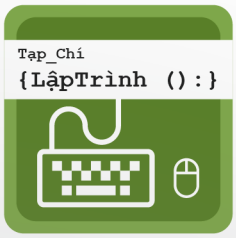


Tạp chí LẬP TRÌNH

tapchilaptrinh.vn

Số phát hành Vol.3 03 2020

Cứ yên tâm mà chọn nghề lập trình đi

Cấu trúc dữ liệu

DevOps

Tài nguyên Học lập trình

TOP 5 bí quyết học tập thượng hạng cho Coder

Vi-rut Corona ảnh hưởng gì đến giới công nghệ?

LỜI NGỎ

Chào các bạn,

Các ấn phẩm hằng tháng của Tạp chí Lập trình đã quay trở lại phục vụ bạn đọc sau một thời gian dài ngủ đông. Trong những năm qua, đội ngũ biên tập và các tác giả của Tạp chí Lập trình đã tạm gác lại công việc yêu thích của mình ở Tạp chí Lập trình để dành thời gian cho các ưu tiên khác trong cuộc sống. Thời gian quả là trôi qua rất nhanh, thấm thoắt cũng đã 7 năm trôi qua kể từ lần cuối cùng một ấn phẩm của Tạp chí Lập trình đến được với các bạn. Những thành viên của Tạp chí Lập trình không thể quên được sự ủng hộ của bạn học trong những năm tháng trước đây, những lời động viên, những lời khen ngợi luôn là động lực để các anh chị em tiếp tục sản xuất nội dung và đóng góp các giá trị mới cho cộng đồng.

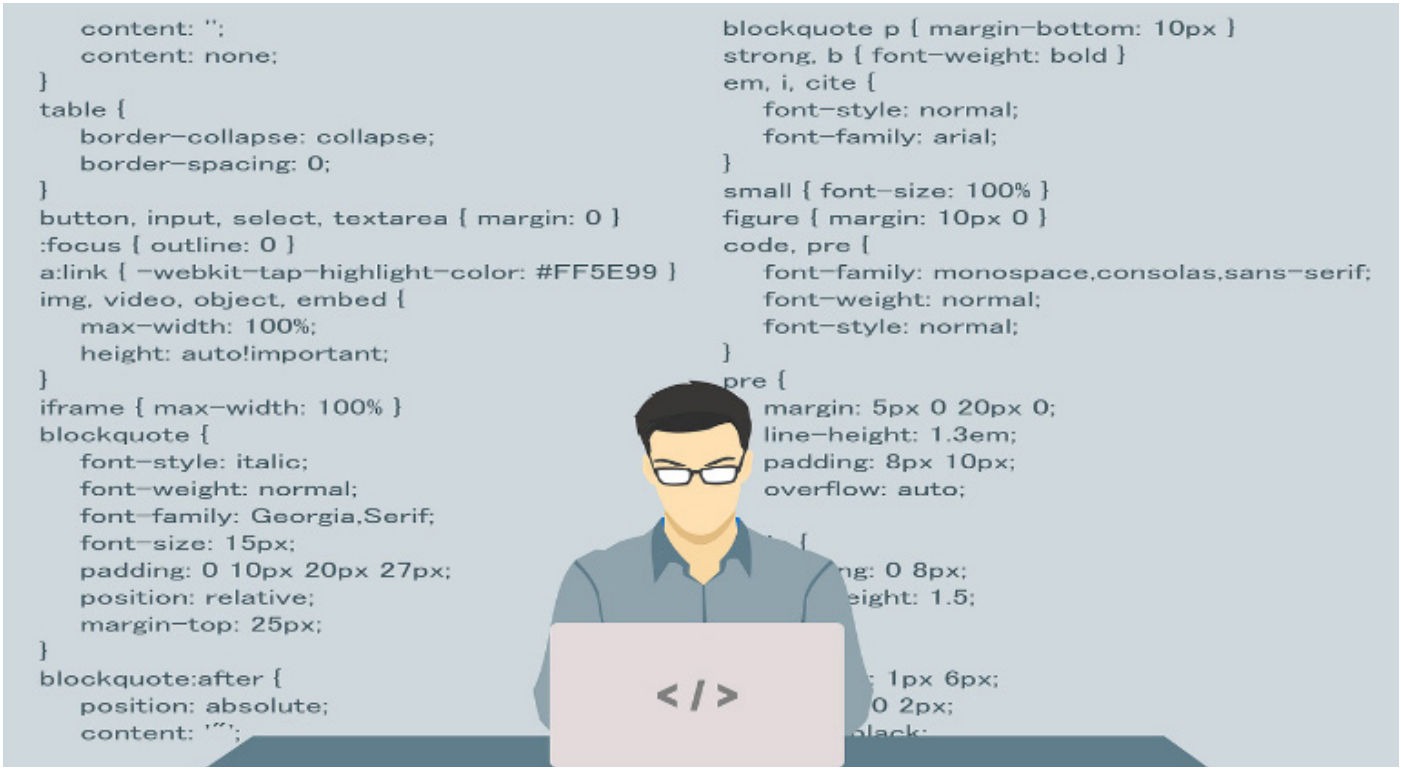
Tạp chí Lập trình đã trở lại trong một hoàn cảnh rất khác so với trước đây, nhưng mục đích thì không thay đổi: Đóng góp giá trị cho cộng đồng lập trình viên Việt Nam thông qua những nội dung chất lượng. Những khác biệt lớn lao của bây giờ so với 7 năm trước có thể kể đến như: cộng đồng người đọc, nền tảng công nghệ, các xu hướng công nghệ, tốc độ thay đổi của công nghệ, nền tảng giao tiếp, thói quen đọc, thói quen học, điều kiện học… Đây sẽ là những thử thách mới và đồng thời cũng là những điều kiện mới để Tạp chí Lập trình có thể một lần nữa tạo được chỗ đứng trong làng lập trình Việt Nam.

Ban biên tập Tạp chí Lập trình sẽ nỗ lực hết sức để có thể mang đến cho cộng đồng những nội dung cập nhật, chất lượng và hữu ích về thế giới công nghệ nói chung và lập trình nói riêng. Hẹn gặp mọi người thường xuyên ở các bài viết hằng ngày cũng như các ấn phẩm hằng tháng.

Cảm ơn các bạn.

Ban biên tập.

SỐ NÀY CÓ GÌ



CỨ YÊN TÂM MÀ CHỌN NGHỀ LẬP TRÌNH ĐI

Cách đây một năm, trong một buổi hội thảo với các bạn sinh viên CNTT của một trường khá to ở đất Hà thành, tôi hỏi một bạn sinh viên rằng bạn ấy kỳ vọng mức lương bao nhiêu sau khi ra trường, bạn ấy không ngần ngại và trả lời rằng 20 triệu, hỏi ra mới biết là bạn ấy đang học Công nghệ thông tin. Thật sự bất ngờ, mặc dù tôi đã làm trong ngành phần mềm hơn chục năm, cũng gần chừng đó năm kinh nghiệm trong việc đào tạo các thế hệ lập trình viên, nhưng cũng không hiểu được ngay tại sao các bạn sinh viên lại kỳ vọng cao như thế.

Gần đây, càng tiếp xúc nhiều với các bạn sinh viên và các bạn lập trình viên mời tập tễnh vào nghề, tôi mới nhận ra rằng đang có một sự ảo tưởng không hề nhẹ ở một bộ phận không nhỏ các bạn trẻ này. Có thể thấy rằng, điều này là hệ quả của nhiều năm sôi sục của ngành Công nghệ thông tin Việt Nam trong thời gian gần đây.

Như chúng ta cũng đã biết, Việt Nam là một trong số các nước sản xuất phần mềm lớn nhất thế giới, cùng với các quốc gia khác như Ấn Độ, Brazil, Trung Quốc, Philipin... Một trong những lợi thế của Việt Nam đó là giá nhân công rẻ, chi phí cho lập trình viên ở Việt Nam thấp hơn so với các nước phát triển như Nhật Bản, Mỹ, các nước châu Âu. Tuy nhiên, trình độ của lập trình viên Việt Nam vẫn ở mức thấp, do đó thông thường chỉ được nhận những công việc gia công đơn giản. Các công việc phức tạp, khó khăn (chẳng hạn như phân tích và thiết kế, lập trình nền tảng...) thường được thực hiện ở nước ngoài, sau đó chuyển giao những công việc đơn giản hơn cho các nhóm ở Việt Nam (chẳng hạn như kiểm thử, viết mã nguồn dựa trên thiết kế có sẵn...).

Tình trạng này cũng giống như ngành

may mặc của chúng ta vậy. Trong bao nhiêu năm nay, Việt Nam là một trong những nước sản xuất các sản phẩm may mặc lớn nhất thế giới, tuy nhiên chúng ta lại không có những sản phẩm thời trang nổi bật trên thị trường quốc tế. Điều này cũng xuất phát từ việc giá nhân công ở Việt Nam khá rẻ. Và các công việc gia công may mặc ở Việt Nam cũng được thực hiện theo các dây chuyền và thiết kế sẵn có.

Những năm gần đây đã có những chuyển biến đáng ghi nhận trong ngành sản xuất phần mềm ở Việt Nam theo hướng nâng cao dần chất lượng của công đoạn sản xuất và đồng thời thu nhập của nhân lực công nghệ thông tin cũng được nâng lên. Điều này xuất phát từ thực tế khách quan là các phần mềm ngày nay không còn đơn giản như trước, mà càng ngày càng phức tạp hơn, đòi hỏi yêu cầu cao hơn về cả mặt tính năng nghiệp vụ cũng như quy trình phát triển và chất lượng. Do đó, các công việc trình độ thấp ngày càng ít đi, thay vào đó là các công việc đòi hỏi trình độ cao hơn của lập trình viên. Thể hiện rất rõ là ngày càng có nhiều các công ty Việt Nam chuyển dần từ mô hình outsourcing sang mô hình offshoring, tham gia sâu vào việt thiết kế và hình thành sản phẩm. Các nhóm ở Việt Nam cũng có cơ hội để làm việc chung với các lập trình viên từ khắp nơi trên thế giới, do đó cần nâng cao trình độ để có thể cộng tác một cách tương xứng.

