu trúc của một plug-in, plug-in chỉ được tải lên an toàn sau khi tất cả đối tượng đã được loại bỏ sạch sẽ.

Không có thứ tự nào trong việc loại bỏ các Singleton khi thoát khỏi chương trình. Điều này có thể gây rắc rối với các ứng dụng có chứa các Singleton có sự phụ thuộc lẫn nhau. Khi tắt các ứng dụng như vậy, một Singleton có thể truy cập một ứng dụng khác.

Một số thiếu sót có thể được khắc phục bằng cách đưa ra cơ chế bổ sung. Tuy vậy, điều này phải trả giá bằng việc sự phức tạp sẽ tăng cao trong code tránh việc thay đổi cấu trúc.

Do đó, bạn nên hạn chế sử dụng các Singleton Pattern cho các class không được khởi tạo nhiều lần. Không sử dụng việc truy cập toàn cầu của một Singleton từ code tùy ý. Thay vào đó, truy cập trực tiếp vào Singleton phải từ một vị trí xác định, từ đó có thể chuyển giao diện của nó qua một đoạn code khác. Đoạn code khác này không được sử dụng trước đó cho nên nó không phụ thuộc vào Singleton hay class nào tạo ra giao diện. Điều này phá vỡ các phụ thuộc ngăn cản việc kiểm tra và cải thiện khả năng bảo trì. Vì vậy, trong lần tới, khi bạn nghĩ về việc sử dụng hay triển khai một Singleton, hy vọng bạn sẽ tạm dừng và suy nghĩ lại.
Chú thích:
(*) Một design pattern mà đảm bảo rằng một class chỉ có một instance và cung cấp truy cập vào instance đó toàn cầu
(**) Thực thể, giữ địa chỉ bộ nhớ

Phần 74: Con đường cải tiến hiệu năng đầy bom do code bẩn

Thông thường việc cải tiến hiệu năng của hệ thống bao giờ cũng yêu cầu bạn phải thay thế code cũ. Mỗi khi chúng ta muốn code, từng đoạn code mà chúng quá phức tạp hoặc rối rắm sẽ giống như những quả code bom bẩn nằm đó chờ đợi để ngăn cản mọi sự nỗ lực của bạn. Và điều đầu tiên mà bạn gặp đó chính là rắc rối với code bẩn. Nếu mục đích của bạn chính là khiến nó trở nên mượt mà hơn thì chúng ta có thể dễ dàng dự đoán được thời gian kết thúc. Tuy nhiên, khi chúng ta phải đối diện với những đoạn code bẩn không ngoài dự đoán sẽ khiến cho công việc ấy trở nên khó khăn hơn trong việc dự đoán chính xác thời gian kết thúc công việc.

Xét trường hợp bạn tìm thấy một thực thi nóng. Hành động bình thường chính là làm giảm sức mạnh của thuật toán của nó. Giả sử bạn bảo với quản lý của bạn là bạn cần khoảng 3 – 4 giờ để làm điều đó. Và khi bạn thực hiện điều đó, bạn nhanh chóng nhận ra bạn đã làm hỏng phần bị phụ thuộc. Và những thứ liên quan với nhau được phải được gắn kết với nhau, sự hỏng hóc này là điều có thể dự đoán và lo liệu được. Nhưng chuyện gì sẽ xảy ra nếu như sửa chữa kết quả phụ thuộc ấy trong những thành phần bị phụ thuộc khác đang bị hỏng? Hơn nữa, vấn đề trên cần phải khắc phục từ nguồn, bạn càng không tìm được nguồn gốc của nó thì sự dự đoán của bạn ngày càng trở nên không chính xác. Và rồi tất cả dự đoán của bạn từ 3-4 giờ bỗng tăng lên thành 3-4 tuần. Thông thường sự tăng trưởng không dự đoán trước về thời gian trong kế hoạch của bạn là 2 – 3 ngày tại một thời điểm. Thật dễ dàng để thấy sự sửa chữa “nhanh” này đột nhiên tốn đến vài tháng để hoàn thành. Những trường hợp như thế này gây thiệt hại nặng nề đến sự tín nhiệm cũng như là thể diện chung của team chịu trách nhiệm. Phải chi chúng ta có một công cụ giúp chúng ta xác định và đo lường khả năng này.

Thật ra chúng ta có nhiều cách để xác định, điều khiển mức độ cũng như chiều sâu của các liên kết và độ phức tạp code. Software metrics có thể được dùng để đến sự xuất hiện của các tính năng đặc biệt. Giá trị của các phép đo này tương tự với chất lượng

của code. Hai giá trị trong metrics đếm sự liên kết là fan-in và fan-out. Xét giá trị fan-out cho các class: nó được định nghĩa là số lượng các class là tham chiếu trực tiếp hoặc gián tiếp từ một class mà chúng ta xét. Bạn có thể xem nó như số lượng của tất cả các class cần phải được compile trước khi class bạn đang xét compile. Fan-in, mặt khác, là số lượng các lớp phụ thuộc vào lớp đang xét. Dựa vào giá trị fan-in, fan-out ta có thể tính toán được nhân tố bất ổn bằng công thức: $I= f_o/(f_i+f_o)$. Khi mà I tiến đến 0 thì gói chức năng ta đang xét càng ổn định và càng bất ổn khi I tiến đến 1. Theo dữ kiện điều tra, các gói tính năng ổn định hầu như không chứa code bẩn và ngược lại. Mục đích trong việc sửa chữa chúng chính là khiến cho trị số I càng gần 0 càng tốt.

Khi sử dụng metrics chúng ta phải luôn nhớ rằng chúng chỉ là những quy tắc của ngón tay. Về mặt thuần toán học chúng ta có thể thấy gia tăng f_i mà không thay đổi f_o sẽ khiến cho I gần 0 hơn. Tuy nhiên, mặt trái của việc giá trị fan-in quá lớn sẽ khiến cho những lớp đó trở nên khó để thay đổi mà không làm hỏng sự phụ thuộc của chúng. Đồng thời việc không tác động đến fan-out, bạn thật sự không làm giảm nguy cơ ấy cho nên hay cố gắng cân bằng chúng.

Một mặt tối của việc sử dụng metrics chính là một mảng số khổng lồ do công cụ metrics sinh ra có thể là trở ngại đối với người mới. Họ nói, software metrics có thể là một công cụ mạnh mẽ trong công cuộc đấu tranh vì code sạch(clean code). Chúng có thể giúp ta xác định mà dự đoán bom code bẩn trước khi chúng trở thành mối nguy hại nghiêm trọng trong việc luyện tập hiệu chỉnh hiệu suất.

Phần 75: Điều đơn giản đến từ sự tối giản

"Làm lại lần nữa..." Ông chủ của tôi vừa nói vừa ấn mạnh phím xóa. Tôi nhìn màn hình máy tính với một cảm giác quá quen thuộc, khi đoạn code của tôi biến mất từng dòng một.

Ông chủ của tôi, Stefan, không phải lúc nào cũng là người có tiếng nói nhất, nhưng ông ấy biết code này ổn hay không chỉ qua quan sát. Và ông ấy biết chính xác phải làm gì với nó.

Tôi đến với vị trí hiện tại là một lập trình viên sinh viên tràn đầy năng lượng, nhiệt huyết nhưng hoàn toàn không biết viết code. Tôi có suy nghĩ kì quái rằng giải pháp cho mọi vấn đề là thêm vào một chút gì đó. Hoặc bổ sung thêm một dòng code nữa. Vào một ngày tồi tệ, thay vì trở nên tốt hơn sau mỗi lần sửa đổi, code của tôi trở nên phức tạp hơn và ngày càng hoạt động không ổn định.

Đó là điều hoàn toàn dễ hiểu, nhất khi bạn đương vội và chỉ muốn thực hiện những thay đổi tối thiểu nhất đối với đoạn code hiện có, ngay cả khi điều đó không ổn cho lắm. Hầu hết các lập trình viên sẽ giữ lại đoạn code lỗi, vì e sợ việc viết một đoạn code mới sẽ đòi hỏi nhiều nỗ lực hơn so với việc quay lại đoạn đầu. Điều đó có thể đúng với code sắp đi vào hoạt động, nhưng chỉ có một số code thực sự thành công.

Nhiều thời gian sẽ bị lãng phí vào việc cố gắng cứu vãn code hỏng hơn cần thiết. Khi cái gì đó dần mất đi giá trị, nó cần nhanh chóng bị loại bỏ.

Không phải ai cũng dễ dàng loại bỏ tất cả các kiểu gõ, đặt tên hay định dạng. Phản ứng của sếp tôi có hơi quá khích, nhưng nó buộc tôi phải suy nghĩ lại trong lần thử thứ hai (hoặc đôi khi là thứ ba). Tuy nhiên, cách tốt nhất để sửa code lỗi là chuyển sang chế độ code được tái cấu trúc, dịch chuyển hoặc xoá bỏ.

Code cần phải đơn giản. Cần có một số lượng tối thiểu các biến, hàm, khai báo và các yêu cầu cú pháp khác. Thêm dòng, biến phụ… hay bất cứ điều gì, nên bị loại trừ ngay lập tức. Chỉ nên có vừa đủ để hoàn thành công việc, hoàn thành thuật toán hay thực hiện các tính toán. Bất cứ điều gì khác chỉ là thứ không cần thiết, vô tình xuất hiện và che khuất những thứ quan trọng.

Tất nhiên, nếu không thành công thì bạn chỉ cần xóa đi và làm lại. Suy nghĩ theo cách đó thường giúp bỏ qua nhiều sự lộn xộn không cần thiết.

Phần 76: The Single Responsibility Principle - SRP

Một trong những nguyên tắc nền tảng của một thiết kế tốt là:

“Tập hợp tất cả những điều mà thay đổi bởi một lý do, và tách chúng thành những điều mà bị thay đổi bởi những lý do khác nhau.”

Nguyên tắc trên thường được biết đến với cái tên Nguyên tắc trách nhiệm duy nhất hay SRP (Single Responsibility Principle). Một cách dễ hiểu chính là một hệ thống con, module, class, hay kể cả function không nên có nhiều hơn một lý do để thay đổi. Một ví dụ điển chính là một class gồm có những phương thức làm việc với những nguyên tắc kinh doanh, báo cáo, và database:

```
public class Employee {
        public Money calculatePay()…
        public String reportHours()…
        public void save()…
}
```

Một vài lập trình viên nghĩa rằng việc kết hợp ba hàm này lại với nhau vào chung một class là một điều vô cùng đúng đắn. Suy cho cùng, Class là nơi tập hợp các hàm tính toán trên những biến chung. Tuy nhiên, vấn đề chính là ba hàm này thay đổi hoàn toàn dựa trên những lý do khác nhau. Hàm calculatePay sẽ thay đổi bất cứ khi nào nguyên tắc kinh doanh của việc tính toán chi phí tăng. Hàm reportHours sẽ thay đổi bất cứ khi nào có người nào đó muốn dùng một định dạng báo cáo khác. Tuy nhiên, nếu lớp Employee bị lạm dụng bởi các bộ phận khác dẫn đến mọi sự thay đổi của Employee sẽ khiến cho những bộ phận khác phải triển khai lại và phủ định toàn bộ những lợi ích của mô hình thiết kế các thành phần(hoặc là SOA nếu bạn thích một cái tên hoa mỹ hơn).

```
public class Employee {
```

```
        public Money calculatePay()…
}
public class EmployeeReporter{
        public string reportHours(Employee e)…
}
public class EmployeeRepository{
        public void save(Employee e)…
}
```

Bằng một sự tách rời đơn giản của class ban đầu ở bảng trên đã giải quyết tất cả các vấn đề mà ta gặp phải. Mỗi class này có thể được đặt trong từng phần riêng của nó. Hoặc là, mọi class phụ trách công việc báo cáo có thể được xếp vào bộ phận báo cáo. Tất cả mọi class liên quan đến việc trao đổi thông tin từ database có thể được xếp vào bộ phận lưu trữ. Và tất cả những class liên quan đến các quy luật kinh doanh thì có thể xếp vào bộ phận kinh doanh.

Nếu bạn nhanh trí bạn có thể nhận ra code ở trên vẫn còn sự phụ thuộc vào nhau. Chính là kiểu Employee vẫn còn bị phụ thuộc vào những lớp khác. Vì vậy nếu Employee bị chỉnh sửa, hầu hết các những class khác đều phải compile lại hoặc tái sản xuất. Chính vì thế, kiểu Employee không thể bị thay đổi và phải được cài đặt một cách độc lập. Tuy nhiên những lớp khác đều có thể làm việc với nó được đồng thời phải được cài đặt độc lập. Không một sự thay đổi nào của một trong những lớp đó có thể khiến cho các lớp khác phải compile lại hoặc tái sản xuất. Kể cả kiểu Employee cũng có thể được cài đặt độc lập bằng cách sử dụng Đảo ngược phần tử độc lập (Dependency Inversion Principle -DIP), nhưng đó là một chủ đề khác cho một cuốn sách khác.

Hãy cẩn thận áp dụng SRP, phân biệt những điều mà bị thay đổi bởi các lý do khác nhau, là một trong những chiếc chìa khoá để tạo ra các thiết kế có cấu trúc các thành phần được triển khai một cách độc lập.

Phần 77: Bắt đầu từ “CÓ”

Có một lần, tôi đến một cửa hàng tạp hoá để tìm "edamame" (mà tôi chỉ biết mơ hồ là một loại rau). Tôi không chắc liệu thứ tôi cần tìm nằm ở khu rau củ, khu đông lạnh hay khu đồ hộp. Tôi đã từ bỏ và tìm một nhân viên để nhờ trợ giúp. Nhưng cô ấy cũng không biết!

Cô nhân viên đó đã có thể trả lời theo nhiều cách khác nhau. Cô ấy có thể khiến tôi cảm thấy mình thật ngốc nghếch khi không biết tìm ở đâu, hoặc cho tôi những gợi ý mơ hồ, hay đơn giản nói với tôi rằng họ không có món đó. Nhưng thay vào đó, cô coi yêu cầu này như một cơ hội để tìm ra giải pháp và giúp đỡ khách hàng. Cô ấy gọi cho các nhân viên khác và chỉ trong vài phút tôi đã tìm được chính xác món đồ mình cần ở khu đông lạnh.

Cô nhân viên trong trường hợp này đã xem xét một yêu cầu và bắt đầu từ tiền đề rằng chúng tôi sẽ giải quyết và đáp ứng yêu cầu đó. Cô ấy bắt đầu từ có thay vì bắt đầu từ không.

Khi tôi lần đầu tiên đảm nhiệm vai trò lãnh đạo kỹ thuật, tôi cho rằng công việc của tôi là bảo vệ phần mềm của tôi khỏi những yêu cầu lố bịch của các nhà quản lý sản phẩm và phân tích kinh doanh. Tôi coi yêu cầu là một điều gì đó để bác bỏ, không phải để thực hiện.

Rồi đến một lúc, tôi bắt đầu cho rằng có lẽ có một cách khác để làm việc liên quan đến việc thay đổi quan điểm của tôi từ bắt đầu từ không sang bắt đầu từ có. Trên thực tế, tôi đã tin rằng bắt đầu từ có thực sự là một phần thiết yếu của việc trở thành một nhà lãnh đạo kỹ thuật.

Biến đổi đơn giản này đã thay đổi hoàn toàn cách tôi tiếp cận công việc của mình. Hóa ra, có rất nhiều cách để nói có. Khi ai đó nói với bạn "Hey, ứng dụng này sẽ tác động mạnh mẽ hơn đến người dùng nếu chúng ta xây dựng một giao diện thuận mắt và có

sự phối hợp màu sắc" bạn có thể từ chối vì nó nghe có vẻ vô lý. Nhưng tốt hơn nên bắt đầu với câu hỏi "Tại sao?". Thường sẽ có một số lý do thực tế và hấp dẫn cho việc người đó yêu cầu tính thẩm mỹ nên được ưu tiên. Ví dụ, bạn có thể sắp hợp tác với một khách hàng lớn với tiêu chuẩn bắt buộc tính thẩm mỹ cao.

Thông thường bạn sẽ thấy rằng khi bạn biết ngữ cảnh của yêu cầu, các khả năng mới sẽ xuất hiện. Nói cách khác, yêu cầu phải được thực hiện với sản phẩm hiện có theo cách cho phép bạn nói "có" mà không cần làm gì cả: "Trên thực tế, trong tùy chọn người dùng, bạn có thể tải xuống giao diện theo ý thích và bật nó lên."

Đôi khi chỉ đơn giản là ai đó có một ý tưởng trái ngược với quan điểm của bạn. Tôi thấy khá hữu ích khi áp dụng câu hỏi "Tại sao?" cho chính bản thân. Đôi khi đưa ra lý do sẽ cho thấy rõ rằng phản ứng ban đầu của bạn không có nghĩa lý gì cả. Hãy nhớ rằng, mục tiêu của tất cả những việc này là đồng ý với người khác và cố gắng dàn xếp ổn thoả, không chỉ cho người khác mà còn cho bản thân và những cộng sự của bạn.

Nếu bạn có thể đưa ra lời giải thích thuyết phục cho lý do tại sao yêu cầu đó không tương thích với sản phẩm hiện có, thì có khả năng bạn sẽ có một cuộc thảo luận liệu sản phẩm bạn đang xây dựng có phù hợp hay không. Cho dù cuộc thảo luận đó kết thúc như thế nào thì mọi người đều xác định rõ được rằng sản phẩm này sẽ là gì và không là gì.

Bắt đầu từ có nghĩa là phối hợp với đồng nghiệp của bạn, không phải là chống lại họ.

Phần 78: Lùi lại và để tự động hóa làm việc

Tôi đã làm việc với các lập trình viên mà khi được yêu cầu tạo một số dòng code trong một module đã dán các tệp vào một trình xử lý văn bản và sử dụng tính năng “đếm

dòng” của nó. Và lần sau, lần sau nữa, họ lặp lại những hành động đó. Cách làm này quả thực không được khôn ngoan cho lắm.

Tôi đã làm việc với một dự án có quy trình triển khai phức tạp, liên quan đến code signing và chuyển kết quả đến một máy chủ, đòi hỏi nhiều thao tác khác nhau. Ai đó đã tự động hóa nó và chương trình đã chạy cực kỳ trơn tru trong lần thử nghiệm cuối cùng, thậm chí còn tốt hơn nhiều so với những gì chúng tôi mong đợi. Hiển nhiên điều đó thật tuyệt.

Vì vậy, tại sao mọi người lại làm cùng một nhiệm vụ nhiều lần thay vì dành thời gian để tự động hóa nó một lần và mãi mãi?

Quan điểm sai lầm phổ biến 1: Tự động hóa chỉ dùng để thử nghiệm

Đương nhiên tự động hóa thử nghiệm là điều tuyệt vời, nhưng tại sao lại chỉ dừng ở đó? Các tác vụ lặp đi lặp lại xuất hiện rất nhiều trong bất kỳ dự án nào: kiểm soát phiên bản, biên dịch, xây dựng các tệp JAR, tạo tài liệu, triển khai và báo cáo… Đối với nhiều tác vụ trong số này, tập lệnh mạnh hơn chuột. Việc thực hiện những nhiệm vụ vốn tẻ nhạt đã trở nên nhanh chóng và đáng tin cậy hơn.

Quan điểm sai lầm phổ biến 2: Đã có IDE thì không cần tự động hóa

Đã bao giờ bạn có một cuộc tranh luận kiểu Nhưng- nó- kiểm tra/ xây dựng/ vượt qua- các- test- trên- máy- của- tôi với đồng nghiệp? Các IDE hiện đại có hàng nghìn cài đặt khác nhau và về cơ bản thì việc đảm bảo tất cả thành viên trong nhóm có cấu hình giống hệt nhau là điều không thể. Xây dựng các hệ thống tự động hóa như Ant hay Autotools cho phép bạn kiểm soát và lặp lại nhiệm vụ khi cần.

Quan điểm sai lầm phổ biến 3: Cần học các công cụ kỳ lạ để tự động hóa

Bạn có thể tiến rất xa với một shell language tốt (như bash hay PowerShell) và một hệ thống tự động hóa. Nếu bạn cần tương tác với các trang web, hãy sử dụng công cụ như iMacros hay Selenium.

Quan điểm sai lầm phổ biến 4: Không thể xử lý các định dạng tệp nên không thể tự động hóa tác vụ

Nếu một phần trong quy trình yêu cầu tài liệu Word, bảng tính hay hình ảnh, thì thực sự khá khó khăn để tự động hóa nó. Nhưng liệu điều đó có thực sự cần thiết? Bạn có thể sử dụng văn bản đơn giản? Giá trị được phân tách bằng dấu phẩy? XML? Một công cụ tạo bản vẽ từ một tệp văn bản? Một tinh chỉnh nhỏ trong tiến trình có thể mang lại kết quả tốt và giảm thiểu đáng kể sự tẻ nhạt.

Quan điểm sai lầm phổ biến 5: Không có thời gian để tìm hiểu

Bạn không cần phải học tất cả bash hay Ant để bắt đầu. Hãy học dần dần. Khi bạn có một nhiệm vụ mà bạn nghĩ có thể và nên được tự động hóa, hãy tìm hiểu vừa đủ về các công cụ mà bạn cần để thực hiện điều đó. Và hãy làm điều đó sớm ngay khi thời gian cho phép. Một khi bạn thành công, bạn (và sếp của bạn) sẽ nhận thấy việc đầu tư vào tự động hóa là hoàn toàn hợp lý.

Phần 79: Thông thạo dụng cụ analysis code

Giá trị của testing đã bước vào đời của các nhà phát triển từ những giai đoạn đầu tiên. Trong một vài năm gần đây sự phát triển mạnh mẽ của unit testing, test-driven và những phương thức nhanh chóng đã cho thấy sự quan tâm đáng kinh ngạc đến việc kiểm tra qua tất cả các giai đoạn của chu trình phát triển sản phẩm. Tuy nhiên, testing là một trong những công cụ để bạn có thể nâng cao chất lượng code.

Trở lại dòng chảy của thời gian, khi mà C vẫn còn là một hiện tượng mới, thời gian mà CPU và bất kì phương tiện lưu trữ nào hoạt động cũng rất chậm. Phiên bản compiler đầu tiên của ngôn ngữ C đã không quan tâm đến vấn đề này và cắt giảm bớt số lượng code đi qua bằng cách loại bỏ một số analyses. Điều này có nghĩa là compiler chỉ kiểm

tra một phần nhỏ của đoạn bug có thể được phát hiện trong thời gian biên dịch. Để khắc phục điều này, Stephen Johnson đã tạo ra một ứng dụng gọi là lint – cái mà xoá bỏ fluff khỏi code của bạn – nó hoạt động bằng cách cài đặt tĩnh một số analyses mà C compiler đã xoá bỏ. Công cụ static analysis, tuy nhiên, đã nhận được nhiều tiếng tăm vì những cảnh báo giả và những cảnh báo về những quy ước mà không phải lúc nào cũng nhất thiết để làm theo.

Mặt bằng chung hiện nay của các ngôn ngữ, compilers, và công cụ analysis phức tạp. Thời gian đọc ghi của CPU và bộ cũng trở nên nhanh hơn, chính vì thế compiler có thể cố gắng kiểm tra thêm một số lỗi. Hầu hết các ngôn ngữ đều tự hào khi có ít nhất một công cụ mà có thể kiểm tra các lỗi do sai cú pháp, lỗi gotchas, và đôi khi khéo léo chặn các lỗi thường rất khó để bắt gặp, như là pointer null dereferences. Những công cụ tinh vi hơn, như là Splint của C hoặc là Pylint của Python, có thể được tuỳ chỉnh, điều này giúp bạn có thể chọn những lỗi hoặc cảnh báo nào mà Compiler phát hiện bằng một file thiết lập, hay bằng dòng lệnh, hoặc điều chỉnh trong IDE của bạn. Splint còn có thể để bạn ghi chú code của bạn bằng comment giúp bạn có gợi ý về cách hoạt động của chương trình.

Nếu tất cả những điều trên sai, bạn nên tìm những lỗi đơn giản hay những xung đột các chuẩn mực mà chúng không thể bị phát hiện bởi compiler, IDE, hoặc công cụ lints, vậy thì bạn phải tự kiểm tra lấy thôi. Việc này không quá khó như bạn nghĩ đâu. Hầu hết các ngôn ngữ, đặc biệt là những ngôn ngữ mang danh dynamic, thường sở hữu hệ thống ngôn ngữ và công cụ của compiler như một phần của thư viện tiêu chuẩn. Đó là một điều đáng giá khi bạn có thể biết những góc khuất của bộ thư viện tiêu chuẩn được dùng bởi các nhóm nhà phát triển của ngôn ngữ mà bạn đang dùng, thường thì những điều này chứa những “viên ngọc quý” cho static analysis và dynamic testing. Ví dụ, thư viện chuẩn của Python có một disassembler cái mà sẽ cho bạn biết bytecode dùng để compile code hoặc đối tượng code. Chức năng này dường bị quên lãng bởi những nhà thiết kế compiler trong đội ngũ phát triển Python, tuy nhiên đây lại là một chức năng cực kỳ hữu ích không thể thiếu trong những tình huống hằng ngày. Một điều nữa mà thư viện này có thể tách rời là sự truy stack của bạn, giúp bạn theo dõi một cách chính xác bytecode nào là nguyên nhân của các ngoại lệ cuối cùng không bị phát hiện.

Chính vì vậy đừng để testing là việc cuối cùng của việc bảo quản chất lượng của sản phẩm của bạn – hãy thông thạo các công cụ analysis và đừng sợ khi thực hiện điều đó một mình.