Cũng trong những năm gần đây, nhiều công ty nước ngoài mở chi nhánh ở Việt Nam để trực tiếp sản xuất phần mềm thay vì thuê các công ty Việt Nam. Cùng với sự bùng nổ của phong trào khởi nghiệp, đặc biệt là các khởi nghiệp trong lĩnh vực công nghệ hoặc có sử dụng công nghệ, môi trường Công nghệ thông tin ở Việt Nam càng trở nên sôi động, nhân lực công nghệ thông tin ở Việt nam lại trở thành một chủ đề mấu chốt cần quan tâm.

Theo một con số được công bố trên Thời báo kinh tế Sài Gòn, mỗi năm Việt nam thiếu **80.000 nhân lực** trong ngành công nghệ thông tin, trong khi các trường đại học và các cơ sở đào tạo chỉ cung cấp được khoảng **32.000**. Phần thiếu hụt còn lại là rất lớn, dẫn đến tình huống các công ty giành giật nhân sự của nhau. Trước đây, nghỉ việc hoặc chuyển công ty là một việc ít khi xảy ra và nếu xảy ra thì đều được cân nhắc rất kỹ. Còn bây giờ, nhảy việc là xu hướng. Theo một khảo sát của Vietnamworks, hơn 60% những người được hỏi đều có ý định nhảy việc trong vòng 6 tháng tới.

Cách đơn giản nhất để thu hút nhân sự đó là nâng mức lương và đãi ngộ. Đối với các công ty nhỏ hoặc mới thành lập thì đôi khi đây là lợi thế duy nhất của họ. Do đó, họ không ngần ngại nâng cao mức lương lên để có thể có được người, dẫn đến một tình trạng thường được gọi là "Bong bóng lương". Chẳng hạn, một lập trình viên mới ra trường, đang đi làm với mức lương là 7 triệu một tháng, có thể sẽ nhận được một lời mời làm việc ở một công ty khác với mức lương 8-9 triệu. Sáu tháng sau, cũng là lập trình viên đó với mức lương 9 triệu, có thể nhận được một lời mời với mức lương 11-12 triệu. Và cứ như thế, mức lương được đẩy lên trong khi trình độ của lập trình viên thì không có thay đổi gì đáng kể. Đặc biệt là đối với các công ty khởi nghiệp có vốn đầu tư từ bên ngoài, họ sẵn sàng giành lợi thế tốc độ thông qua việc "đốt" tiền để tuyển được nhân sự.

Thông thường, mức thu nhập ở một ngành nào đó cao thì đó là tín hiệu tốt của ngành, điều này cũng đúng với ngành Công nghệ thông tin. Tuy nhiên, có một khía cạnh khác khá nghiêm trọng đó là các lập trình viên dễ tự mãn, ảo tưởng về trình độ của mình bởi vì mức lương, công việc mà mình đang có đang vượt trội so với mức trung bình của các ngành nghề thông thường khác. Từ đó, dẫn đến hiện tượng không coi trọng các giá trị đích thực của công việc, của nhân viên, của một chuyên gia. Thể hiện cụ thể nhất đó là sự gắn kết với công ty ngày càng giảm, tỉ lệ nhảy việc rất cao, gây tổn thất lớn cho bản thân các doanh nghiệp.

Thể hiện thứ hai đó là các lập trình viên không đầu tư để phát triển chuyên môn bản thân một cách bài bản và bền vững. Thay vào đó chỉ cần đếm "số năm kinh nghiệm" là đã có thể làm mưa làm gió trên thị trường. Nếu cứ tiếp tục tình trạng này, trình độ của lập trình viên Việt Nam không những không được nâng lên mà ngày càng chững lại, lạc hậu so với thế giới trong môi trường phát triển rất năng động của công nghệ.

Lỗi từ đâu?

Trước tiên là nhà trường, vì đã không đào tạo được các lập trình viên hiện đại có trình độ cao tương xứng với yêu cầu của ngành, và cũng không đào tạo được ý thức nghề nghiệp và tác phong làm việc chuyên nghiệp cho các lập trình viên. Sau đó là lỗi từ các doanh nghiệp với tư duy ăn xổi, mong muốn nhanh chóng lôi kéo nhân sự từ các công ty khác bằng cách nâng mức lương trong ngắn hạn. Vô hình trung đã tự mình làm khó mình bởi vì các công ty khác cũng có thể lôi kéo nhân viên của mình với cách làm tương tự. Quả bong bóng lương đã to nay lại càng ngày càng to hơn.

Trong tình hình của ngành công nghệ thông tin như hiện nay, lựa chọn làm việc trong ngành này là một định hướng tốt,

xét trên nhiều góc độ, từ mức thu nhập, điều kiện làm việc, khách hàng, sản phẩm và tương lai. Tôi muốn có lời khuyên cho các bạn trẻ đang là lập trình viên hoặc các bạn đang tìm hiểu để đi theo ngành này:

- Đối với những người đang làm việc trong ngành Công nghệ thông tin, đây là thời điểm mà các bạn có rất nhiều cơ hội và lợi thế. Cần tận dụng lợi thế này để nâng tầm của từng cá nhân, của công ty và của cả nền công nghiệp phần mềm Việt Nam thông qua việc liên tục học hỏi, nâng cao trình độ và tạo ra các sản phẩm có chất lượng. Đừng sớm hài lòng với chính mình vì điều kiện sống và làm việc hiện tại đã tốt, mà hãy trau dồi năng lực để tạo cho mình một nền tảng chắc chắn để đảm nhận được các công việc thú vị và ý nghĩa hơn.
- Đối với những bạn đang tìm hiểu để gia nhập vào đội ngũ các lập trình viên thì hãy an tâm mà lựa chọn. Với ảnh hưởng to lớn của Công nghệ thông tin lên các ngành nghề khác thì đây vẫn là một ngành phát triển rất nhanh và bền vững. Hãy tiếp cận một cách chuyên nghiệp, học lập trình chuyên nghiệp, làm lập trình chuyên nghiệp. Nhưng để ý, đừng quá ảo tưởng, mà hãy chăm lo cho năng lực của bản thân mình – đó mới là nền tảng phát triển lâu dài.

Nguyễn Khắc Nhật
Theo CodeGym

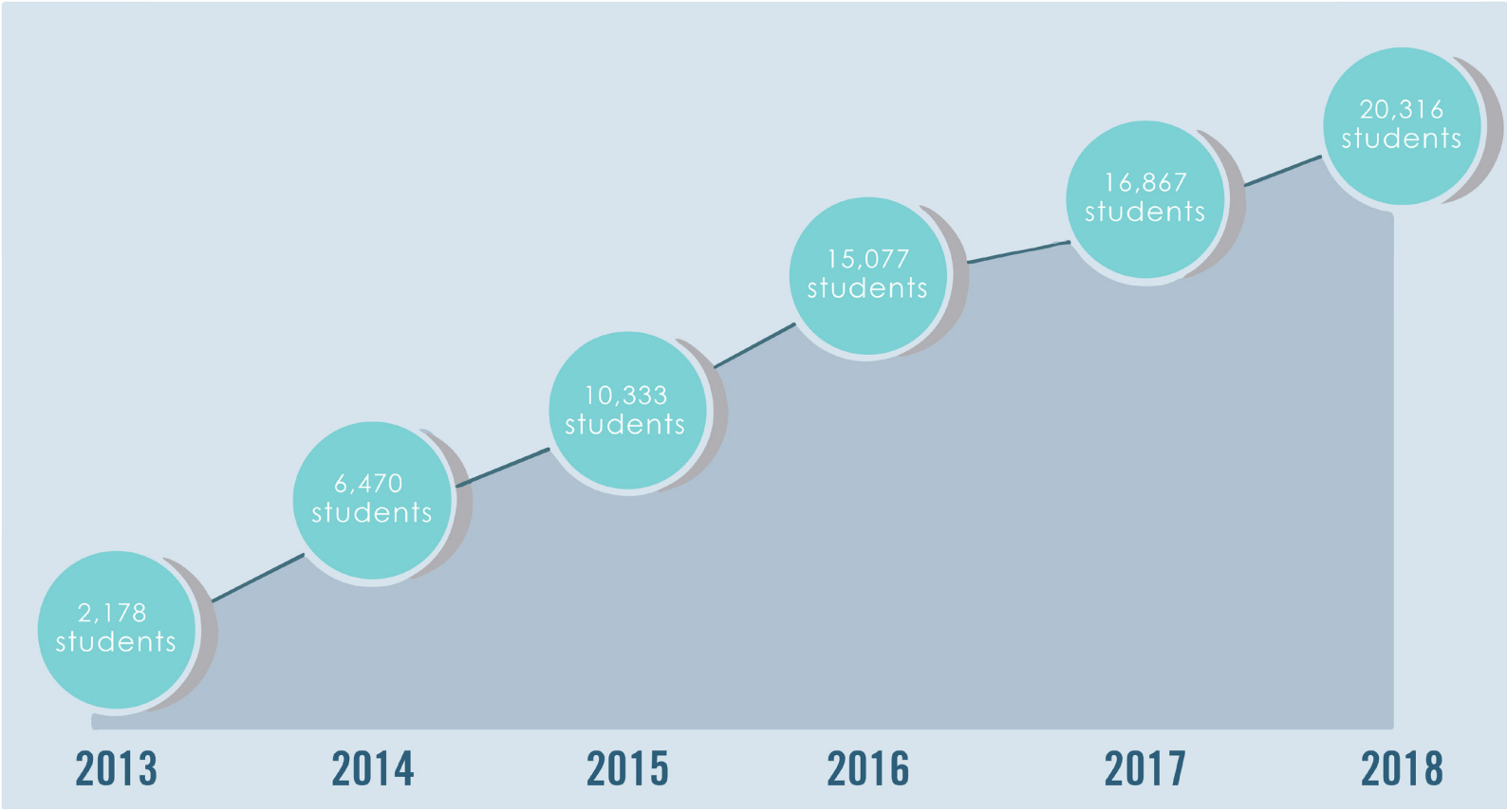
CODING BOOTCAMP

LỜI GIẢI CHO BÀI TOÁN NHÂN LỰC CÔNG NGHỆ THÔNG TIN

Coding Bootcamp là mô hình đào tạo lập trình hướng đến việc nâng cao hiệu quả học tập thông qua tổ chức môi trường học tập dưới dạng các "trại huấn luyện" cường độ cao, thời gian ngắn, thực chiến và thực tế. Đây là một mô hình đang ngày càng được ưa chuộng và rất phát triển ở các khu vực như Mỹ, Châu Âu, Nhật bản và nhiều quốc gia khác. Theo một báo cáo được Course Report thực hiện vào năm 2018, riêng ở Mỹ và Canada đã có 108 Coding Bootcamp, với hơn 20.000 học viên tốt nghiệp (*).