Phần 80: Test for Required Behavior, not Incidental Behavior

Một điều đáng tiết thường xảy ra trong việc kiểm thử chính là dự đoán tất cả những gì mà mà một chương trình làm một cách chính xác với bộ thử của bạn. Nhìn sơ qua thì nó chả giống như một điều gì đáng tiếc cả. Tuy nhiên, ở một góc độ khác, vấn đề trở nên rõ ràng hơn: một điều đáng tiết trong testing thường xảy ra chính là kiểm tra một cách cứng nhắc đến một điều đặc biệt của chương trình mà chúng lại không quan trọng cũng như không thể hiện được tính năng mong muốn của chương trình.

Khi bài test được viết tràn lan, tự động, những thay đổi tác động đến chúng để thích hợp với những trường hợp chính khiến chúng không chính xác hoặc phát sinh những cảnh báo giả. Để đáp lại chúng, lập trình viên sẽ sửa lại code hoặc bộ test ấy. Tuy nhiên nếu như báo động ấy là thật thì nó chính là hậu quả của nỗi sợ hãi, không chắc chắn, hoặc nghi ngờ. Điều này khiến cho những trường hợp phát sinh xảy ra nhiều hơn. Khi viết lại một test mới, lập trình viên chúng ta hoặc tập trung vào những trường hợp chính, thông dụng(tốt) hoặc đơn giản ràng buộc nó bằng cài đặt mới(không tốt). Những bộ test không chỉ phải đúng mà còn phải thật chính xác.

Lấy ví dụ, trong một sự so sánh có 3 kết quả, như là hàm strcmp của ngôn ngữ C hay String.compareTo của ngôn ngữ Java. Những yêu cầu của kết quả chính là trả về -1 nếu bên trái lớn hơn bên phải, 1 nếu như bên phải lớn hơn bên trái và 0 nếu 2 bên bằng nhau. Kiểu so sánh này được dùng trong nhiều API, kể cả phép so sánh trong qsort của C và compareTo của Java. Mặc dù giá trị -1 và 1 thường dùng để phân biệt nhỏ hơn hay lớn hơn, nhưng các lập trình viên thường nhầm lẫn cho rằng những giá trị

này là mới chính là yêu cầu của bài toán từ đó viết những bộ test chung dựa trên lầm tưởng ấy.

Một vấn đề tương tự trong việc test đó chính là những đòi hỏi về khoảng cách, từ ngữ, và những khía cạnh khác của định dạng và biểu diễn văn bản là không là không cần thiết. Trừ khi bạn đang viết, ví dụ, một chương trình sinh file XML mà yêu cầu chỉnh sửa định dạng và khoảng cách là không đáng kể. Tương tự như thế, đặt nút nhấn và nhãn tuân thủ những quy tắc UI có thể giảm khả năng để thay đổi và tái thiết lập chúng trong tương lai. Những thay đổi nhỏ trong quá trình cài đặt hoặc những thay đổi ngẫu nhiên do phát sinh đột nhiên trở thành kẻ phá hoại.

Những bài test được tinh chỉnh quá mức sẽ xảy ra vấn đề hộp trắng(White Box) khi tiến tới unit testing. Những bài test hộp trắng sử dụng cấu trúc của đoạn code để quyết định những trường hợp nào dùng để test. Sự thất bại điển hình nhất chính là khiến cho đoạn code thực hiện việc nó thực hiện. Điều này đơn giản tái khởi động tất cả mọi thứ, đoạn công không tạo giá trị nào và dẫn đến những sai lầm trong quá trình phát triển và bảo mật.

Cuối cùng, chúng ta không thể đạt được sự hiệu quả nếu cài đặt những bộ test như một con vẹt. Vấn đề phải được nhìn nhận dưới góc nhìn hộp đen(black box) của từng đơn vị test, phác thảo những liên kết bằng hình thức thực thi. Từ đó, điều chỉnh bộ test theo những hành vi được yêu cầu của đoạn code.

Phần 81: Kiểm tra một cách cụ thể và chính xác

Từ lâu việc kiểm tra những hành vi cần thiết, cụ thể của các đoạn code luôn là điều quan trọng, hơn việc kiểm tra những hành vi ngẫu nhiên của chúng bằng một thiết lập cụ thể. Tuy nhiên điều này không nên được thực hiện hoặc nhầm lẫn như một cái cớ cho các kiểm tra mơ hồ. Việc kiểm tra cần phải đúng đắn và chính xác.

Một điều gì đó như là một sự cố gắng, kiểm tra, và kiểm tra cổ điển, thói quen sắp xếp đưa ra các ví dụ minh hoạ. Công việc mỗi ngày của một lập trình viên không nhất thiết là cài đặt thuật toán sắp xếp, tuy nhiên sắp xếp lại chính là một ý tưởng quen thuộc mà nhiều người nghĩ rằng họ biết điều gì đáng để mong đợi. Đây là một điều quen thuộc, tuy nhiên, nó có thể khiến cho việc xem qua một giả định cụ thể gặp khó khăn.

Khi các lập trình viên được hỏi: bạn test đoạn code đó vì điều gì? Cho đến nay câu trả lời xuất hiện thường xuyên nhất chính là “Kết quả của việc sắp xếp chính là một dãy có các phần tử được sắp xếp theo một thứ tự nhất định”. Mặc dù đây là điều chính xác tuy nhiên nó chưa đủ. Khi cài đặt thêm một điều kiện chính xác vào, nhiều lập trình viên thêm vào điều kiện đó là “ dãy kết quả phải có độ dài bằng độ dài dãy ban đầu”.Tuy điều trên là đúng nhưng vẫn chưa đủ. Lấy ví dụ, cho một dãy như sau:

3 1 4 1 5 9

Dãy dưới đây thoả mãn các điều kiện đã được đặt ra khi sắp xếp chúng với thứ tự tăng dần và có cùng độ dài với dãy đã cho:

3 3 3 3 3 3

Tuy nó đạt đủ điều kiện đã được đặt ra nhưng nó đã sai với ý định của chúng ta. Đây là một lỗi dựa trên một trường hợp thực tiễn được trích xuất từ code của một sản phẩm(may mắn được phát hiện trước khi ra mắt), điều mà chỉ với một phím nhầm lẫn hoặc điều mà chúng ta đặt ra không đủ chặt dẫn đến một cơ chế phức tạp cho việc lặp lại phần tử đầu tiên của dãy ban đầu trong kết quả.

Như vậy điều kiện đầy đủ chính là dãy con phải được sắp xếp theo một thứ tự nhất định đồng thời nó là một chỉnh hợp của dữ liệu nguyên bản. Có như thế chúng mới có thể ràng buộc hành vi cần thiết và khiến cho kết quả lúc nào cũng thật chỉnh chu.

Ngay cả thiết lập dữ kiện cho bài test bằng cách mô tả nó cũng không đủ tốt để trở thành một bài test tốt. Một bài test tốt phải đọc được và đủ đơn giản để bạn có thể dễ dàng xem rằng liệu nó có chính xác hay chưa. Trừ khi bạn đã code một đoạn chương trình để kiểm tra rằng kết quả đã được sắp xếp hay chưa và liệu nó có phải là một chỉnh hợp của dữ liệu ban đầu không. Điều này đơn giản có nghĩa là code dùng để test code sẽ phức tạp hơn nhiều so với code được test như Tony Hoare được chiêm ngưỡng:

“ Có 2 cách để xây dựng nên một bản thiết kế phần mềm: một chính là khiến nó trở nên đơn giản đến mức chả còn gì khiến ta phải bận tâm, hai là khiến nó trở nên vô cùng phức tạp đến nỗi hoàn hảo”
Bằng cách áp dụng một vài ví dụ thích hợp chúng ta đã có thể loại bỏ sự phức tạp ngẫu nhiên này, kể cả khả năng xảy ra tai nạn. Ví dụ, cho dãy sau đây:

3 1 4 1 5 9

Kết quả sau khi sắp xếp là:

1 1 3 4 5 9

Nó là duy nhất, không chấp nhận thay thế.

Những ví dụ thích hợp giúp chúng ta minh hoạ chung các hành vi bằng những cách thức dễ tiếp cận và rõ ràng. Kết quả của việc thêm một phần tử vào tập rỗng không phải là phủ định nó. Kết là tập mà bây giờ có một phần tử, chính là phần tử được thêm vào. Thêm vào hai hoặc nhiều hơn các phần tử cũng thoả mãn yêu cầu tuy đúng nhưng chưa đủ. Kể cả thêm một phần tử nhưng có giá trị khác cũng sai nốt. Kết quả của việc thêm một hàng vào bảng không chỉ đơn thuần là bảng đó sẽ lớn thêm một hàng. Nó đồng thời phải đảm bảo giá trị của hàng có thể được sử dụng để tái tạo lại hàng được thêm vào,và tương tự như thế.

Trong việc phân loại hành vi, bài test không chỉ đơn giản là đảm bảo tính đúng đắn mà còn phải đảm bảo sự chính xác.

Part 82: Hãy test khi bạn đang ngủ (và cả cuối tuần)

Hãy thư giãn. Tôi không nói về việc sắp xếp đến trung tâm phát triển ở nước ngoài, làm quá giờ vào cuối tuần hay tăng ca buổi tối. Tôi muốn bạn chú ý đến chúng ta có khả năng tính toán như thế nào theo quan điểm của chúng tôi. Cụ thể là, chúng ta đã không khai thác như thế nào để khiến việc trở thành lập trình viên dễ hơn một chút. Bạn có thường xuyên cảm thấy khó khăn để có đủ khả năng tính toán trong suốt ngày làm việc? Nếu vậy, máy chủ thử nghiệm của bạn đang làm gì ngoài giờ làm việc chính?

Thông thường, các máy chủ thử nghiệm đều rảnh rỗi vào buổi tối và cuối tuần. Bạn có thể sử dụng chúng cho lợi ích của bạn.

- **Bạn có bao giờ cảm thấy tội lỗi khi cam kết một cuộc giao dịch mà không chạy hết các bài kiểm tra?** Một trong các lí do chính mà lập trình viên không chạy hết bộ thử nghiệm trước khi cam kết code là do khoảng thời gian mà họ có thể mất. Khi deadlines tới gần thì con người thường bắt đầu cắt giảm bớt vài thứ. Một cách để giải quyết điều này là chia bộ test thành hai hoặc nhiều profiles. Việc này sẽ giúp đảm bảo các tests được chạy trước khi chúng được xác nhận. Toàn bộ profiles thử nghiệm (bao gồm profile bắt buộc - chỉ để chắc chắn thôi) sẽ được chạy tự động qua đêm và sẵn sàng báo cáo kết quả vào sáng hôm sau.

- **Bạn có bao giờ có cơ hội để kiểm tra tính ổn định của sản phẩm của bạn?** Các tests có thời gian chạy lâu rất quan trọng để xác định memory leaks và các vấn đề ổn định khác. Chúng hiếm khi được chạy trong ngày bởi vì chúng cần thời gian và nguồn tài nguyên. Bạn có thể tự động hóa bộ kiểm tra độ thẩm thấu để chạy chúng suốt đêm, và chạy lâu hơn một chút vào cuối tuần. Từ 6.00 chiều thứ Sáu đến 6.00 sáng thứ Hai ta sẽ có 60 tiếng để kiểm tra tiềm năng của nó.

- **Bạn có nhận được thời gian chất lượng cho Môi trường kiểm nghiệm hiệu năng (Performance testing environment) của bạn?** Tôi đã nhìn thấy vài đội cãi nhau để nhận được thời gian cho performance testing environment. Phần lớn các trường hợp không đội nào có được thời gian chất lượng trong ngày, trong khi môi trường (environment) hầu như không hoạt động sau nhiều giờ. Các máy chủ và mạng lưới cũng không bận rộn suốt đêm hay các ngày nghỉ. Đây là thời gian lí tưởng để chạy một vài performance tests chất lượng.

- **Có nhiều hoán vị để có thể test thủ công?** Trong nhiều trường hợp, sản phẩm của bạn sẽ được chỉ định để chạy trên các nền tảng (platform) đa dạng khác nhau. Ví dụ, cả 32-bit và 64-bit trên Linux, Solaris, và Windows, hay đơn giản là ở các phiên bản khác nhau của cùng một hệ điều hành. Để làm mọi thứ tệ hơn, nhiều ứng dụng hiện đại đã để lộ ra một lượng lớn không cần thiết về cơ chế và

giao thức vận chuyển (HTTP, AMQP, SOAP, CORBA,...). Việc test thủ công toàn bộ các hoán vị này rất tốn thời gian và rất có thể nó được hoàn thành như một sự giải thoát do áp lực tài nguyên. Than ôi, điều đó có thể là quá chậm trong tuần hoàn tìm một số lỗi khó chịu.

Bộ thử nghiệm tự động chạy suốt đêm hoặc suốt cuối tuần sẽ đảm bảo các hoán vị đó được kiểm tra thường xuyên hơn. Với một chút tư duy và kiến thức cơ bản, bạn có thể sắp xếp một số cron jobs để bắt đầu vài bài kiểm tra vào buổi tối và xuyên suốt ngày nghỉ cuối tuần. Ngoài ra còn có các phần mềm kiểm thử (testing tool) có thể giúp ích cho bạn. Một vài tổ chức còn có những lưới máy chủ chứa máy chủ đến từ các ban và đội khác nhau nhằm đảm bảo nguồn tài nguyên được tận dụng hiệu quả. Nếu tổ chức của bạn đang thực hiện điều này, bạn có thể nộp những bài tests để chạy vào buổi tối và cuối tuần.