Coding Bootcamp là một mô hình đào tạo mới, khá khác biệt so với các mô hình đào tạo truyền thống ở nhiều khía cạnh khác nhau, chẳng hạn như mục đích, thời gian, chương trình, đánh giá, công nghệ, chi phí... Bảng sau đây liệt kê một số khác biệt chính giữa mô hình đào tạo truyền thống và Coding Bootcamp.

	Coding Bootcamp	Mô hình truyền thống
Mục đích	Cung cấp Lập trình viên thực chiến cho các doanh nghiệp	Đào tạo các kỹ sư CNTT có nền tảng
Thời gian	Ngắn. Từ 3-7 tháng	Dài. 2-3 năm đối với các trường nghề, 4-5 năm đối với cá trường Đại học
Chi phí	Thấp. Nhờ thời gian ngắn, học viên không chỉ phải đóng ít học phí hơn, mà còn tiết kiệm được rất nhiều chi phí sinh hoạt	Cao. Do thời gian dài, học viên không chỉ phải đóng học phí nhiều hơn mà còn mất rất nhiều chi phí cho sinh hoạt trong suốt thời gian học
Thời gian thu hồi vốn đầu tư cho việc học	Nhanh. Chỉ sau một thời gian ngắn theo học, học viên đã có thể tìm được việc làm và bắt đầu có thu nhập	Chậm. Sau một thời gian khá dài học viên mới có thể hoàn thành chương trình học và bắt đầu quá trình tìm việc.
Mức độ tập trung	Cao. Học viên dành toàn bộ thời gian cho việc học, gần như bỏ qua tất cả các hoạt động khác, do đó ít bị ảnh hưởng bởi các yếu tố bên ngoài, giúp cho hiệu quả học tập tăng lên	Thấp. Trong một khoảng thời gian dài, học viên không tập trung hoàn toàn thời gian cho việc học, thay vào đó thường có thêm rất nhiều hoạt động khác, đôi khi ảnh hưởng đến hiệu quả học tập
Nội dung chương trình	Tập trung vào ngôn ngữ lập trình, công cụ và kỹ năng để đáp ứng được ngay nhu cầu công việc ở các doanh nghiệp.Lựa chọn các nội dung mang tính "High impact" – tức là tạo được sự thay đổi lớn về mặt kỹ năng và kiến thức	Tập trung vào các kiến thức nền tảng, rộng. Thông thường, sau khi tốt nghiệp thì học viên sẽ dành thêm một khoảng thời gian ở doanh nghiệp để tìm hiểu về các kỹ năng thực tế.
Ứng dụng công nghệ	Ứng dụng nhiều công nghệ để nâng cao hiệu quả học tập và hiệu quả triển khai đào tạo	Có ứng dụng công nghệ trong một số mảng như nội dung học tập số, quản lý học viên
Định hướng nghề nghiệp	Lập trình viên chuyên nghiệp, nhà khởi nghiệp	Kỹ sư công nghệ thông tin chất lượng cao, nhà khởi nghiệp



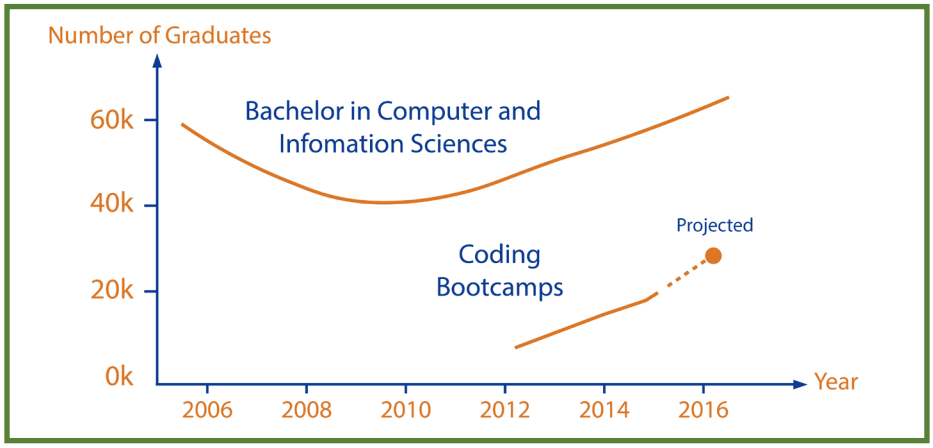
Số lượng học viên tốt nghiệp Coding Bootcamp tốt nghiệp ở Mỹ theo các năm
Nguồn: Course Report

Coding Bootcamp đang phát triển rất mạnh

Theo báo cáo của Course Report, kể từ năm 2013 cho đến 2019, số lượng học viên tốt nghiệp các Coding Bootcamp ở Mỹ đã tăng gấp 9 lần. Đây là một tỷ lệ tăng trưởng rất lớn, nếu biết rằng Coding Bootcamp đầu tiên mới xuất hiện lần đầu vào khoảng năm 2011.

Theo một nghiên cứu khác được thực hiện bởi Paul G. Allen vào năm 2015 (**), số lượng nhân lực CNTT thông tin tốt nghiệp Coding Bootcamp đã chiếm gần bằng 1/3 so với số lượng sinh viên tốt nghiệp trình độ cử nhân. Điều này có thể cho thấy rằng các Coding Bootcamp đang ngày càng đóng góp nhiều hơn cho ngành công nghiệp CNTT thông qua việc cung cấp nguồn nhân lực đông đảo vào kịp thời trước các nhu cầu ngày càng nhiều và đa dạng.

Sự thành công của Coding Bootcamp đã tạo được hiệu ứng tốt đến mức mô hình "Bootcamp" đã được mang vào áp dụng cho nhiều lĩnh vực khác. Bắt đầu xuất hiện các Bootcamp cho Data Science (Khoa học dữ liệu), UX Design (Thiết kế trải nghiệm người dùng), Digital Marketing (Tiếp thị số), Security (Bảo mật)...



Số lượng học viên tốt nghiệp Coding Bootcamp so với số lượng sinh viên tốt nghiệp bậc cử nhân theo từng năm
Nguồn: Nghiên cứu của Paul. G. Allen

Các đặc điểm của một CODING BOOTCAMP TỐT

Đặc điểm mang tính quyết định đến hiệu quả của một Coding Bootcamp chính là thiết kế của chương trình, bao gồm toàn bộ các quy trình, tài nguyên, con người, công nghệ và công cụ để triển khai hoạt động đào tạo. Một Coding Bootcamp tốt cần kết hợp được tất cả các yếu tố trên để tạo thành một chuỗi các trải nghiệm học tập hiệu quả dành cho học viên, tính từ thời điểm đầu tiên tham gia chương trình, cho đến khi tốt nghiệp, và cả sau đó.

Bài viết trên Firehose Project(***) có liệt kê một số đặc điểm để nhận diện một Coding Bootcamp tốt, bao gồm các yếu tố:

- Chất lượng của giảng viên và người hướng dẫn
- Đội ngũ sáng lập viên là người biết lập trình
- Học viên được lập trình ngay từ ngày đầu
- Hỗ trợ kỹ thuật toàn thời gian.

Coding Bootcamp mang đến CÁCH HỌC MỚI

Coding Bootcamp là một cách tiếp cận mới về phương pháp đào tạo, do đó đòi hỏi học viên có một cách học mới, không giống với các cách học vẫn thường thấy ở các mô hình truyền thống.

Ở Coding Bootcamp, học viên vất vả hơn rất nhiều, vì tham gia rất nhiều vào các hoạt động học tập. Đây chính là thể hiện rõ nhất của hình thức "active learning" (Học tích cực) – trong đó kết quả của việc học phụ thuộc rất nhiều vào sự tham gia, mức độ cam kết và nỗ lực của các học viên. Học viên ở Coding Bootcamp gần như không có thời gian để "nghỉ ngơi", mà chuyển từ hoạt động học tập này sang hoạt động học tập khác, nhằm tạo ra một sự tăng trưởng nhanh chóng về mặt kỹ năng và kiến thức trong thời gian ngắn.

Các hoạt động chính của học viên của các Coding Bootcamp bao gồm: Lập trình giải quyết các bài toán đã được thiết kế, học trên các hệ thống trực tuyến, trả lời các câu hỏi, tự tìm kiếm các kiến thức liên quan, thảo luận nhóm, hỏi, tham gia các cuộc thi lập trình, tham gia các phiên luyện code cường độ cáo, viết báo cáo...

Với cách học mới này, học viên sẽ gặp khá nhiều khó khăn, nhất là trong những thời gian đầu. Do đó, ở các Coding Bootcamp, thường có khác nhiều các tutor, mentor hỗ trợ toàn thời gian để giúp học viên vượt qua các thử thách.

CÁC KHÓ KHĂN đối với Coding Bootcamp

Thời gian ngắn vừa là một ưu điểm nhưng cũng mang lại khó khăn cho các Coding Bootcamp, chẳng hạn như nó không đủ để các học viên trẻ kịp hình thành một nền tảng về thái độ và văn hoá trưởng thành. Đối với những học viên đã trưởng thành (chẳng hạn như người đã đi làm các ngành nghề khác, các sinh viên năm cuối ở các trường Đại học) thì đây thường không phải là vấn đề, bởi vì họ đã được rèn luyện trước đó.

Chi phí cao – là cảm giác đối với những người vừa mới nhìn vào học phí trên từng tháng. Trong thực tế thì không phải như vậy, nếu tính theo toàn bộ chương trình học thì tổng chi phí dành cho Coding Bootcamp sẽ ít hơn nhiều so với việc học viên theo học các chương trình khác trong

vòng 2-3 năm hoặc nhiều hơn. Ngoài ra, nếu tính trên đơn vị thời gian là giờ làm việc (giờ được hỗ trợ) thì chi phí cũng khá rẻ, so với các hình thức đào tạo bán thời gian – học viên chỉ tham gia học một số giờ nhất định trong tuần. Tuy nhiên, lúc mới nghe mức học phí theo tháng thì thường sẽ khiến các ứng viên đắn đo.

Quy mô lớp nhỏ – bởi vì nếu quy mô lớp lớn thì không thể hỗ trợ tốt được đến từng học viên. Các Coding Bootcamp thường mở các lớp dưới dạng các nhóm nhỏ dưới 20 người, thậm chí là chỉ 3-5 người học cùng nhau. Quy mô lớp nhỏ sẽ khiến cho chi phí triển khai đào tạo (giảng viên, quản lý...) tăng lên.

Chiếm dụng mặt bằng lớn. Khác với các hình thức đào tạo bán thời gian, một phòng học có thể đáp ứng được cho nhiều lớp. Ở Coding Bootcamp, mỗi học viên tham gia toàn thời gian, do đó mỗi phòng học chỉ đáp ứng được cho một lớp học duy nhất, do đó chi phí cho địa điểm sẽ tăng lên. Đó cũng là lí do tại sao nhiều Coding Bootcamp lựa chọn hình thức xây dựng các co-working space để học viên tự do ngồi làm việc, chứ không ngồi cố định như các mô hình truyền thống.

Học viên cần chuẩn bị gì khi tham gia Coding Bootcamp?

Một số câu hỏi mà một học viên cần tìm hiểu trước khi quyết định tham gia vào một chương trình Coding Bootcamp:

- Đâu là mô hình phù hợp với mình? Coding Bootcamp hay là Đại học, Cao đẳng, học trực tuyến...
- Mình đã sẵn sàng để tham gia Coding Bootcamp chưa? Điều kiện thời gian, điều kiện tài chính, tính cam kết của cá nhân, nhu cầu việc làm...
- Mình nên học công nghệ nào? Web back-end, web front-end, web full-stack, mobile, Java, PHP, iOS, Android...
- Những doanh nghiệp nào sẽ phù hợp với mình?

Khi đã quyết định tham gia Coding Bootcamp, học viên hãy:

- Loại bỏ hoàn toàn các hoạt động khác có ảnh hưởng đến thời gian và sự tập trung cho việc học
- Dành tối thiểu 8-12 giờ học mỗi ngày
- Dành nhiều thời gian nhất có thể để luyện tập kỹ năng lập trình
- Nhanh chóng tạo ra các sản phẩm nhỏ để chứng minh tiến bộ về năng lực của bản thân
- Tích cực hỏi và tham gia các hoạt động học tập
- Đến các doanh nghiệp phần mềm để tìm hiểu về môi trường làm việc ở đó
- Để ý đến sức khỏe thể chất và sức khỏe tinh thần, đảm có đủ sức lực và động lực để hoàn thành các hoạt động hằng ngày
- Thường xuyên đánh giá lại bản thân, để biết được các tiến bộ và các khó khăn mà mình đang gặp phải
- Tìm kiếm sự hỗ trợ nhiều nhất có thể từ những người hướng dẫn để mất ít thời gian hơn khi gặp phải những nút thắt khó nhằn

Nguyễn Khắc Nhật

THẾ GIỚI CỦA NHỮNG LẬP TRÌNH VIÊN **ĐANG THAY ĐỔI**

Thế giới ngày càng khắc nghiệt...

20 năm, 10 năm trước, chuẩn mực của một LTV là gì? Dù là béo phì (LTV Mỹ) hay gầy còm (LTV Việt) thì cũng đều căng thẳng, đầu tóc rối bời, những cặp kính cận lớn và... xa lánh cộng đồng; họ đặt mình vào những căn phòng với thuốc lá, bia, chất kích thích và chỉ giao tiếp với nhau hoặc với máy tính. Chẳng khó để nhận ra một LTV trong đám đông.

Ngày nay, thật khó để nhận ra một LTV trong đám đông. Thật sự xin lỗi, nhưng có lẽ dấu hiệu duy nhất để nhận ra những người làm nghề lập trình là họ thường mang theo laptop. Vậy thôi. Những LTV ngày nay, quá giống một người bình thường với áo thun, quần jeans, đầu vuốt keo, đôi khi cùng khuyên tai hay quần tụt.

Công bằng mà nói, những LTV ngày nay đang có cuộc sống xã hội phức tạp hơn. 20 năm trước, LTV là những người đi tiên phong trong ngành CNTT với chỉ số IQ bắt buộc phải nằm trong top 10% nhân loại, họ luôn phải giải quyết những bài toán khó đòi hỏi nền tảng khoa học tốt; và họ tạo cho mình 1 đặc quyền: tách ra khỏi đời sống xã hội. Giờ đây, như xu thế của mọi ngành công nghiệp phát triển khác, việc lập trình trở nên đại chúng hơn, và LTV dần mất chất hơn. Họ bị kéo trở lại đời sống xã hội bình thường.

Và góc nhìn của LTV cũng buộc phải thay đổi. Cảm giác của LTV cũng buộc phải thay đổi. 20 năm trước, LTV vui mừng vì viết ra 3 dòng code giúp tiết kiệm được 1KB bộ nhớ sau 1 tuần trăn trở. Giờ đây, LTV vui mừng vì tạo ra sản phẩm giúp ích tới hàng ngàn người dùng. Hệ quy chiếu thay đổi, và thước đo cũng đã thay đổi. LTV thay vì đặt mình trong một không gian riêng, chiêm ngưỡng vẻ đẹp của những đoạn code về mặt khoa học thuần tuý; giờ họ buộc phải gắn mình với đời sống kinh doanh phức tạp hơn: Tiết kiệm 1KB bộ nhớ chẳng có ý nghĩa gì nếu nó không mang lại giá trị cho người dùng. 20 năm trước, những LTV được coi là tệ hại nếu không biết cách tiết kiệm thêm 10KB bộ nhớ. Ngày nay, những LTV được coi là thiếu đạo đức nếu viết ra những dòng code tuy tối ưu nhưng khiến đồng nghiệp khó hiểu. Tất nhiên,

ở đâu đó trên thế giới, hệ quy chiếu truyền thống vẫn tồn tại, nhưng nó đang ít dần đi.

Và rồi những LTV truyền thống cảm thấy thất vọng, họ thấy thật nực cười khi phải quan tâm tới những giá trị mang lại cho khách hàng, bực bội khi phải quan tâm tới lợi nhuận của doanh nghiệp. Không, tôi muốn làm thế này, vì nó chạy rất nhanh, vì nó là thử thách, vì tôi muốn giải quyết bài toán này... Họ bị stress khi bị lôi trở lại đời sống xã hội phức tạp, nơi những giá trị kinh doanh là thứ họ chưa bao giờ muốn biết. Tôi chỉ muốn lập trình thôi, trời ơi. – một LTV gào lên và ngay lập tức ông chủ của anh ta sẽ đáp lại: Anh bạn à, một bức ảnh đẹp phải được đo bằng view và like chứ không phải vì nó tuân theo tỉ lệ vàng. Thế đấy.

Nhưng thế giới cũng đang đẹp hơn...

Những LTV chân chính có cảm giác rằng LTV không còn là một nghề cao quý và đáng trân trọng khi mà công việc lập trình được bình dân hoá đến mức #kids_can_code và luôn bị cuốn theo giá trị kinh doanh của doanh nghiệp. Song không thể phủ nhận rằng, cuộc sống của họ đang tốt dần lên chính nhờ những điều đó. Họ biết thêm nhiều kiến thức, kỹ năng mới, hòa nhập hơn đồng nghiệp và xã hội. Họ có thời gian để tập gym, quan tâm tới thời trang, những show ca nhạc... thay vì chỉ vùi đầu trong những đoạn code và chất kích thích.

Nhưng không có nghĩa là những LTV chân chính muốn mất đi không gian riêng. Họ vẫn muốn và vẫn cần có những không gian để không quan tâm tới giá trị kinh doanh, rời xa những ồn ào hiện tại và quay trở lại niềm vui chỉ với những dòng code. Đấy là lý do hàng loạt những website như topcoder, projecteuler, hackerrank... ra đời cùng hoạt động Coderetreat, Code Kata... Và doanh nghiệp cũng ngày càng chú trọng đến những hoạt động như Hackathon nơi họ cố gắng tạo ra một không gian để nhân viên của mình được làm những LTV chân chính dù chỉ 1 vài lần trong năm.

Cũng giống như mọi ngành nghề khác, nhiều LTV không muốn nghề nghiệp của mình bị bình dân hoá; nhưng số nhiều và toàn xã hội lại hưởng lợi khi công việc này trở thành đại chúng với những con người cân bằng giữa công việc và đời sống xã hội. Nhưng cuộc chơi giờ là vậy, những LTV, hãy sống cân bằng, biết cách giao tiếp với đồng nghiệp, quan tâm tới những chỉ số kinh doanh và giá trị mang lại cho khách hàng. Nhưng đừng quên rằng, vẫn còn đó những nơi cho chúng ta không gian để chỉ quan tâm tới kỹ thuật, chỉ sống như một LTV chân chính. Hãy tìm lấy không gian để sống như một LTV chân chính nếu bạn thực sự là một LTV chân chính.

Nguyễn Văn Hiển

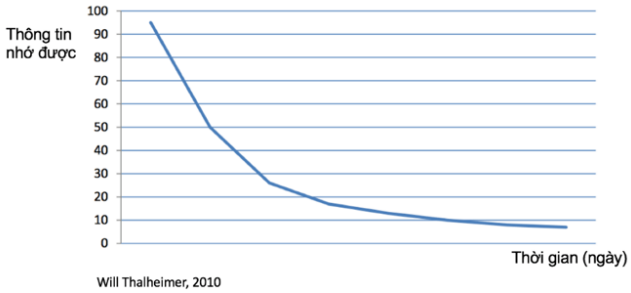
TOP 5

bí quyết học tập thượng hạng cho coder

Dù bạn là người đang học code hay đã có thâm niên coding vài năm thì những cuộc "cách mạng" công nghệ hiện nay vẫn có thể khiến bạn tụt hậu nhanh chóng nếu không giắt túi vài bí kíp tự học hiệu quả để luôn học những điều mới mẻ, và nâng cấp những năng lực sẵn có trong người.