Phần 83: Kiểm thử là một quá trình nghiêm ngặt trong phát triển phần mềm

Các nhà phát triển thích sử dụng các ẩn ý bắt buộc khi cố gắng giải thích những gì họ làm với thành viên gia đình, vợ hoặc chồng và những người không có chuyên môn. Chúng tôi thường xuyên sử dụng để xây dựng các cầu nối và các khuôn khổ kỹ thuật cứng nhắc khác. Tuy nhiên, tất cả những ẩn ý này sẽ bị sai lệch nhanh chóng, khi bạn bắt đầu cố gắng sử dụng nó quá nhiều. Nó chỉ ra rằng, phát triển phần mềm không giống như những ngành kỹ thuật “cứng nhắc” theo nhiều khía cạnh quan trọng.

So với các ngành kỹ thuật “cứng”, việc phát triển phần mềm được so sánh với những người xây dựng một cây cầu khi mục tiêu đặt ra là hoàn thiện cây cầu và có thể chịu được sức nặng. Nếu nó giữ nguyên, đó sẽ là một cây cầu tốt. Nếu không, thời gian sẽ đưa tất cả trở về bản vẽ. Trong vài nghìn năm qua, các kỹ sư đã nghiên cứu và tìm ra các nguyên lý toán học và vật lý mà họ có thể sử dụng cho giải pháp về cấu trúc mà không cần phải thử xem điều gì sẽ xảy ra.

Chúng ta không có bất kỳ thứ gì như vậy trong việc phát triển phần mềm và có lẽ sẽ không bao gì vì phần mềm trong thực tế là rất khác nhau. Đối với một nghiên cứu sâu về sự khác biệt giữa kỹ thuật phần mềm là các ngành kỹ thuật thông thường, bài viết "Thiết kế phần mềm là gì?", được viết bởi Jack Reeves trong tạp chí C++ năm 1992, một tác phẩm kinh điển. Mặc dù nó đã được viết cách đây hơn hai thập kỷ, nhưng nó vẫn rất chính xác. Ông đã "vẽ" một bức tranh ảm đạm trong so sánh này, nhưng điều còn thiếu năm 1992 là một thử nghiệm đầy đủ.

Thử nghiệm những thứ "cứng nhắc" là khó khăn vì bạn phải xây dựng và thử nghiệm, điều này không khuyến khích xây dựng các quá trình trên lý thuyết chỉ để xem điều gì sẽ xảy ra. Nhưng xây dựng quá trình này trong phần mềm rất rẻ. Chúng tôi đã phát triển toàn bộ hệ sinh thái của các công cụ giúp dễ dàng thực hiện điều đó: đơn vị thử nghiệm, đối tượng giả lập, việc khai thác thử nghiệm và nhiều công cụ khác. Các kỹ sư rất thích xây dựng một các gì đó rồi chạy thử trong điều kiện thực tế. Là một nhà phát triển phần mềm, chúng ta nên chấp nhận thử nghiệm như là cơ chế xác minh chính (nhưng không phải là duy nhất) cho phần mềm. Thay vì chờ đợi một số tính toán phần mềm, chúng tôi đã có sẵn các công cụ để đảm bảo thực hiện các kỹ thuật tốt. Nhìn dưới góc độ này, chúng ta hiện có đủ "đạn dược" chống lại các nhà quản lý cho rằng: "Chúng tôi không có thời gian để kiểm tra". Một người xây dựng cây cầu sẽ không bao giờ nghe thấy từ ông chủ của họ: "Đừng bận tâm phân tích cấu trúc căn nhà đó – chúng tôi có thời hạn cố định." Sự thừa nhận rằng thử nghiệm thực sự là con đường dẫn đến tái tạo và củng cố chất lượng trong phần mềm cho phép chúng tôi – nhà phát triển đẩy lùi các lập luận chống lại nó là sự thiếu chuyên nghiệp.

Kiểm tra sẽ mất một thời gian, giống như phân tích cấu trúc. Cả hai hoạt động đều sẽ đảm bảo cho chất lượng của sản phẩm cuối cùng. Đã đến lúc các nhà phát triển phần mềm phải chịu trách nhiệm về những gì họ làm ra. Từng cá nhân kiểm tra là không đủ nhưng nó cần thiết. Kiểm tra là kỹ thuật nghiêm ngặt trong phát triển phần mềm.

Phần 84: Suy nghĩ trong từng States

Mọi người thường có một mối quan hệ kì lạ với các states(1). Sáng nay tôi đã thưởng chút cafe tại một cửa hàng gần nhà để chuẩn bị cho công việc. Thức uống yêu thích nhất của tôi là latte, nhưng nhân viên bán hàng lại không cho vào ly bất kì giọt sữa nào, và họ đã nói với tôi thế này:

“Xin lỗi quý khách, chúng tôi không còn một tí tì ti sữa nào”

Đối với một lập trình viên, đó là một câu nói kì quặc. Cho dù bạn có hết sữa hay là không, chẳng có gì có thể khẳng định cho điều đó. Có thể cô ấy đang cố nói với tôi rằng quán của họ đã hết sữa cả tuần nay, nhưng kết quả chẳng có gì thay đổi - ngày hôm nay tôi vẫn phải uống một ly espresso.

Trong các tình huống ngoài đời thực, thái độ ung dung của mọi người đối với state không phải là một vấn đề lớn. Tuy nhiên nhiều lập trình viên vẫn còn đang khá mơ hồ về state, và đó mới là vấn đề đáng nói.

Hãy xem như một webshop đơn giản chỉ xài thẻ tín dụng và không lập hóa đơn cho khách hàng, sử dụng một class Oder chứa phương thức này:

```
public boolean isComplete() {
        return isPaid() && hasShipped();
 }
```

Rất hợp lí đúng không? Kể cả nếu biểu thức được trích xuất độc đáo thành một phương thức thay vì sao chép ở mọi nơi, thì biểu thức này vẫn không nên được tồn tại và sự thật là nó đã nêu bật được một vấn đề. Tại sao ư? Bởi vì một đơn đặt hàng không thể được vận chuyển trước khi nó được thanh toán. Do đó, hasShipped không thể cho là đúng trừ khi isPaid đúng, là một phần làm nên một biểu thức rườm rà. Bạn có thể vẫn muốn isComplete để code được rõ ràng hơn, nhưng điều đó sẽ làm cho nó trở thành như thế này:

```
public boolean isComplete() {
        return hasShipped();
```

}

Trong công việc của mình , tôi luôn thấy được cả những lần kiểm tra thiếu và những lần kiểm tra dư. Đây chỉ là một ví dụ nhỏ, nhưng khi bạn tăng cường “hủy bỏ” và “trả lại”, nó sẽ trở nên ngày càng phức tạp, đồng thời những nhu cầu cho việc xử lý state tốt cũng ngày một gia tăng. Trong trường hợp này, một order chỉ có thể là một trong ba states riêng biệt:

- Trong quá trình: Có thể thêm vào hoặc loại bỏ items. Không thể vận chuyển.
- Thanh toán: Không thể thêm hoặc loại bỏ items. Có thể được vận chuyển.
- Vận chuyển: Hoàn thành. Không có thêm bất kì thay đổi nào được chấp nhận

Những states đó rất quan trọng và bạn cần kiểm tra xem bản thân có đang ở trong state mình mong đợi trước khi thực hiện các thao tác hay không, nếu không thì hãy di chuyển đến một state khác phù hợp hơn. Nói ngắn gọn thì bạn phải bảo vệ các đối tượng của mình một cách cẩn thận và tại một địa điểm thực sự hợp lí.

Nhưng bằng cách nào để bắt đầu thực hiện “suy nghĩ trong các states”? Trích xuất các biểu thức cho những phương thức có ý nghĩa là một sự khởi đầu không tồi, nhưng đó vẫn chỉ là bước thứ nhất. Nền tảng của việc này chính là để am hiểu về các state machines(2). Tôi biết là các bạn có những kí ức không tốt về CS class(3), nhưng tạm thời hãy quên chúng đi, các state machines không hề làm khó bạn chút nào. Hãy hình dung để làm cho chúng trở nên đơn giản và dễ trao đổi hơn. Hãy chạy thử code để chỉ ra các states hợp lệ và không hợp lệ, sau đó chuyển tiếp và giữ nguyên sự chính xác của chúng. Khi nghiên cứu về các State pattern(4), nếu thực sự thoải mái, hãy xem thử công nghệ Design by Contract, việc đó sẽ giúp bạn biết được rằng một state là hợp lệ bằng cách xác nhận dữ liệu đầu vào và đầu ra của mỗi phương thức được công khai.

Nếu state của các bạn không chính xác, chắc hẳn đó là một bug, và nếu bạn không ngưng nó lại, nguy cơ bị mất dữ liệu là rất cao. Một điều nữa, khi nhận thấy các state checks bị nhiễu, hãy học cách sử dụng một công cụ, tạo code, weaving(5), các khía cạnh để giải quyết vấn đề đó. Bất kể các bạn chọn cách tiếp cận nào, “suy nghĩ ở trong mỗi state” sẽ làm cho các đoạn code được đơn giản và mạnh mẽ hơn.

(*)Chú thích:

(1) State: State là một đối tượng Javascript lưu trữ dữ liệu động của một component. State là dữ liệu động, nó cho phép một component theo dõi thông tin thay đổi ở giữa các kết xuất (render) và làm cho nó có thể tương tác. State chỉ có thể được sử dung ở trong một component sinh ra nó.

(2) State machines: State machine là một mô hình quản lý quá trình chuyển trạng thái. Nó được ứng dụng rộng rãi trong nhiều ứng dụng. Một trong đó là về cài đặt các workflow.

(3) CS class: C# class.

(4) State pattern: State Pattern là một trong những mẫu thiết kế thuộc nhóm behavioral cho phép một object có thể biến đổi hành vi của nó khi có những sự thay đổi trạng thái nội bộ. Mẫu thiết kế này có thể được hiểu gần giống như Strategy, nó có thể chuyển đổi các chiến lược thông qua các phương thức được định nghĩa trong interface.

(5) Weaving: Quá trình kết nối các thành phần Aspect và Non-aspect(chẳng hạn như Object, Types) của một chương trình để tạo nên Advised Object gọi là Weaving.

Phần 85: Một cây làm chẳng nên non

Lập trình đòi hỏi suy nghĩ sâu sắc, và suy nghĩ sâu sắc đòi hỏi sự cô độc. Vì vậy, nhiều lập trình viên lựa chọn hướng đi này.

Cách tiếp cận "sói đơn độc" này đã nhường chỗ cho cách tiếp cận mang tính hợp tác hơn, giúp cải thiện chất lượng, năng suất và sự hài lòng trong công việc cho các lập trình viên. Cách tiếp cận này có các nhà phát triển phối hợp chặt chẽ với nhau và với cả những người phi chuyên môn- nhà phân tích kinh doanh và hệ thống, chuyên gia đảm bảo chất lượng và người dùng.

Điều này có ý nghĩa gì đối với các nhà phát triển? Làm một chuyên gia công nghệ không còn là tất cả nữa. Bạn phải phối hợp hiệu quả khi làm việc với những người khác.

Hợp tác không phải là hỏi và trả lời câu hỏi hay ngồi trong phòng họp. Đó là bắt tay với người khác để cùng hoàn thành công việc.

Tôi là “fan ruột” của lập trình cặp. Bạn có thể gọi đây là “sự hợp tác cực độ”. Là một nhà phát triển, kỹ năng của tôi phát triển khi tôi ghép cặp. Nếu tôi kém hơn cộng sự của mình ở mặt nào, tôi hoàn toàn có thể học hỏi từ kinh nghiệm của anh hoặc cô ấy. Khi tôi có thế mạnh ở mặt nào đó, tôi tìm hiểu thêm về những gì tôi biết và không biết khi phải tự mình giải thích vấn đề. Lúc nào cũng vậy, cả hai chúng tôi đều đưa ra ý kiến và học hỏi lẫn nhau.

Khi ghép cặp, mỗi chúng ta sẽ chia sẻ trải nghiệm lập trình của riêng mình- domain cũng như kỹ thuật- cho vấn đề hiện có, mang lại cái nhìn sâu sắc và kinh nghiệm độc đáo về việc viết phần mềm sao cho hiệu quả. Ngay cả trong trường hợp mất cân bằng kiến thức về domain hay kỹ thuật, người có kinh nghiệm hơn luôn học được điều gì đó từ người khác- có thể là một phím tắt mới, hay cơ hội tiếp xúc với một công cụ hoặc thư viện mới. Đối với những thành viên ít kinh nghiệm hơn, đây là cơ hội tuyệt vời để tăng tốc.