Có những bí kíp thuần kinh nghiệm, có vài bí kíp nghe thì hay nhưng hên xui, và cũng có những bí kíp đã được khoa học kiểm chứng về tính hiệu quả. Bài này chia sẻ vài cách thức được đúc rút từ các nghiên cứu khoa học về việc học tập hiệu quả, hoặc của những cao thủ trong nghề viết mã.

1. Hãy luyện tập phân bổ, đừng học cuốn chiếu, và thật có chủ đích



Đồ thị lãng quên: Trung bình, chúng ta sẽ quên nhanh những gì học được

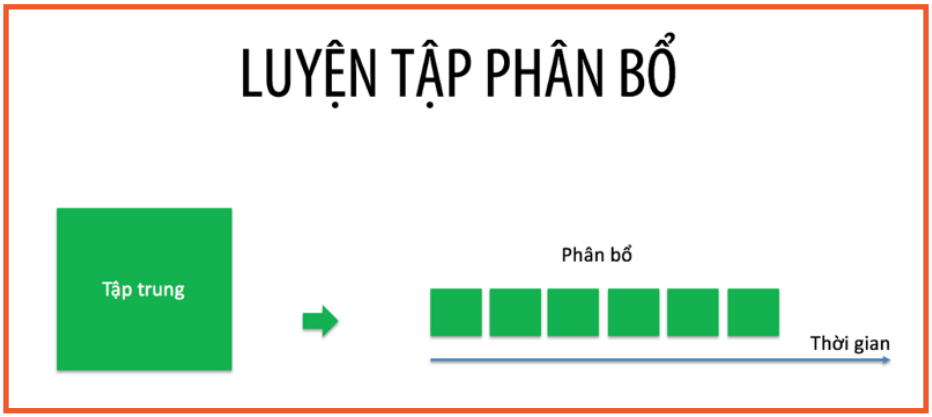
Theo một nghiên cứu của nhiều nhà khoa học đăng trên tập san "Psychological Science in the Public Interest ", bạn không nên học kiểu dồn ép và cuốn chiếu mà nên phân bổ kiến thức ra. Ví dụ, khi bạn học một ngôn ngữ mới (ví dụ Java), thì đừng vội đặt mục tiêu "làm chủ Java trong 7 ngày" rồi bỏ hết tất cả các việc khác để dồn 100% công lực vào học Java trong vòng 7 ngày. Điều đó nghe rất hấp dẫn nhưng không khả thi, bạn sẽ không thu

hoạch được nhiều sau 7 ngày. Và đặc biệt, nếu sau 7 ngày đó bạn coi như đã "xong việc" thì đảm bảo là bộ nhớ của bạn sẽ chẳng còn bao nhiêu sau một thời gian ngắn.

Thay vào đó, hãy rải việc học ra mỗi tuần một ít, có chủ đích lặp đi lặp lại để "luyện cơ" và khắc sâu kiến thức vào bộ nhớ trong đầu bạn.

Ví dụ

- Tuần 1 bạn học căn bản về ngôn ngữ Java, các khối điều khiển, cách tổ chức chương trình.
- Tuần 2 bạn có thể tập cách tổ chức một trang web đơn giản, vận dụng các kiến thức về cấu trúc chương trình, hàm/cấu trúc điều khiển trước đó.
- Tuần 3 bạn nâng cấp trang web đó dưới dạng kiến trúc MVC, có liên hệ hay dở với cách thức tổ chức chương trình kiểu cũ.



Luyện tập phân bổ, mỗi lúc một ít. Cứ thế cứ thế, kiến thức sẽ được xây dựng vững chắc từng khối từng khối một.

Ở đây có ba từ khoá: phân bổ, cuốn chiếu và chú tâm. Luyện tập phân bổ nghĩa là tách các kỹ năng ra để làm chủ từ từ, đừng dồn lại một lúc. Cuốn chiếu chính là cách học [không hiệu quả] được các sách dạng tutorial quảng cáo (thật đáng tiếc là một số trường học vẫn đang dạy lập trình cuốn chiếu kiểu này). Còn có chủ đích là cái mà cao thủ Peter Norvig đã đề cập (đọc thêm bài "Luyện code như luyện cơ"): " Điều quan trọng là bàn về phương pháp thực hành: không chỉ là việc lặp đi lặp lại đơn thuần, mà còn thử thách chính mình bằng những nhiệm vụ như vượt qua khả năng hiện tại của bản thân, cố gắng, phân tích hiệu xuất của mình trong và sau quá trình rèn luyện, và sửa chữa bất kỳ sai lầm nào. Cứ như vậy, lặp đi lặp lại."".

2. Hãy thư giãn, và ngủ đủ

Coder rất hay dán mắt vào màn hình trong nhiều giờ, kể cả khi làm việc cũng như khi chơi game. Điều này gia tăng sức ép lên não bộ, khiến nó mệt mỏi và khó hoạt động hiệu quả.

Theo nghiên cứu của nhà khoa học thần kinh John Medina ghi trong sách "Luật trí não", não bộ chúng ta hoạt động theo một chu kì 10 phút tập trung rồi sau đó là sao nhãng.

Mẹo hay

Hãy dùng đồng hồ bấm giờ và sử dụng phương pháp Pomodoro để áp dụng hài hòa cơ chế tập trung-thư giãn khi làm việc.

Pomodoro

(Pomodoro Technique) là 1 phương pháp quản lí thời gian để nâng cao tối đa sự tập trung trong công việc. Trong tiếng Italia Pomodoro có nghĩa là quả cà chua – Lý giải cho việc tại sao lại dùng 1 chiếc đồng hồ hình quả cà chua để theo dõi thời gian trong phương pháp này...

Theo hai chuyên gia khác về học tập và thần kinh học Gs. Barbara Oakley và Gs. Terence Sejnowski, chúng ta nên kết hợp tập trung và thư giãn để giúp não bộ làm việc hiệu quả. Khi bạn thư giãn, không có nghĩa là não ngừng hoạt động, mà nó hoạt động ở cơ chế "khuếch tán" (diffused). Nhiều ý tưởng hay, nhiều sáng tạo quan trọng được phát sinh trong lúc thư giãn ấy chứ không phải tập trung hết cỡ để học tập hay giải quyết vấn đề.

Lời khuyên cực kì quan trọng khác là chúng ta nên ngủ đủ. Lời khuyên này tầm thường như cân đường hộp sữa dễ bị bỏ qua. Các coder thường kháo nhau về trình độ của một tay nào đó "làm việc 3 ngày liền không cần ngủ", nghe rất "ngầu". Thực ra, nếu bạn muốn năng suất giảm sút gấp đôi, và rước bệnh vào thân thì cứ việc "OT" liên tục, "over night" liên tục. Chúc bạn may mắn.

3. Hãy đọc thật nhiều code

Bill Gates từng tâm sự, thời học code, cậu bé tò mò tọc mạch Bill Gates đã lục tung code của các chương trình máy tính sẵn có. Nhờ đó mà học cách viết code. Nhiều người cũng học code bằng cách đọc code của người khác, bao gồm cả người viết bài này.

Bạn được lợi gì từ việc đọc code? Thứ nhất là có thể biết được cách các cao thủ tổ chức mã lệnh thế nào, từ đó mà bắt chước được phần nào đấy. Thứ nhì là mở rộng khả năng "hiểu" code. Bản thân code là một văn bản. Mặc dù code viết ra để cho máy nó chạy, nhưng một "người viết code tốt là người viết ra để con người hiểu được", "còn viết code chỉ để máy hiểu được thì là code bình thường". Đọc hiểu được code cũng giúp bạn phần nào trong việc viết ra những đoạn code mà người khác đọc được. Tất nhiên, nếu là người mới học, hãy bắt đầu từ đọc những chương trình nhỏ, đừng vội xông ngay vào mã nguồn của Linux nếu bạn không muốn ngất trên bàn phím.

4. Hãy kiến tạo

Ý tôi là, hãy viết gì đó dùng được. Vui thôi cũng được. Học không gì hay bằng tự làm ra sản phẩm. Học mà làm, làm thì học.

Khi học Android chẳng hạn, một trong những cách học hay nhất là làm một cái app nào đó giải quyết vấn đề nào đó của chính mình và người xung quanh. Càng cụ thể và "nhỏ bé" càng tốt. Vì bạn sẽ có những "chiến thắng nhỏ" của riêng mình. Điều đó chính là liều doping cực mạnh để thúc đẩy bạn viết những chương trình lớn hơn, vươn lên những mục tiêu mới.

Hơn chục năm trước, tôi cũng từng làm một cái app Java (J2ME) có tên rất buồn cười là "Mama tổng quản", cài lên máy Sony Ericson của người thân để theo dõi chi tiêu khi đi chợ. Nghe ngớ ngẩn phải không? Nhưng tôi của dạo ấy thì vui lắm, vì có thể dùng những dòng code của mình vào những việc hữu ích hằng ngày

5. Hãy tự đánh giá

Tự đánh giá là một trong những cách học tốt nhất theo nghiên cứu đã dẫn ở bên trên. Nó vừa giúp ta khắc sâu kiến thức, vừa giúp ta tránh những hiểu nhầm. Đó vừa là cơ hội củng cố kiến thức, nhớ dai hơn, nhưng cũng là cơ hội để sửa sai, tránh tình trạng "ảo tưởng sức mạnh".

Có nhiều hình thức tự đánh giá. Bạn có thể dùng các bài tập, câu hỏi ôn tập, danh mục kiểm tra ở cuối chương và tự làm lấy rồi so kết quả, hoặc tự kiểm tra sản phẩm làm ra. Nếu tham gia một lớp học, hãy yêu cầu giảng viên cung

cấp các câu hỏi nhanh (quiz), rubrics, checklist. Đó sẽ là những công cụ tuyệt vời để bạn tự kiểm tra kiến thức của mình.

Đôi khi tự mình viết ra những tóm tắt và suy tư (reflection) về bài học, hoặc tự thuyết trình kiến thức học được cho người khác cũng là một cách tự kiểm tra, xem mình đã thực sự lĩnh hội được mấy phần kiến thức.

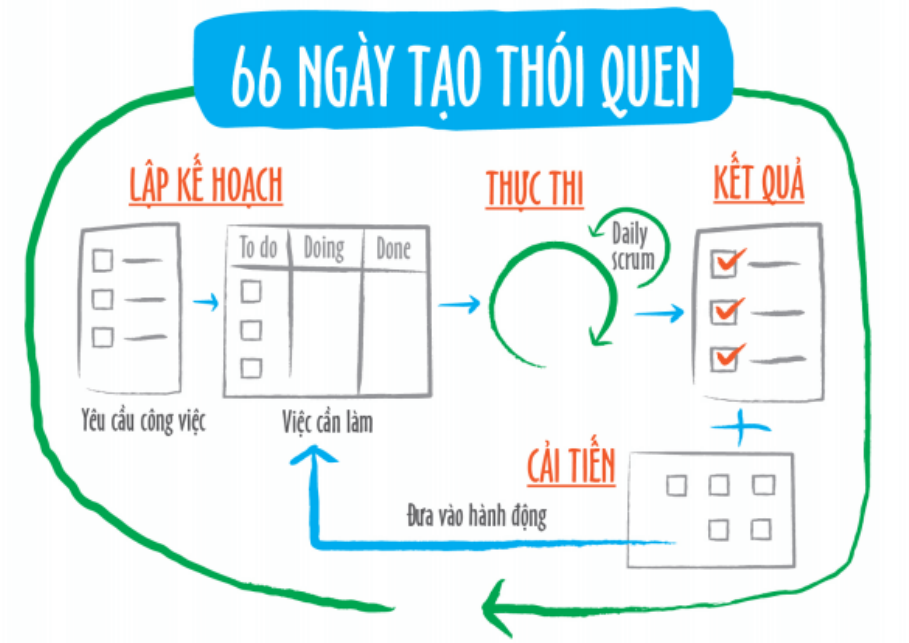
> *Tự học luôn là một hành trình đầy thú vị. Hy vọng năm chiến thuật trên đây giúp bạn đi nhanh hơn và thu hoạch tốt hơn trên hành trình ấy.*

Dương Trọng Tấn

SỨC MẠNH CỦA **THÁI ĐỘ** VÀ **THÓI QUEN**

> *"Niềm tin sẽ làm nên suy nghĩ của bạn, Suy nghĩ đó sẽ làm nên lời nói của bạn, Lời nói đó sẽ làm nên hành động của bạn, Hành động đó sẽ làm nên thói quen của bạn, Thói quen đó sẽ làm nên giá trị của bạn, Giá trị đó sẽ làm nên số phận của bạn."*

Mahatma Gandhi



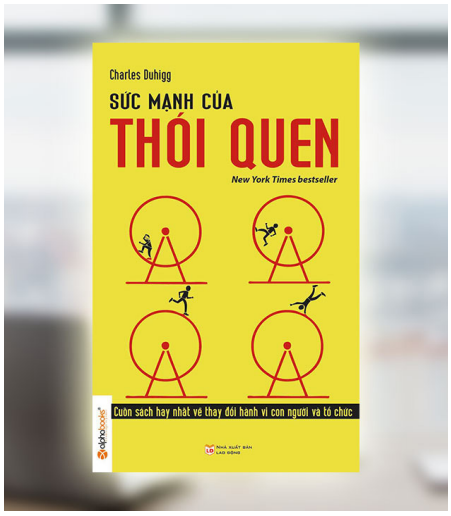
Hãy bắt đầu từ thái độ

Chúng ta có thể gặp những người ngoài 60 tuổi vẫn rất chịu khó học hỏi, những cụ 70 vẫn lọ mọ học máy tính để chat với con cháu ở xa. Họ là ví dụ của những người có mô thức phát triển (growth mindset, chữ của nhà tâm lý học Carol Dweck ở đại học Stanford). Những người này không để tuổi tác đè nặng, họ học hỏi không ngừng, thích chinh phục những thử thách, thấy khó tìm cách vượt qua, mỗi cơ hội làm việc được tận dụng để hoàn thiện chính mình, luôn coi các chỉ trích hướng vào mình như là cơ hội học tập, và vui mừng khi thấy sự thành công ở người khác. Đây là mẫu tư duy thường gặp ở những người thành công trong các lĩnh vực học thuật, kinh doanh, hay nghệ thuật

Ngược lại, lại có những người không thích các thử thách, thấy khó khăn là chùn bước, thấy phê phán là phản ứng tức thì, thấy việc làm là ngại, thấy người khác thành công thì ghen tị. Họ được xếp vào những người có mô thức đông cứng (fixed mindset). Sự khác biệt giữa hai mô thức nằm ở thái độ đối với các mục tiêu trong cuộc sống, trong cách thức phản hồi

với thử thách và thất bại, niềm tin về nỗ lực và chiến lược, cũng như thái độ với sự thành công của người khác.

Các nghiên cứu về não bộ gần đây cho thấy tính "mềm dẻo" của não người là rất lớn, và nó có thể thay đổi từ tấm bé cho tới lúc già. Số lượng tế bào thần kinh có thể không tăng, nhưng cơ cấu tổ chức não bộ với những kết nối phức tạp giữa các tế bào đó thì thay đổi luôn luôn. Ngay cả những công nghệ như Internet, Facebook, Google cũng khiến cho hoạt động của não bộ thay đổi đến ngạc nhiên; điều này được Nicholas Carr bàn kĩ trong cuốn "Trí tuệ giả tạo" bán chạy. Từ nghiên cứu về mô thức (Mindset) của Carol Dweck,chúng ta có thể rút ra được kết luận hết sức quan trọng là con người hoàn toàn có thể phát triển trong suốt cuộc đời. Cách phân biệt mô thức đông cứng và mô thức phát triển đặt nền móng khoa học vững chắc cho niềm tin về việc học tập suốt đời và sự làm chủ cuộc đời cho những người làm giáo dục-đào tạo trên toàn thế giới. Nó cũng giúp ta chọn lựa một thái độ tích cực và chủ động đối với các nỗ lực vươn tới thành công và hạnh phúc.



Thói quen hoạt động như thế nào

Có thái độ thôi chưa đủ, chúng ta phải hành động, và còn phải tạo lập các thói quen tốt. Một nghiên cứu ở Đại học Duke cho thấy 40% hoạt động của chúng ta hằng ngày thuần túy là do thói quen chứ không phải do suy nghĩ thấu đáo. Con số đó cho thấy tầm ảnh hưởng rất lớn của những việc lặp đi lặp lại theo kiểu "phản xạ có điều kiện" này.

Khi đã có thói quen, chúng ta lặp lại các phản xạ trước các tính huống của cuộc sống mà không phải suy nghĩ nhiều. Điều đó có nghĩa là nếu thói quen tốt và được rèn luyện tốt, thì trong phần lớn các tình huống lặp đi lặp lại ấy, chúng ta đang sử dụng cách làm tốt, mang lại kết quả tốt, mà không phí sức. Ngược lại, nếu thói quen xấu, nó sẽ vô tình chung ảnh hưởng tới kết quả mà chúng ta cũng không để ý, tới khi sự việc đã muộn rồi.

Tác giả Charles Duhigg của cuốn sách bán chạy "Sức mạnh của thói quen" dẫn các nghiên cứu khoa học cho biết, sở dĩ chúng ta hình thành các thói quen vì não chúng ta muốn được giảm tải. Ví dụ như khi chúng ta mới học lái xe, đầu óc sẽ suy tính rất mệt mỏi, đạp ga thế nào, phanh thế nào, tay giữ vô lăng chặt như thế có được không, v.v. Nhưng qua thời gian luyện tập, chúng ta hầu như không còn phải nghĩ nhiều về những việc này nữa, chúng ta hoạt động theo thói quen, và não thì rảnh để quan sát đường, và suy nghĩ về những việc khác khi lái xe.

Thói quen được hình thành theo một chu trình ba bước: Bước Kích hoạt có tác dụng khởi động não bộ vào trạng thái tự động và lựa chọn thói quen để sử dụng; bước Hoạt động (có thể là hoạt động thể chất, tinh thần, cảm xúc); và cuối cùng là Phần thưởng cho những hành động vừa trải qua (phần thưởng này có thể là sự khoan khoái, hoặc được ngợi khen, hoặc bằng những vật có giá trị). Não bộ sẽ đánh giá phần thưởng để xem xét khả năng lưu giữ lại hoạt động

đó để lặp lại khi có kích hoạt tương tự. Sự liên hệ Kích hoạt với Phần thưởng sẽ giúp não bộ phát triển cảm giác kỳ vọng, từ đó dẫn đến hình thành thói quen. Khi được lặp đi lặp lại, chu trình "Kích hoạt, Hoạt động, Phần thưởng" sẽ được tự động hóa, và chúng ta có thói quen.

Bạn không thể bỏ thói quen xấu, nhưng có thể tạo thói quen mới. Nguyên tắc vàng để thay đổi thói quen là giữ nguyên phần Kích hoạt và Phần thưởng, chỉ thay đổi phần Hành động. Lặp đi lặp lại, ta sẽ có thói quen mới. Đây là cách thay đổi thói quen phổ biến và rất hữu hiệu.

Cần 66 ngày để có một thói quen mới

Một nghiên cứu gần đây đăng trên Tạp chí Tâm lí học xã hội Châu Âu cho thấy, trung bình bạn cần 66 ngày để hình thành một thói quen. Trong 66 ngày đó bạn sẽ phải làm đi làm lại điều bạn muốn nó trở thành cơ hữu với con người bạn, sẽ thành "phản xạ" ngay cả khi bạn không còn nghĩ gì về nó nữa. Dĩ nhiên 66 là con số trung bình, con số cụ thể sẽ phụ thuộc vào nhiều yếu tố khác như tính chất của thói quen bạn định hình thành, cá tính của bạn, và cả điều kiện mà bạn đang có. Những con số này cho thấy, để tạo lập thói quen, bạn cần một sự kiên trì đáng kể. Đó là lí do vì sao các chuyên gia về thói quen khuyên chúng ta cần tạo lập sự liên kết chặt chẽ giữa phần Kích hoạt với Phần thưởng để liên tục duy trì động lực khi thực hành thay đổi thói quen.