Lập trình cặp khá phổ biến, không chỉ dành riêng cho những người đề xuất phát triển phần mềm nhanh. Một số phản đối việc ghép đôi cho rằng "Vì sao tôi phải trả tiền cho hai lập trình viên chỉ để làm công việc của một người?" Phản ứng của tôi đối với điều này là bạn thực sự không nên nghĩ vậy. Tôi cho rằng việc ghép đôi làm tăng chất lượng, hiểu biết về domain và công nghệ, kỹ thuật (như thủ thuật IDE) và giảm thiểu tác động của rủi ro xổ số (một trong những nhà phát triển chuyên nghiệp của bạn trúng xổ số và nghỉ việc ngay hôm sau).

Giá trị lâu dài của việc học một phím tắt mới là gì? Làm thế nào để đo lường sự cải thiện chất lượng tổng thể của sản phẩm tạo ra từ các cặp ghép đôi? Làm thế nào để đo lường tác động của cộng sự của bạn khi ngăn cản bạn theo đuổi một cách tiếp cận nào đó để giải quyết một vấn đề khó khăn? Một nghiên cứu đã chỉ ra sự gia tăng 40% về hiệu suất và tốc độ (JT Nosek, "Trường hợp lập trình hợp tác", Truyền thông của ACM, tháng 3 năm 1998). Giá trị của việc giảm thiểu "rủi ro xổ số" của bạn là gì? Hầu hết những lợi ích này rất khó đo lường chính xác.

Ai nên ghép cặp với ai? Nếu bạn là người mới trong nhóm, điều quan trọng là tìm một thành viên có kiến thức tốt. Bạn cũng cần tìm một người có kỹ năng giao tiếp và khả năng huấn luyện tốt. Nếu bạn không có nhiều kinh nghiệm về domain, hãy ghép cặp với một thành viên là chuyên gia về domain.

Nếu bạn cảm thấy chưa thuyết phục, hãy thử hợp tác với đồng nghiệp của bạn. Ghép cặp để giải quyết một vấn đề thú vị. Hãy thử một vài lần xem cảm giác thế nào.

Phần 86: Hai cái sai tạo thành một cái đúng (và rất khó để fix)

Code không bao giờ nói dối nhưng chúng có thể liên hệ bản thân. contradict nghĩa là “mâu thuẫn”=> dịch là mẫu thuẫn với chính mình Và chúng dẫn đến những khoảnh khắc “Làm sao mà chúng có thể hoạt động được?”

Trong một bài phỏng vấn, trưởng nhóm thiết kế phần mềm Apollo 11 Lunar, Allan Klump, tiết lộ rằng phần mềm điều khiển động cơ tồn tại một bug có thể khiến cho bộ phận hạ cánh không ổn định. Tuy nhiên, một lỗi khác bù lại cho nó và phần mềm đã được dùng cho sự đáp Apollo 11 và 12 trước khi lỗi ấy có thể được tìm thấy và sửa chữa.

Xét một hàm trả về một trạng thái hoàn chỉnh. Tưởng tượng rằng nó sẽ trả về false trong khi nó phải là true. Tiếp theo ta tiếp tục tưởng tượng rằng cài hàm điều khiển quên kiểm tra giá trị trả về. Mọi thứ đều hoạt động vô cùng ổn định cho đến khi ai đó nhận ra điều này và thêm vào.
Hoặc xét một ứng dụng lưu trữ các trạng thái của nó bằng XML. Tưởng tượng rằng một trong những nốt đã bị viết sai thành TimeToLive thay vì TimeToDie như trong tài liệu đã ghi chép. Mọi thứ có vẻ ổn khi bộ phận viết code và đọc đều mắc chung một lỗi. Nhưng khi ta sửa một trong số chúng, hoặc là thêm một ứng dụng mới để đọc cùng những tài liệu ấy, khiến cho sự đối xứng bị phá hỏng tương tự với code.

Khi hai sai lầm trong đoạn code tạo nên một lỗi có thể thấy được,các phương pháp tiếp cận đều trở nên không hiệu quả. Nhà phát triển nhận được bản thông tin về bug, tìm và sửa sai lầm ấy sau đó thử test lại. Bản báo lỗi ấy vẫn tồn tại nhưng là do sai lầm thứ hai vẫn còn hoạt động. Vì vậy cái đầu tiên đã được loại bỏ, đoạn code tiếp tục đoạn kiểm tra cho tới khi sai lầm thứ hai được tìm thấy và sửa chữa. Tuy nhiên sai lầm thứ nhất quay lại, bản báo lỗi vẫn còn đó, điều tương đương xảy ra với sai lầm thứ hai. Quá trình lặp lại nhưng giờ đây nhà phát triển ấy dừng khắc phục hai lỗi và tiến hành quá trình khắc phục khác mà chắc chắn không bao giờ hoạt động.

Cuộc chơi của hai sai lầm trong đoạn code xuất hiện với một lỗi không chỉ khiến nó trở nên rất khó để khắc phục mà còn khiến cho các nhà phát triển lạc lối, chỉ để thấy họ thử đúng câu trả lời từ sớm.

Vấn đề này không chỉ xuất hiện trong code mà còn xuất hiện trong việc soạn thảo các tải liệu cần thiết. Và chúng còn có thể lan rộng từ nơi này sang nơi khác. Một lỗi trong code đã làm lu mờ lỗi nằm trong phần mô tả.

Không dừng lại ở đó, chúng còn có thể lan sang người. Người dùng học được rằng khi ứng dụng bảo trái có nghĩa là phải và từ đó họ điều chỉnh hành vi của họ. Họ còn có thể truyền sang người dùng khác: “ Hãy nhớ rằng khi ứng dụng bảo nhấn nút bên trái hãy nhấn nút bên phải”. Và khi lỗi được sửa khiến cho người dùng phải làm quen lại từ đầu.

Một lỗi sai đơn lẻ có thể dễ dàng phát hiện và khắc phục. Vấn đề ở đây chính là chính có quá nhiều nguyên nhân, do đó cần phải thay đổi nhiều lần, cuối cùng khiến việc sửa chữa trở nên khó khăn. Một phần là bởi vì những vấn đề này vô cùng dễ dàng để sửa do đó mọi người chủ quan nghĩ chúng có thể khắc phục nhanh chóng, tuy nhiên khó khăn ập đến sau đó.

Chúng tôi không có một lời khuyên nào giúp bạn xác định các lỗi mà nguyên nhân đến từ những sai lầm đáng tiếc trên. Hãy giữ cho mình một cái đầu lạnh và sẵn sàng kiểm tra tất cả các khả năng có thể xảy ra nếu cần thiết.

Phần 87: Ubuntu coding cho bạn bè

Chúng ta thường viết code riêng rẽ và code ấy phản ánh cách giải quyết của riêng chúng ta cho một vấn đề, cũng như một giải pháp mang tính cá nhân. Chúng ta có thể là một phần của nhóm, nhưng chúng ta vẫn làm việc độc lập. Chúng ta quên mất rằng code này- được tạo ra một cách độc lập, sẽ được thực thi, sử dụng, mở rộng và dựa vào bởi người khác. Thật dễ dàng để bỏ qua khía cạnh xã hội của việc tạo ra phần mềm. Lập trình phần mềm là một bài tập kỹ thuật kết hợp với một bài tập xã hội. Chúng ta cần ngẩng đầu khỏi màn hình máy tính thường xuyên hơn để nhận ra chúng ta không làm việc một mình và mọi người cùng chia sẻ trách nhiệm đối với việc tăng xác suất thành công cho tất cả, không chỉ riêng nhóm phát triển phần mềm.

Bạn có thể tự mình viết những code với chất lượng tốt, lạc trong tâm trí của bản thân trong suốt khoảng thời gian đó. Ở một khía cạnh nào đó, đó là một cách tiếp cận tự nhiên (không phải bản ngã trong kiêu ngạo, mà là trong cá nhân). Nó là điểm nhìn Zen và vào thời điểm viết code, mọi thứ xoay quanh bản thân bạn. Tôi luôn cố gắng sống cho hiện tại, bởi nó giúp tôi đến gần hơn với chất lượng tốt, nhưng đó là khi tôi sống trong khoảnh khắc của riêng mình. Còn khoảnh khắc của cả nhóm thì sao? Liệu khoảnh khắc của tôi có giống với khoảnh khắc của cả nhóm không?

Ở Zulu, triết lý của Ubuntu được tóm tắt là "Umfox ngum Ubuntu ngabantu" tạm dịch: "Tôi được là chính mình nhờ có những người xung quanh". Tôi trở nên tốt hơn nhờ có những hành động tốt của bạn. Mặt trái là bạn trở nên yếu kém hơn với những gì bạn làm khi tôi kém những gì tôi làm. Đối với các nhà phát triển, chúng ta có thể chuyển nó thành "Một người là nhà phát triển thông qua những nhà phát triển khác".Tương tự "Code là code thông qua code khác".

Chất lượng code tôi viết ảnh hưởng đến chất lượng code bạn viết. Vậy điều gì sẽ xảy ra khi code của tôi có chất lượng kém? Ngay cả khi code bạn viết rất tốt, thì khi bạn sử dụng code của tôi, chất lượng code của bạn sẽ giảm xuống. Bạn có thể áp dụng nhiều

mô hình và kỹ thuật để hạn chế thiệt hại, nhưng thiệt hại ấy đã được thực hiện rồi. Tôi đã khiến bạn phải làm nhiều hơn cần thiết chỉ đơn giản vì tôi đã không nghĩ về bạn khi sống trong khoảnh khắc của riêng mình.

Tôi có thể coi code của mình là tốt, nhưng tôi vẫn có thể làm cho nó tốt hơn với Ubuntu coding. Ubuntu code trông như thế nào? Nó là một code tốt. Nó không phải là về code, mà là về hành động tạo ra code đó. Coding cho bạn bè với Ubuntu sẽ giúp cả nhóm sống theo giá trị và củng cố các nguyên tắc của bạn. Người tiếp theo chạm vào code của bạn, bằng mọi cách, sẽ là người tốt hơn và là nhà phát triển tốt hơn.

Zen là cá nhân. Ubuntu là Zen cho một nhóm người. Rất hiếm khi chúng ta tạo code chỉ để cho cá nhân mình.

Phần 88: Unix tool là bạn

Nếu sắp bị đi đày đến một hoang đảo và buộc phải chọn giữa IDE và Unix tool, tôi sẽ ngay lập tức chọn Unix tool. Dưới đây là những lý do vì sao bạn nên thành thạo Unix tool.

Đầu tiên, IDE hướng tới các ngôn ngữ cụ thể, trong khi Unix tool có thể hoạt động với bất kỳ thứ gì xuất hiện ở dạng văn bản. Trong môi trường phát triển ngày nay, nơi các ngôn ngữ và ký hiệu mới xuất hiện hàng năm, học cách làm việc với Unix là một khoản đầu tư đáng giá.

Hơn nữa, trong khi IDE chỉ cung cấp những lệnh được nhà phát triển nghĩ ra, thì với Unix tool bạn có thể thực hiện bất kỳ tác vụ nào. Hãy nghĩ về chúng như các khối Lego (tiền- Bionicle cổ điển): Bạn tạo lệnh của riêng mình bằng cách kết hợp những Unix tool nhỏ nhưng linh hoạt.

Chẳng hạn, trình tự sau đây là một triển khai dựa trên văn bản phân tích chữ ký của Cunningham- một chuỗi các dấu chấm phẩy, dấu ngoặc và dấu ngoặc kép có thể tiết lộ rất nhiều về nội dung của tệp.

```
for i in *.java; do
    echo -n "$i: "
    sed 's/[^"{};]//g' $i | tr -d '\n'
    echo
done
```

Ngoài ra, mỗi thao tác IDE bạn học được dành riêng cho nhiệm vụ đó, ví dụ thêm một bước mới trong cấu hình debug của dự án. Ngược lại, mài giũa kỹ năng sử dụng Unix tool giúp bạn làm việc hiệu quả hơn. Ví dụ, tôi đã dùng sed tool được sử dụng trong chuỗi lệnh trước đó để biến đổi bản dựng của dự án để biên dịch chéo trên bộ xử lý.

Unix tool được phát triển trong thời đại mà một máy tính nhiều người dùng có 128kB RAM. Sự khéo léo trong thiết kế của họ nghĩa là ngày nay họ có thể xử lý các tập dữ liệu khổng lồ cực kỳ hiệu quả. Hầu hết các công cụ hoạt động như bộ lọc, xử lý chỉ một dòng duy nhất tại thời điểm đó, tức là không có giới hạn về lượng dữ liệu có thể xử lý. Bạn muốn tìm kiếm số lượng chỉnh sửa được lưu trữ trong bãi chứa nửa terabyte của Wikipedia tiếng Anh? Một thao tác đơn giản

```
grep '<revision>' | wc -l
```

sẽ cho bạn câu trả lời. Nếu bạn thấy một chuỗi lệnh hữu ích, bạn có thể chuyển nó thành tập lệnh shell, sử dụng một số cấu trúc lập trình mạnh mẽ, chẳng hạn dẫn dữ liệu vào các vòng lặp và điều kiện. Ấn tượng hơn nữa, các lệnh Unix hoạt động như các đường ống, giống như các lệnh trước đó, sẽ tự phân phối trong số nhiều đơn vị xử lý của CPU đa lõi hiện đại.

Các triển khai open source và trình giả lập của Unix tool làm cho nó khả dụng ở mọi nơi, ngay cả trên các nền tảng bị hạn chế tài nguyên, như trình phát đa phương tiện hàng đầu hoặc bộ định tuyến DSL. Các thiết bị như vậy không có khả năng cung cấp giao diện người dùng mạnh mẽ, nhưng chúng thường bao gồm ứng dụng BusyBox, cung cấp các công cụ được sử dụng phổ biến nhất. Và nếu bạn đang phát triển trên Windows, Cygwin cung cấp tất cả Unix tool, cả dưới dạng thực thi và dạng source code.

Cuối cùng, nếu không có công cụ nào phù hợp với nhu cầu của bạn, thì bạn có thể dễ dàng để mở rộng thế giới của Unix tool. Chỉ cần viết một chương trình (bằng bất kỳ ngôn ngữ nào bạn thích) theo một vài quy tắc đơn giản: Chương trình của bạn chỉ thực hiện một nhiệm vụ duy nhất; nó nên đọc dữ liệu dưới dạng văn bản từ input tiêu chuẩn của nó; và nó sẽ hiển thị kết quả- không được đánh dấu bởi các tiêu đề hay nhiễu loạn khác ở output tiêu chuẩn. Các tham số ảnh hưởng đến hoạt động của công cụ được đưa vào dòng lệnh. Thực hiện các quy tắc này và "Trái đất và mọi thứ trong đó là của bạn".

Phần 89: Sử dụng đúng thuật toán và cấu trúc dữ liệu

Một ngân hàng lớn với nhiều chi nhánh phàn nàn rằng các máy tính mới mà họ mua cho các giao dịch viên chạy quá chậm. Vào thời điểm đó ngân hàng điện tử và cây ATM không phổ biến như bây giờ. Mọi người đến ngân hàng thường xuyên hơn, và những chiếc máy tính chậm chạp đang khiến mọi người phải chờ đợi. Do đó, ngân hàng đe dọa sẽ phá vỡ hợp đồng với nhà cung cấp.

Nhà cung cấp đã gửi đến một chuyên gia phân tích và điều chỉnh hiệu suất để xác định nguyên nhân của sự chậm trễ. Anh ta sớm tìm thấy một chương trình tiêu thụ gần như toàn bộ dung lượng CPU. Sử dụng một công cụ định hình, anh ta phân tích chương trình và tìm thấy thủ phạm:

```
for (i = 0; i <strlen (s); ++ i) {if (... s [i] ...) ...}
```

Và một chuỗi trung bình gồm hàng nghìn ký tự. Code (được viết bởi ngân hàng) đã nhanh chóng được sửa đổi, và các giao dịch viên sống hạnh phúc mãi mãi về sau ....

Không phải là có cách tốt hơn việc sử dụng code được chia tỷ lệ không cần thiết theo phương pháp bậc hai sao? Mỗi hàm strlen đi qua một trong số hàng ngàn ký tự trong chuỗi để tìm ký tự null kết thúc. Chuỗi, ngược lại, không bao giờ thay đổi. Bằng cách

xác định trước độ dài của nó, lập trình viên có thể lưu hàng nghìn cuộc gọi vào strlen (và hàng triệu lần thực hiện vòng lặp):

```
n = strlen (s); for (i = 0; i <n; ++ i) {if (... s [i] ...) ...}
```

Mọi người đều biết câu ngạn ngữ "Trước hết hãy làm cho nó hoạt động, sau đó mới làm cho nó hoạt động nhanh" để tránh những cạm bẫy của tối ưu hóa vi mô. Nhưng ví dụ nêu trên sẽ khiến bạn tin rằng lập trình viên đã tuân theo Machiavellian adagio "Trước tiên hãy làm cho nó hoạt động chậm".

Sự thiếu suy nghĩ này bạn có thể đã bắt gặp nhiều lần. Và nó không chỉ là một thứ "không- phát- minh- lại- bánh xe". Đôi khi các lập trình viên mới bắt tay vào làm mà không thực sự suy nghĩ và đột nhiên họ "phát minh ra" bubble sort. Họ thậm chí còn khoe khoang về nó.

Mặt khác của việc chọn thuật toán phù hợp là lựa chọn cấu trúc dữ liệu. Nó có thể tạo ra sự khác biệt lớn: Sử dụng danh sách được liên kết với bộ sưu tập một triệu mục bạn muốn tìm kiếm- so với cấu trúc dữ liệu hoặc hệ nhị phân- sẽ có tác động lớn đến sự đánh giá của người dùng đối với chương trình của bạn.

Các lập trình viên không nên phát minh lại bánh xe, mà nên sử dụng các thư viện hiện có. Nhưng để có thể tránh các vấn đề như của ngân hàng, họ cũng cần được giáo dục về các thuật toán và cách chúng mở rộng quy mô. Có phải đó chỉ là trò chơi trong các trình soạn thảo văn bản hiện đại khiến chúng chậm như các chương trình trường học cũ như Wordstar vào những năm 1980? Nhiều người nói tái sử dụng trong lập trình là tối quan trọng. Tuy nhiên, các lập trình viên nên biết khi nào, cái gì và làm sao có thể sử dụng lại. Để có thể làm điều đó, họ cần có kiến thức về miền, các thuật toán và cấu trúc dữ liệu.

Một lập trình viên giỏi cũng nên biết khi nào cần sử dụng một thuật toán phức tạp. Ví dụ: nếu miền ra lệnh không bao giờ có nhiều hơn năm vật phẩm (như số xúc xắc trong trò Yahtzee), bạn biết rằng bạn chỉ có thể sắp xếp tối đa năm vật phẩm. Trong trường hợp đó, bubble sort thực sự là cách hiệu quả nhất để sắp xếp các mục. Ai rồi cũng có lúc gặp vận.

Vì vậy, hãy đọc những cuốn sách hay- và chắc chắn rằng bạn hiểu chúng. Nếu bạn thực sự đọc hiểu "Nghệ thuật lập trình máy tính" của Donald Knuth, bạn có thể gặp may mắn: Tìm một lỗi sai của tác giả và nhận được tờ séc trị giá 2,56 $ của Don Knuth.

Phần 90: Verbose Logging sẽ làm gián đoạn giấc ngủ của bạn

Khi tôi gặp một hệ thống đã được phát triển hay sản xuất trong một khoảng thời gian, dấu hiệu đầu tiên của sự cố luôn là một bản log rối rắm. Bạn hiểu ý tôi mà. Khi nhấp vào một liên kết trên một trang web sẽ dẫn đến một loạt các thông báo trong log hệ thống cung cấp. Quá nhiều logging sẽ trở nên vô ích như không có.

Nếu hệ thống của bạn giống như của tôi, khi công việc của bạn hoàn thành thì công việc của người khác mới bắt đầu. Sau khi hệ thống được phát triển; nếu bạn may mắn, nó sẽ sống một cuộc sống lâu dài và thịnh vượng để phục vụ khách hàng. Làm thế nào để bạn biết liệu có sự cố nào xảy ra trong quá trình hệ thống được sản xuất, và bạn sẽ đối phó với nó như thế nào?
Có thể ai đó sẽ phụ trách giám sát hệ thống cho bạn, hoặc bạn có thể tự giám sát lấy. Dù bằng cách nào, log sẽ là một phần của việc giám sát. Nếu có chuyện gì xảy ra và bạn phải thức dậy để đối phó với nó, tốt hơn hết nên có một lý do hợp lý. Nếu hệ thống của tôi sắp chết, tôi muốn biết. Nhưng nếu đó chỉ là một tiếng nấc, tôi thà tận hưởng giấc ngủ của mình còn hơn.

Đối với nhiều hệ thống, dấu hiệu đầu tiên cho thấy có gì đó không đúng là một bản tường trình được ghi vào log. Thông thường, đây sẽ là error log. Vì vậy, hãy tự giúp mình: Hãy chắc chắn ngay từ đầu rằng nếu có gì được ghi vào error log, bạn sẽ sẵn sàng bị đánh thức giữa đêm vì nó. Nếu bạn có thể mô phỏng load trên hệ thống trong quá trình kiểm tra, xem xét error log không- tiếng- ồn cũng là một dấu hiệu tốt cho thấy hệ thống của bạn đang hoạt động khá trơn tru. Hoặc được cảnh báo sớm nếu hệ thống có vấn đề.

Distributed system gia tăng độ phức tạp. Bạn phải quyết định làm thế nào để đối phó với một sự phụ thuộc thất bại từ bên ngoài. Nếu hệ thống của bạn phân tán, đây có thể là một sự cố phổ biến. Hãy chắc chắn đã tính đến điều này trong chính sách logging của bạn.

Nói chung, dấu hiệu tốt nhất cho thấy mọi thứ đều ổn là những tin nhắn ở mức ưu tiên thấp hơn đang tiến triển tốt. Tôi muốn có một thông điệp log cấp INFO cho mỗi sự kiện quan trọng.
Bản log lộn xộn là dấu hiệu cho thấy hệ thống sẽ trở nên khó kiểm soát khi đi vào sản xuất. Nếu bạn không mong đợi bất cứ điều gì xuất hiện trong error log, sẽ dễ dàng hơn nếu biết phải làm gì khi có gì đó thực sự xuất hiện.

Phần 91: Nguyên tắc WET làm giảm nghẽn cổ chai

Điều quan trọng của nguyên tắc DRY (đừng lặp lại bản thân) là thực hiện hoá ý tưởng mỗi kiến thức nhất định trong chương trình chỉ nên được thể hiện bởi một thứ. Nói cách khác, kiến thức ấy chỉ nên được cài đặt một lần duy nhất. Từ trái nghĩa của nó chính là WET(viết mọi lúc). Code của chúng ta WET khi chúng ta thực hiện hoá một kiến thức bằng nhiều sự cài đặt khác nhau.DRY và WET có sự ảnh hưởng vô cùng lớn đến hiệu năng được thể hiện bằng các số liệu cụ thể.

Hãy bắt đầu bằng việc xét một tính năng trên hệ thống của chúng ta, xem X, là cổ chai CPU (Các luồng xử lý hợp lại thành 1 số lượng “luồng” chính đi vào CPU để xử lý). Tiếp theo, giả sử X tiêu thụ 30% hiệu năng của CPU. Bây giờ hãy tưởng tượng X có mười cài đặt khác nhau, trung bình thì mỗi cài đặt này tiêu thụ hết 3%. Ở mức độ tiêu thụ hiệu năng này thì chúng ta có thể không cần quan tâm đến nó nếu như bạn đang tìm giải pháp nhanh chóng, giống như là chúng ta đã quên mất sự ùn tắc mà nó tạo ra. Tuy nhiên, giả sử rằng chúng ta bằng một cách nào đó phát hiện ra sự ùn tắc đó. Và bây giờ chúng ta đối mặt với vấn đề tìm kiếm và sửa chữa từng thiết lập ấy. Với WET

chúng ta có đến mười thiết lập để tìm và sửa chữa. Với DRY chúng ta có thể thấy chúng chiếm 30% hiệu năng nhưng chúng ta không phải sửa chúng 10 lần.

Không phải tôi đã nói sao, chúng ta không có thời gian dành cho việc tìm và sửa từng cái một.
Có một trường hợp mà chúng ta thường làm trái quy tắc DRY đó là nhu cầu về các dãy dữ liệu. Một kỹ thuật thường dùng để cài đặt các query chính duyệt qua bộ lưu trữ ấy và sau đó thực hiện query với từng phần tử:

```
public class UsageExample {
    private ArrayList<Customer> allCustomers = new ArrayList<Customer>();
    // ...
    public ArrayList<Customer> findCustomersThatSpendAtLeast(Money amount) {
        ArrayList<Customer> customersOfInterest = new ArrayList<Customer>();
        for (Customer customer: allCustomers) {
            if (customer.spendsAtLeast(amount))
                customersOfInterest.add(customer);
        }
        return customersOfInterest;
    }
}
```

Bằng việc cho client thấy dãy dữ liệu thô này, chúng ta đã làm trái quy tắc. Điều này không chỉ giới hạn khả năng sửa chữa, nó còn ép buộc người dùng code của bạn vi phạm DRY bởi từng thứ trong đó tiềm ẩn khả năng có thể bị thay đổi trong cùng query. Tình huống này có thể được tránh bằng cách loại bỏ việc thể hiện bộ lưu trữ trên khỏi API. Trong ví dụ này chúng tôi sẽ giới thiệu cho bạn một kiểu dữ liệu miền chuyên biệt mới gọi là CustomerList. Class mới này đóng vai trò vô cùng ý nghĩa trong miền chương trình. Nó sẽ đóng vai trò là ngôi nhà chung của tất cả các query.