Hình thành thói quen làm việc hiệu quả dựa trên Scrum

Một trong điểm mạnh cơ bản nhất của Scrum, một phương pháp Agile phổ biến nhất, chính là giúp bạn có một thói quen làm việc tốt: tính toán kĩ khi bắt đầu công việc, lập kế hoạch khả thi và linh hoạt, kiểm soát công việc liên tục, đánh giá cẩn thận kết quả đạt được và cải tiến liên tục. Các bước đó được sắp đặt và kết hợp logic với nhau, lặp đi lặp lại để sớm hình thành nếp nghĩ, nếp làm việc một cách khoa học và hiệu quả. Scrum không chỉ giúp bạn sớm nhìn ra hiệu quả công việc, vui sướng ngay trong khi làm việc, mà còn duy trì một thói quen tốt.

Tác giả của Scrum, tiến sĩ Jeff Sutherland đã từng viết trên blog cá nhân và trong cuốn sách "Scrum: Nghệ thuật làm được gấp đôi chỉ trong một nửa thời gian" về những câu chuyện thành công trong sử dụng tư duy Scrum vào mọi mặt của đời sống, từ việc dạy học ở trường phổ thông, đến việc marketing, hay tổ chức công việc hằng ngày ở nhà thờ. Chúng ta gọi thói quen này là Scrum Life, để chỉ việc áp dụng tư duy Scrum vào hình thành thói quen làm việc hằng ngày cho thật tốt.

Hãy bắt đầu Scrum Life đơn giản như thế này

1. Chúng ta sẽ khởi đầu tuần làm việc bằng việc lập kế hoạch, gồm 2 bước: xác định việc cần làm, và cách để hiện thực hóa việc cần làm.

2. Xong rồi ta cập nhật các công việc đó lên Bảng công việc (kanban board), bắt đầu làm những việc có độ ưu tiên cao hơn, dần dần cho tới hết. Mỗi ngày ta thực hiện 15 phút Daily Scrum để tự theo dõi tiến độ và thích ứng với các tình huống thay đổi.

3. Cuối tuần ta rà soát lại xem đã làm việc gì, so với dự kiến trong kế hoạch đầu tuần thì hoàn thành được bao nhiêu phần trăm, so với tuần trước thì thế nào?

4. Cuối cùng, ta suy nghĩ về cách làm việc , có ổn không, cần cải tiến gì không. Hãy cố gắng rút ra ít nhất một điều cải tiến, để tuần sau làm tốt hơn. Lưu ý đây là cải tiến cách làm; ví dụ, nếu tuần rồi ta viết thư điện tử cho sếp mà quên không đính kèm file, dẫn đến sếp bực mình, và đây là lỗi lặp lại lần thứ 3 rồi, thì ta có thể kĩ đến cách thức viết thư mới (ví dụ: đảo ngược quy trình viết thư: Đính kèm đầu tiên, rồi mới viết tiêu đề, rồi viết nội dung, cuối cùng là đọc lại và thêm phần To và gửi cho sếp).

Cách làm này áp dụng cho 1 tuần làm việc hăng say và duy trì sự hiệu quả liên tục. Nhưng bạn cũng có thể áp dụng tư duy ấy cho mỗi ngày làm việc với cấu trúc tương tự.

Trước khi bắt đầu thực hiện thói quen này, hãy nghĩ về mối liên hệ Kích hoạt – Phần thưởng. Phần thưởng cho việc hình thành thói quen này là gì? Bạn hãy nghĩ về những mối bận tâm thường trực khi làm việc: Bạn bắt đầu và kết thúc công việc trong tuần thế nào? Năng suất ra sao? Có hiệu quả không? Có thành tựu không? Có thấy vui sướng khi làm việc không? Và lưu ý, hãy tự thưởng cho mình, hoặc tìm kiếm các phần thưởng từ bên ngoài (từ Sếp, từ gia đình, bạn bè ...) khi bạn hoàn thành tốt công việc. Nó không chỉ là cơ chế duy trì động lực tạo thói quen, mà còn là sự tận hưởng cuộc sống.

Nếu bạn thực hành Scrum được 2 ngày và thấy hơi gò bó, thì xin chúc mừng bạn, bạn đang trong quá trình hình thành một thói quen mới. Nhưng hãy nhớ, bạn còn khoảng 64 ngày nữa. Cứ lặp đi lặp lại sẽ hình thành một nhịp làm việc và sinh hoạt đều đặn, dần trở nên tự nhiên và phát huy tác dụng. Hãy thử vài tuần đi, bạn sẽ thấy rất nhiều điều bất ngờ đấy.

Bạn có biết: Scrum từng được đề nghị trao giải Nobel về quản trị?

Đó chính là giải (không có thật) được trao bởi một cây viết quen thuộc trên tạp chí nổi tiếng Forbes, Steve Denning, cho hai ông Ken Schwaber và Jeff Sutherland vì đã có công phát minh ra phương pháp tổ chức công việc nổi tiếng Scrum làm thay đổi thế giới phần mềm trong hơn một thập kỷ qua.

Chuyện giải Nobel là do ông Denning "bịa" cho vui, nhưng để nhấn mạnh sự hiệu quả nổi bật mà Scrum có thể mang lại cho các nhóm làm việc khắp thế giới.

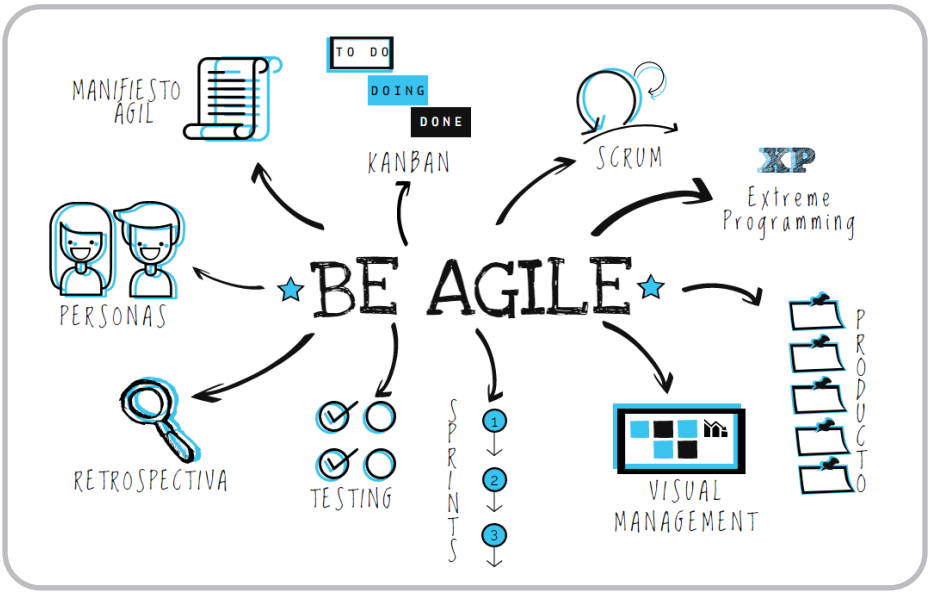
Trong một bài viết năm 2011, Denning tóm tắt 10 đặc điểm của Scrum (có thể không giống cách nói của chính các tác giả của Scrum):

1. Tổ chức công việc theo các chu trình ngắn (gọi là phân đoạn)
2. Khi nhóm làm việc của họ trong các chu trình ngắn này, cấp quản lí không can thiệp (tức nhóm được trao quyền tối đa)
3. Nhóm báo cáo trực tiếp cho khách hàng, không phải cho nhà quản lí
4. Nhóm ước tính thời gian để hoàn thành công việc
5. Nhóm quyết định cách hoàn thành công việc trong phân đoạn

6. Nhóm đo lường hiệu suất làm việc của chính mình
7. Nhóm đo lường hiệu suất của chính mình
8. Xác định mục tiêu công việc trước khi mỗi chu kỳ bắt đầu
9. Xác định mục tiêu công việc thông qua các câu chuyện của người dùng
10. Loại bỏ các trở ngại một cách có hệ thống

Sưu tầm

THẾ GIỚI AGILE NGÀY CÀNG PHỨC TẠP



Agile đang bị bias – thành kiến: "kẻ thích trở nên cuồng, người không thích trở thành căm ghét?"

Agile là gì (và không là gì)?

Tại chính thời điểm này, định nghĩa về Agile hoàn toàn không rõ ràng. Quay lại lịch sử, thuật ngữ này lần đầu tiên được giới thiệu là "Agile Software Development", sau một cuộc hội thảo giữa 17 người là tác giả của một số phương pháp phát triển phần mềm hiện đại (có tên và không tên), 2001. Họ chia sẻ chung một số phương pháp, nguyên lý... và Manifesto for "Agile Software Development" ra đời với 4 điều trong Tuyên ngôn và 12 nguyên lý.

Vậy là Agile Software Development có định nghĩa từ thời điểm đó, nhưng không bị giới hạn. Sau này có một số phương pháp nữa ra đời, dựa trên những nguyên lý trên, đồng thời khái niệm Agile cũng dần được thay thế cho Agile Software Development và lan dần sang một số lĩnh vực khác. Định nghĩa này chưa bao giờ được "nâng cấp version" một cách chính thức bởi những tác giả cũ cũng như mới nên nó trở nên không rõ ràng.

Agile thế nào (và không thế nào)?

Nhìn lại 4 điều trong Tuyên ngôn và 12 nguyên lý, dễ nhận thấy là chúng khá chung và đúng. Điều này dễ hiểu vì tìm kiếm điểm chung của nhiều phương pháp cũng như đồng thuận của 17 người, đại diện cho 17 tư duy, quan điểm thì khó mà cụ thể được.

Thời điểm đó, Agile là tập hợp của các phương pháp phát triển phần mềm dựa trên sự kết hợp của phương pháp lặp và tăng trưởng. Lưu ý, bài viết đang nói về phần "thể hiện" của các phương pháp, không

phải "tư duy" (nguyên lý) phía sau.

Nhưng sau này, dựa trên nguyên lý, nhiều thứ khác được coi là Agile với nhưng thể hiện không giống như ban đầu. Người ta bắt đầu nói về no planning, no estimation, no Sprint,... đủ thứ. Một ví dụ là Kanban, nó chỉ có thể hiện phần tăng trưởng, không rõ ràng đề cập tới lặp nhưng vẫn nằm trong Agile.

Agile có gì (và không có gì)?