Kiểu dữ liệu mới này cho phép chúng ta theo dõi liệu những query tiêu thụ hiệu năng như thế nào. Bằng việc hợp nhất những query này lại với nhau vào chung một class từ đó giúp dễ dàng đánh giá nhu cầu lựa chọn thứ cần được thể hiện, như là một

ArrayList, đến cho client. Điều này cho phép chúng ta sự tự do trong việc thay đổi cách thiết lập này mà không phải lo lắng về việc ảnh hưởng đến mối liên lạc của người dùng.

```
public class CustomerList {
    private ArrayList<Customer> customers = new ArrayList<Customer>();
    private SortedList<Customer> customersSortedBySpendingLevel = new
SortedList<Customer>();
    // ...
    public CustomerList findCustomersThatSpendAtLeast(Money amount) {
        return new
CustomerList(customersSortedBySpendingLevel.elementsLargerThan(amount));
    }
}

public class UsageExample {
    public static void main(String[] args) {
        CustomerList customers = new CustomerList();
        // ...
        CustomerList customersOfInterest =
customers.findCustomersThatSpendAtLeast(someMinimalAmount);
        // ...
    }
}
```

Trong ví dụ này, bằng việc bám chặt DRY đã cho phép chúng ta giới thiệu một chương trình lập chỉ mục thay thế với danh sách đã được sắp xếp gắn với giá trị phụ thuộc vào mức độ chi của khách hàng. Quan trọng hơn những chi tiết của chương trình trên, làm theo nguyên tắc DRY giúp chúng ta tìm và sửa chữa sự nghẽn cổ chai dễ dàng hơn so với việc thực hiện điều đó với code được viết theo nguyên tắc WET.

Phần 92: Khi coder và tester hợp tác lại với nhau

Điều kỳ diệu nào đó sẽ xảy ra khi người lập trình và kiểm thử phần mềm bắt đầu hợp tác với nhau.Ta sẽ dành ít thời gian để gửi lỗi qua lại thông qua hệ thống quản lí lỗi. Ta cũng không cần phải tốn nhiều thời gian cố gắng tìm ra đây là lỗi hay tính năng mới, và ta sẽ có nhiều thời gian phát triển phần mềm tốt hơn để đáp ứng nhu cầu của khách hàng. Bạn sẽ có nhiều cơ hội để hợp tác, thậm chí trước khi việc viết code bắt đầu.

Người kiểm thử phần mềm có thể giúp khách hàng viết và tự động hóa các bộ acceptance tests bằng cách sử dụng ngôn ngữ trong lĩnh vực của họ như là Fit (Framework for Integrated Test). Khi những bài tests này được gửi đến lập trình viên trước khi họ bắt đầu viết code, cả nhóm sẽ luyện tập Acceptance Test Driven Development (ATDD). Người lập trình sẽ viết những thứ cố định để chạy tests, và họ code để hoàn thành bài tests đó. Những bài tests này dần trở thành một phần của bộ hồi quy. Khi việc hợp tác này diễn ra, những bài kiểm tra chức năng sẽ được hoàn thành sớm, việc này cho phép thời gian để thử nghiệm thăm dò ở các điều kiện khác hoặc thông qua quá trình làm việc ở tình huống nói chung.

Chúng ta có thể tiến xa thêm một bước nữa. Là một người kiểm thử, tôi có thể cung cấp hầu hết các ý tưởng của bài tests trước khi lập trình viên bắt đầu code một tính năng mới. Khi tôi hỏi ý kiến của họ, họ hầu như lúc nào cũng đưa tôi những thông tin có ích cho các bộ tests tiếp theo được tốt hơn, hoặc giúp tôi tránh dành thời gian quá nhiều cho các bộ tests không cần thiết. Chúng tôi thường ngăn chặn được các lỗi vì các tests thể hiện rõ ý tưởng ban đầu. Ví dụ, trong một dự án tôi tham gia, những bộ Fit tests tôi gửi cho lập trình viên đã cho thấy kết quả mong đợi của một truy vấn để phản hồi cho cuộc tìm kiếm wildcard. Người lập trình đã hoàn toàn xong dự định code các từ khóa hoàn chỉnh. Chúng tôi đã có khả năng nói chuyện với khách hàng và giải thích chính xác trước khi bắt đầu code. Bằng cách hợp tác, chúng tôi đã ngăn được cái lỗi cũng như tiết kiệm được thời gian.

Lập trình viên cũng có thể hợp tác với người kiểm thử phầm mềm để tạo sự tự động hóa thành công. Họ hiểu good coding practices và có thể giúp người kiểm thử phần mềm thiết lập bộ tests tự động hóa mạnh mẽ để hoạt động cho cả nhóm. Tôi thường thấy những dự án test tự động hóa thất bại bởi vì chúng được thiết kế sơ sài. Bộ tests cố gắng kiểm tra quá nhiều hoặc người kiểm thử vẫn không hiểu rõ về công nghệ để tạo ra bộ tests có thể hoạt động độc lập. Người kiểm thử phần mềm thường là những người làm chậm trễ tiến độ công việc, vì vậy cho lập trình viên làm việc với họ trong các nhiệm vụ như tự động hóa rất hợp lí. Làm việc với họ, ta sẽ biết được những gì có thể kiểm tra sớm nhất, có lẽ bằng cách cung cấp một công cụ đơn giản, sẽ cho người lập trình các phản hồi khác giúp họ code tốt hơn sau này.

Chừng nào người kiểm thử phần mềm ngừng nghĩ rằng công việc duy nhất của họ là phá vỡ phần mềm và tìm lỗi trong mã code của người lập trình, lập trình viên ngừng nghĩ rằng không nên làm phiền những người kiểm thử, việc hợp tác sẽ cởi mở hơn. Khi lập trình viên nhận ra rằng họ cần có trách nhiệm nâng cao chất lượng code của họ, khả năng kiểm tra của code là tự nhiên, cả nhóm có thể cùng nhau tự động hóa các bộ tests quy hồi. Sự kỳ diệu khi làm việc nhóm sẽ bắt đầu.

Phần 93: Viết code như thể bạn phải hỗ trợ nó đến hết đời

Bạn có thể hỏi 97 người liệu lập trình viên nên biết gì và làm gì, và bạn sẽ nhận được 97 câu trả lời khác nhau. Điều này có thể vừa áp đảo vừa đáng sợ. Mọi lời khuyên đều tốt, mọi nguyên tắc đều hợp lý, và mọi câu chuyện đều hấp dẫn, nhưng bạn nên bắt đầu từ đâu? Quan trọng hơn, một khi bạn bắt đầu, làm thế nào để theo kịp những thực tiễn tốt nhất bạn đã học được và biến chúng thành một phần không thể thiếu trong quá trình thực hành lập trình của bạn?

Tôi nghĩ câu trả lời nằm trong suy nghĩ của bạn hoặc, rõ ràng hơn, trong thái độ của bạn. Nếu bạn không quan tâm đến nhà phát triển, tester, quản lý, người tiếp thị và bán

hàng cũng như người dùng, thì bạn sẽ không bị thúc đẩy sử dụng Phát triển dựa trên thử nghiệm hoặc viết comment rõ ràng trong code của bạn. Tôi nghĩ có một cách đơn giản để điều chỉnh thái độ của bạn và tạo động lực để cung cấp sản phẩm với chất lượng tốt nhất:

Viết code như thể bạn phải hỗ trợ nó cho đến hết cuộc đời.

Nếu bạn chấp nhận khái niệm này, những điều tuyệt vời sẽ đến. Nếu bạn chấp nhận rằng bất kỳ nhà tuyển dụng nào- cả trong quá khứ lẫn hiện tại có quyền gọi cho bạn vào lúc nửa đêm, yêu cầu bạn giải thích các lựa chọn bạn đã thực hiện khi viết phương pháp fooBar, bạn sẽ dần cải thiện để trở thành một lập trình viên chuyên nghiệp. Bạn sẽ tự nhiên nghĩ ra những phương thức và biến tốt hơn. Bạn sẽ tránh được việc viết những code dài hàng trăm dòng. Bạn sẽ tìm kiếm, tìm hiểu và sử dụng design pattern. Bạn sẽ viết comment, kiểm tra và tái cấu trúc liên tục code của bạn. Hỗ trợ tất cả code bạn từng viết cho đến hết đời cũng là một nỗ lực có thể mở rộng. Do đó, bạn không có lựa chọn nào khác ngoài việc trở nên tốt hơn, thông minh hơn và hiệu quả hơn.

Những code bạn viết nhiều năm trước vẫn ảnh hưởng đến sự nghiệp của bạn, dù bạn có thích hay không. Bạn để lại dấu vết về kiến thức, thái độ, sự kiên trì, tính chuyên nghiệp, mức độ cam kết và mức độ thích thú trên mọi phương pháp, class hay module bạn thiết kế và viết. Mọi người sẽ hình thành ý kiến về bạn dựa trên những code mà họ thấy. Nếu những ý kiến đó tiêu cực, bạn sẽ nhận được ít hơn từ sự nghiệp của bạn hơn bạn mong muốn. Hãy chăm chút sự nghiệp của bạn, của khách hàng và của người dùng với mọi dòng code- viết code như thể bạn phải hỗ trợ nó cho đến giây phút cuối cùng của cuộc đời mình.

Phần 94: Xây dựng những hàm nhỏ bằng ví dụ

Chúng ta bao giờ cũng muốn code được viết một cách chính xác, đồng thời có những bằng chứng cho tính đúng đắn của code. Nó có thể giúp ta nghĩ về vấn đề “kích thước”

của hàm đấy. Không phải về mặt số lượng code mà chúng ta viết trong hàm - mặc dù chúng rất thú vị - mà là về kích thước của các phép tính toán của chúng ta sử dụng trong hàm.

Ví dụ trong trong một trò chơi của Go có một điều kiện gọi lại atari thể hiện các viên đá của người chơi có thể bị cướp bởi đối thủ của họ: Một viên đó mà có 2 hoặc nhiều hơn những khoảng trống kề cạnh (gọi là liberties) không phải là atari. Việc đếm các liberties của một viên đá có thể gặp khó khăn tuy nhiên nó sẽ giúp việc xác định atari một cách dễ dàng. Chúng ta có thể bắt đầu viên hàm như thế này:

```
boolean atari(int libertyCount)
        libertyCount < 2
```

Đây chỉ là một phần nhỏ của cả đoạn code. Một hàm số toán học có thể được hiểu là một tập hợp, vài tập con của tích đề các của nó là miền giá trị (ở đây là, int) và phạm vi(ở đây, boolean). Nếu tập hợp các giá trị trên có kích thước tương tự trong Java thì chúng ta sẽ có 2L*(integer.MAX_VALUE +(-1L*Integer.MIN_VALUE) +1L) hoặc là 8,589,934,592 phần tử của tập int*boolean. Một nửa trong số chúng chính là những phần tử của tập con của hàm chúng ta đang viết. Chính vì thế để cung cấp những chứng minh hàm mà chúng ta viết chính xác chúng ta cần phải thử khoảng 4.3*10^9 ví dụ.

Điều này xác nhận rằng test không chứng minh được bugs không có khả năng xuất hiện trong code của chúng ta mà chúng chỉ có thể thể các tính năng của code. Nhưng chúng ta vẫn gặp phải vấn đề về kích thước.

Chính miền xác định của vấn đề sẽ giúp chúng ta hiểu ra. Bản chất của Go chính là số lượng Liberties của một viên đó không phải là một giá trị bất kì mà chúng chỉ nằm trong tập {1,2,3,4}. Chính vì thế chúng ta có thể thay thế đoạn code trên bằng:

```
libertyCount = {1,2,3,4}
boolean atari(LibertyCount libertyCount) libertyCount == 1
```

Bằng đoạn code này chúng ta có thể dễ dàng để làm việc cùng hơn: những tính toán của hàm giờ đây đã trở thành một tập hợp với nhiều nhất 8 phần tử. Bằng bốn ví dụ trên chúng ta đã nắm chắc được bằng chứng chứng minh cho tính đúng đắn của chương trình. Đó chính là lý do vì sao chúng ta nên sử dụng những kiểu có sự tương đồng gần nhất với miền xác định của vấn đề để viết code thay vì dùng những kiểu nguyên sơ. Hơn nữa việc sử dụng các dữ liệu càng giống với miền xác định của của vấn đề có thể khiến cho hàm xử lý của chúng ta trở nên gọn gàng hơn. Chỉ có một cách để tìm ra những vấn đề đó chính là tìm những ví dụ để kiểm tra vấn đề nhằm tính toán miền xác định của vấn đề trước khi bắt tay vào việc viết hàm.

Phần 95: Viết Tests Cho Mọi Người

Bạn đang viết các bài kiểm tra tự động cho các bài code của bạn. Xin chúc mừng! Bạn viết các bài kiểm tra đó trước khi viết code? Điều này thậm chí còn tốt hơn! Việc này sẽ giúp bạn trở thành một trong những người chấp nhận công nghệ tiến gần hơn với thực hành kỹ thuật phần mềm. Nhưng bạn đã viết tests tốt chưa? Bạn có thể thể hiện như thế nào? Cách duy nhất là hãy tự hỏi “Mình đang viết tests cho ai?” Nếu câu trả lời là “Cho chính mình, để giảm bớt việc sửa lỗi ấy mà” hay “Cho trình biên dịch, để chúng có thể hoàn thiện hơn” và điều kỳ quái là bạn chẳng viết tests tốt nhất có thể. Vậy bạn nên viết tests cho ai? Cho những người cố gắng hiểu code của bạn.

Những tests tốt đóng vai trò như là một tài liệu tham khảo cho bài code mà họ đang kiểm tra. Họ mô tả quá trình làm việc của mã code, cho từng trường hợp sử dụng bộ test:

- Mô tả bối cảnh, điểm bắt đầu hay điều kiện thích hợp.
- Minh họa phần mềm được hoạt động như thế nào.
- Mô tả kết quả hay điều kiện xác định sau cùng.

Trường hợp sử dụng khác nhau sẽ cho ra các kết quả tương đối khác nhau. Người cố gắng để hiểu code của bạn phải có khả năng tìm hiểu số ít các tests và nghi vấn so

sánh ba phần trên của các tests với nhau, có khả năng thấy được điều gì làm cho phần mềm hoạt động khác biệt. Mỗi test phải được minh họa rõ ràng mối liên hệ nguyên nhân-kết quả giữa ba phần của nó. Điều này có nghĩa rằng những thứ không hiện hữu trong test cũng quan trọng không kém. Code quá nhiều làm cho người đọc trở nên mất tập trung bởi những chi tiết nhỏ nhặt không quan trọng. Bất cứ khi nào có thể ẩn đi những chi tiết nhỏ nhặt và sử dụng hàm để gọi lại chúng- việc tái cấu trúc lại bằng Extract Method sẽ giúp bạn rất nhiều. Và chắc chắn rằng bạn đặt tên thể hiện được cụ thể trường hợp sử dụng cho test để người đọc chẳng cần phải dò lại để nhớ lại từng trường hợp đó là gì. Giữa chúng, tên của test class và class method nên bao gồm ít nhất điểm bắt đầu và cách phần mềm hoạt động. Điều này cho phép toàn bộ test được xác nhận thông qua một lần quét nhanh các tên của method. Cũng có ích khi bao gồm cả kết quả mong đợi vào tên của bộ kiểm tra method miễn là điều này không làm cho cái tên quá dài.

Kiểm tra tests của bạn là một ý tưởng hay. Bạn có thể xác nhận cách chúng phát hiện ra các lỗi mà bạn nghĩ rằng chúng phát hiện lỗi bằng cách thêm nó vào code sản xuất (một bản sao chép riêng của bạn mà bạn tất nhiên sẽ không vứt đi). Hãy chắc chắn chúng sẽ báo cáo lỗi một cách có ích và có nghĩa. Bạn cũng nên chắc chắn rằng người đọc sẽ hiểu được rõ ràng tests của bạn. Cách duy nhất để thực hiện được điều đó là nhờ một ai không biết gì về code của bạn đọc tests và kể lại những gì họ học được. Hãy lắng nghe cẩn thận những gì họ nói. Nếu họ không thiểu thứ gì đó rõ ràng, điều đó có lẽ không phải vì họ không sáng suốt. Nhiều khả năng là do bạn viết không rõ ràng. (Được rồi, hãy đổi vị trí - bạn sẽ đọc các tests của họ!)

Phần 96: Để tâm đến code

Không cần đến Sherlock Holmes để biết rằng các lập trình viên giỏi sẽ viết code tốt. Lập trình viên kém thì... không. Họ tạo ra những điều quái dị mà chúng ta phải dọn dẹp. Bạn muốn làm tốt, phải không? Bạn muốn trở thành một lập trình viên giỏi.

Code tốt không tự nhiên xuất hiện từ hư không; cũng không phải nhờ may mắn khi các hành tinh xếp thẳng hàng. Để có được code tốt, bạn phải làm việc chăm chỉ. Và bạn sẽ chỉ có được code tốt khi bạn thực sự chú tâm đến nó.
Khả năng lập trình tốt không được sinh ra từ năng lực kỹ thuật đơn thuần. Tôi đã gặp các lập trình viên có trí thông minh, những người có thể tạo ra những thuật toán mãnh liệt và ấn tượng, những người nắm lòng tiêu chuẩn ngôn ngữ của họ, nhưng lại là những người viết những code tệ nhất. Thật đau đớn khi đọc, sử dụng và sửa đổi. Tôi đã gặp những lập trình viên khiêm tốn hơn, làm việc với những code cơ bản, nhưng chương trình họ viết đầy thanh lịch và biểu cảm. Làm việc cùng họ như thể một niềm vui vậy.

Dựa trên kinh nghiệm nhiều năm trong ngành công nghiệp phần mềm, tôi đã đi đến kết luận rằng sự khác biệt giữa lập trình viên được và lập trình viên tuyệt vời nằm ở thái độ. Lập trình tốt nằm ở cách tiếp cận chuyên nghiệp và mong muốn viết phần mềm tốt nhất có thể, bất chấp những ràng buộc và áp lực từ Thế giới thực của ngành công nghiệp phần mềm.

Code đến địa ngục được trải đầy những ý định tốt. Để trở thành một lập trình viên xuất sắc, bạn phải vượt lên trên cả những ý định tốt và thực sự quan tâm đến code- thúc đẩy những quan điểm tích cực và thái độ lành mạnh. Một code tuyệt vời sẽ được chế tác cẩn thận bởi các nghệ nhân bậc thầy, chứ không phải bị hack bởi các lập trình viên cẩu thả hay dựng lên một cách bí ẩn bởi các bậc thầy mã hóa tự xưng.

Bạn muốn viết code tốt. Bạn muốn trở thành một lập trình viên giỏi. Vì vậy, bạn quan tâm đến code:

- Trong bất kỳ tình huống mã hóa nào, bạn từ chối hack thứ gì đó chỉ trông có vẻ hoạt động. Bạn cố gắng tạo ra những code chính xác, rõ ràng (và có cả các test để chứng minh độ chính xác).
- Bạn viết những code mà các lập trình viên khác có thể dễ dàng tiếp nhận và hiểu được, có thể duy trì được (để trong tương lai bạn hoặc các lập trình viên khác sẽ dễ dàng sửa đổi) và chính xác (bạn có thể thực hiện tất cả các bước để xác định rằng bạn đã giải quyết vấn đề, không chỉ làm cho nó trông giống một chương trình đang hoạt động).

- Bạn phối hợp tốt với các lập trình viên khác. Không có lập trình viên nào là “một hòn đảo”. Rất ít lập trình viên làm việc một mình; hầu hết làm việc theo nhóm trong công ty hoặc trong một dự án open source. Bạn xem xét các lập trình viên khác và xây dựng code mà người khác có thể đọc. Bạn muốn nhóm viết phần mềm tốt nhất có thể, thay vì làm cho bản thân trông thông minh.
- Bất cứ khi nào bạn tiếp cận một đoạn code, bạn cố gắng khiến nó trở nên tốt hơn lúc bạn tìm thấy (cấu trúc tốt hơn, được kiểm tra tốt hơn, dễ hiểu hơn...).
- Bạn quan tâm đến code và lập trình, vì vậy bạn không ngừng học hỏi những ngôn ngữ, thành ngữ và kỹ thuật mới. Nhưng bạn chỉ áp dụng khi thích hợp.

May mắn thay, bạn đang đọc tập hợp lời khuyên này vì bạn thực sự để tâm đến code. Và nó cũng chú ý đến bạn. Đó là đam mê của bạn. Chúc các bạn lập trình vui vẻ. Thưởng thức việc phân tích code để giải quyết những vấn đề khó khăn. Sản xuất những phần mềm khiến bạn tự hào.

Phần 97: Khách Hàng Không Chắc Chắn Những Gì Họ Nói

Tôi vẫn chưa gặp được người khách hàng nào có thể thoải mái nói tôi nghe những gì họ muốn, hay cung cấp nhiều thông tin. Vấn đề là khách hàng sẽ ít khi nói toàn bộ sự thật. Thường thì họ không nói dối, nhưng họ trò chuyện với vị trí khách hàng chứ không phải là một nhà phát triển. Họ sử dụng từ ngữ và bối cảnh của họ và bỏ qua những chi tiết cần thiết. Họ tự cho rằng bạn đã làm việc ở công ty được 20 năm, giống như họ vậy. Điều này cũng có nghĩa là họ không thực sự biết rằng mình muốn gì! Một vài người có thể hiểu “tình hình” rõ hơn, nhưng họ hiếm khi có khả năng nói lại chi tiết những gì họ nhận thấy. Có thể những người khác sẽ không có cái nhìn hoàn thiện, nhưng họ biết họ cần gì. Như vậy, sao bạn có thể chắc chắn giao một dự án phần mềm cho người thậm chí còn không nói với bạn toàn bộ những gì họ muốn? Đơn giản thôi. Hãy tương tác với họ nhiều hơn.

Hãy đưa ra các yêu cầu với khách hàng ngay từ đầu và thường xuyên. Đừng chỉ thuật lại những gì họ muốn bằng lời của họ. Hãy nhớ rằng: Họ không chắc chắn những gì họ yêu cầu bạn đâu. Tôi thường thay đổi một số từ trong cuộc trò chuyện và quan sát phản ứng của họ. Bạn sẽ rất kinh ngạc bởi số lần những customer và client có ý tưởng hoàn toàn khác nhau. Song người mà nói với bạn những gì anh ta muốn trong dự án phần mềm của anh ta sẽ sử dụng lộn xộn các thuật ngữ và mong bạn theo dõi những gì anh ta đang đề cập đến. Bạn sẽ cảm thấy lúng túng và điều này sẽ ảnh hưởng đến phần mềm bạn đang viết.

Hãy thảo luận các vấn đề với khách hàng của bạn nhiều lần trước khi cho rằng bạn thực sự hiểu họ cần gì. Cố gắng đề cập vấn đề với họ hai hoặc ba lần. Trò chuyện về những thứ diễn ra trước và sau chủ đề mà bạn và họ đang thảo luận sẽ khiến không khí trở nên tốt hơn. Nếu có thể, hãy nhờ nhiều người kể cho bạn nghe chủ đề đó trong các cuộc giao tiếp khác nhau. Nghe những câu chuyện khác nhau từ họ sẽ giúp bạn biết được những điều bạn muốn. Hai người cùng nói về một chủ đề thường sẽ trở nên mâu thuẫn với nhau. Tốt nhất là bạn nên thảo luận về những bất đồng và đưa ra kết luận trước khi bạn bắt đầu công việc với phần mềm cực-kì-phức-tạp của bạn.

Hãy sử dụng các vật dụng trực quan trong cuộc đối thoại. Bạn có thể làm việc này đơn giản như dùng một tấm bảng trắng trong cuộc gặp mặt, hay dễ như giai đoạn tạo giao diện nền trong thiết kế, hay phức tạp như tạo một nguyên bản hoạt động. Làm những việc trên trong suốt quá trình giao tiếp sẽ làm thu hút sự chú ý lâu dài và tăng khả năng ghi nhớ thông tin. Biết được điều này sẽ giúp cho dự án của bạn tốt hơn nhiều đấy.

Trước đây, khi tôi là “lập trình viên đa phương tiện” của một nhóm sản xuất các dự án lớn. Một khách hàng đã mô tả rất nhiều quan điểm và cảm xúc của họ về dự án. Bảng màu chung được thảo luận trong cuộc đối thoại đã chọn nền màu đen cho buổi thuyết trình. Chúng tôi đã nghĩ mọi việc đã xong. Nhóm thiết kế đồ họa đã nhanh chóng cho ra hàng trăm tập tin lớp đồ họa. Chúng tôi mất rất nhiều thời gian để chỉnh sửa ở các công đoạn cuối cùng. Vào ngày ra mắt sản phẩm với người khách ấy, và chúng tôi đã rất hoảng hốt. Khi cô ấy nhìn thấy sản phẩm, chính xác những gì cô ấy nói là “Khi tôi nói màu đen, thì ý của tôi là màu trắng.” Như bạn đã thấy đấy, sẽ không bao giờ có quan điểm rõ ràng đâu.

Thông tin bản quyền:

Thông tin bản dịch và thành viên team dịch:

Phiên bản: v1.0
Phát hành ngày: 31/03/2020
Bởi: Cộng đồng học lập trình miễn phí CodersX
Website: https://coders-x.com

Lưu ý: Đây là phiên bản dịch v1.0, để cập nhật và tải về bản dịch mới nhất, chính xác nhất, vui lòng truy cập địa chỉ dưới đây:
https://coders-x.com/97-things
https://laptrinhcuocsong.com/97-things.html

Các thành viên team dịch CodersX:

- Thịnh Phạm
- Bùi Văn Nguyện
- Đỗ Thành Nhân
- Henry Thành
- Hoàng Tiến Thịnh
- Lê Thành Đạt
- Mạnh Tuấn Trần
- Minh Hiếu
- Minh Khôi
- Minh Tâm
- Ngô Kim Tuyết
- Nguyễn Ích Hòa
- Nguyễn Nam Long
- Nguyễn Ngọc Sơn
- Nguyễn Tấn Phát
- Nguyễn Xuân Hoàng
- No Promise
- Phan Xuân Dũng
- Tăng Tường Thoại
- Trần Phương Nam