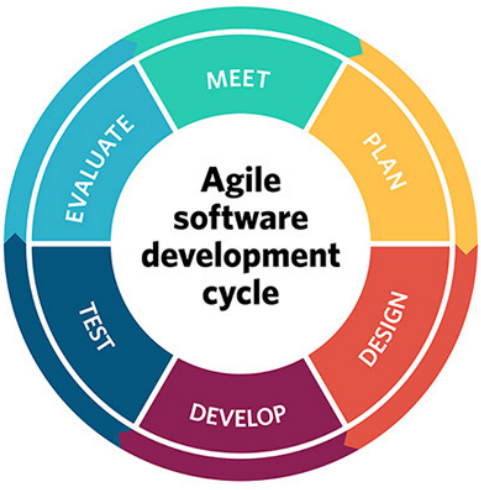
Lưu ý rằng Agile Software Development cũng phải dựa trên một số nguyên tắc về quản lý, cách tổ chức nói chung nhưng những thứ này không được định nghĩa rõ ràng. Điều này dẫn đến một số hiểu lầm khá tệ. Ví dụ như "self-organized team" và "cross-functional team", Agile sử dụng mà không sở hữu; vậy nên không thể hiểu chỉ Agile mới xây dựng những team như vậy. Hay như mô hình về team stage của Tuckman cũng không thuộc về Agile. Kỹ thuật 5 Whys, Kaizen... cũng vậy. Đó là những công cụ để nhóm hay người quản lý nhóm thực hiện tiến hoá quy trình.

Tương tự, DevOps ra đời sau, và "có liên quan" tới Agile. Nhưng ngày nay, cách hiểu Agile quá lớn nên DevOps được hiểu như là "nằm trong". Holacracy?

Trách nhiệm có lẽ thuộc về những nhà tư vấn, đào tạo không phân biệt rõ, gây ra những hiểu nhầm rằng "Agile là con đường duy nhất để đạt được những điều này".

Agile hướng tới gì (và không hướng tới gì)?

Agile hướng tới giá trị (value oriented), không hướng quy trình (process oriented). Quy trình trong Agile cũng buộc phải thể hiện được giá trị mới có lý do để tồn tại. Để tìm được giá trị, bắt buộc phải trả lời câu hỏi "tại sao (why)?". Để tìm được quy trình, câu hỏi là "thế nào (how)?". Câu hỏi why khó hơn rất, rất nhiều câu hỏi how. Và thông thường, để trả lời được câu hỏi why, chúng ta cần dựa trên how đang tồn tại.



Rắc rối ở đây, các nhà tư vấn, huấn luyện hoặc quản lý mong muốn đưa how vào trước; và kỳ vọng thời gian sau tổ chức sẽ tự trả lời được câu hỏi why. Công bằng mà nói, cách làm này không sai, nhưng nếu why không dần được tổ chức nhận ra, họ vẫn cứ loay hoay trong how. Và vô tình, Agile trở thành process oriented. Dù vậy, các "process" trong các phương pháp Agile vẫn khá tinh giản (so với các mô hình phát triển phần mềm khác), nên các team đã có quy trình trước đó khi chuyển sang Agile vẫn thấy bình thường. Điều tệ hại là, hầu hết các team ngày nay đều không áp dụng một phương pháp nào trước đó cho tới khi thực hành Agile, họ bỗng thấy Agile "cồng kềnh" và trở nên căm ghét.

> ***Ví dụ***
>
> Team trong Scrum là self-organized, thật khó tin là đưa 5, 7 con người vào một team và hy vọng self-organized được nên vẫn cần có một số quy tắc, quy trình cơ bản ban đầu; nếu team không có sự tiến hoá, sẽ trở thành process oriented.

Sau phong trào "doing Agile", giờ là phong trào "being Agile" – dù không dễ, nhưng có thể nhận thấy thiết thực của nó. Nhưng "go Agile" thì khó có thể biết trông nó sẽ thế nào.

Agile phù hợp với gì (và không phù hợp với gì)?

Nên nhớ, Agile ra đời với "Agile Software Development" tức là để giải quyết bài toán của phát triển phần mềm. Scrum nói rõ nó phù hợp với các dự án/sản phẩm nằm trong miền "phức hợp và phức tạp tương đối" về yêu cầu và kỹ thuật – tức là các dự án/sản phẩm đơn giản (đã rõ ràng) hoặc quá phức tạp (VD: R&D, công nghệ quá cao…) cần được xem xét. Thế nào là đơn giản, phức tạp… phụ thuộc nhiều yếu tố: con người, trình độ, công cụ,… nên (ví dụ) Scrum có thể phù hợp với team này ở sản phẩm này nhưng không phù hợp với team khác ở chính sản phẩm đó.

Agile dựa trên giả định về việc "chào đón sự thay đổi" (welcome change, Scrum: adapt to change, XP: embrace change); tức là, mọi cá nhân trong tổ chức đều thống nhất rằng "thay đổi là điều được chấp nhận". Tổ chức không đồng tình với sự thay đổi, không "chào đón" mà áp dụng Agile là sai từ gốc.

Hầu hết các phương pháp Agile đều dựa trên giả định "cá nhân có động lực làm việc" nên các tổ chức còn loay hoay trong việc này cũng nên cẩn trọng.

Nếu nhìn lại các câu hỏi đã được trả lời trên, Agile có thể giải quyết rất nhiều bài toán khác, không chỉ với việc phát triển phần mềm (trong Agile Y tôi còn nói tới cá nhân). Cùng với sự thống trị của các công ty công nghệ (mà lõi là sản phẩm phần mềm), Agile lan rộng sang những lĩnh vực khác, "nuốt chửng cả thế giới".

Lưu ý

- *Vì vậy Agile càng không rõ ràng.*
- *Các lĩnh vực khác không có chỉ dấu rõ ràng về hiệu quả vượt trội của Agile như trong phát triển phần mềm – nơi Agile ra đời do trải qua sự khủng hoảng về phương pháp. Hãy cẩn trọng.*

Agile trở thành "thuốc chữa bách bệnh"

Agile thực sử trở nên không rõ ràng. Cách hiểu gần nhất (của tôi) có thể là "Agile là công cụ (mindset, toolset) để tăng tính linh hoạt (agility) của tổ chức, qua đó thích ứng tốt với thay đổi (hoặc dẫn dắt thay đổi)".

Và cho đến giờ này, đây có vẻ là "bệnh" rõ nhất của các doanh nghiệp; và với hổ lốn những thứ được gắn tag Agile, Agile có vẻ được kỳ vọng quá lớn để chữa bách bệnh. Và khi không chữa được, đương nhiên Agile bị đổ lỗi.

Như trên, trách nhiệm nằm ở những nhà tư vấn, đào tạo Agile trong việc rạch ròi Agile là gì, và Agile trong phạm vi anh cung cấp dịch vụ là gì, nó thế nào, hướng tới đâu. Bằng cách vơ vét tất cả mọi thứ vào Agile cùng sự quảng cáo quá đà, gây ra sự ảo tưởng về sức mạnh từ tổ chức.

Agile trở thành "đạo"

Agile dự trên pull-system, giả định "cá nhân có động lực làm việc", điều này đi ngược với phần lớn cách tư duy về quản lý hiện có. Dù có hệ thống lý thuyết chặt chẽ đứng sau, Agile vẫn cần niềm tin để thực hiện trong tổ chức. Khi phương pháp cần tới "niềm tin" là điểm xuất phát thì nó không khác gì "đạo" – sẽ có kẻ sùng đạo và phản đạo.

> Có fan và anti-fan là điều dễ hiểu.

Scrum đã làm tốt gì (và không làm tốt gì)?

Scrum thực sự tuyệt vời. Nó là một mô hình tuyệt vời với đủ sự tinh gọn, hiệu quả cho phép tổ chức "nhận diện vấn đề". Đây là điều siêu quan trọng để phát triển sản phẩm cũng như phát triển tổ chức.

Scrum không làm việc giải quyết vấn đề. Vậy nên nếu nhóm không giải quyết những vấn đề được nhận diện, Scrum là đống rác cạnh hàng ăn. Nhưng có phương pháp nào giải quyết vấn đề không?

Scrum làm tốt trong việc ghi chú "Scrum dễ hiểu, khó làm chủ".

Scrum (đúng hơn là các tổ chức cấp chứng chỉ Scrum (Master, Developer, PO) làm không tốt trong việc đào tạo. Khá dễ dãi. Có 2 lỗ hổng lớn, khiến các ScrumMaster cầm chứng chỉ và nếu không chịu học hỏi thêm sẽ mắc phải:

1. Không trả lời được rõ ràng các câu hỏi trên. Cái gì thuộc Scrum, cái gì không? Dẫn đến ngộ nhận trong lý thuyết lẫn thực hành.
2. ScrumMaster không yêu cầu kiến thức về phát triển phần mềm. Về lý thuyết, điều này không sai; song thực tế thật khó để tham gia vào nhóm phát triển phần mềm mà không hiểu gì về cách nhóm thực hiện.

Từ đây dẫn tới những hệ luỵ vô cùng phức tạp. Chuyển giao cái gì? Technical debt là gì? Simple design là gì? Xử lý thế nào? Tài liệu thế nào là đủ?...

ScrumMaster yếu về chuyên môn (ngành) và non tay dẫn dắt nhóm có trình độ cao hơn mình sẽ là thảm hoạ. Đội bóng cần HLV tương xứng. Vấn đề là, tốc độ phát triển của thành viên Nhóm phát triển thường nhanh hơn (và khởi đầu tốt hơn) những ScrumMaster. Vậy nên nhiều nhóm phát triển bắt đầu chán ghét Scrum.

Trách nhiệm này thuộc một phần vào các tổ chức cấp chứng chỉ Scrum, phần lớn thuộc về các ScrumMaster khi không thể hiện được vai trò tương ứng, và cả các nhà đào tạo – không trang bị đủ hệ thống lý thuyết và thực hành.

Trách nhiệm của những nhà đào tạo, tư vấn?

Đừng trách sự tồn tại của những nhà đào tạo, tư vấn. Đừng trách những quảng cáo, khóa học. Đó là nhu cầu tất yếu, và thiết thực, bởi họ giúp rút ngắn con đường tổ chức tiếp cận Agile (nếu muốn). Nếu họ làm việc trên nguyên tắc cung cấp đúng giá trị và đạo đức.

Nhưng, đến lúc này, những nhà đào tạo, tư vấn cần phải có câu trả lời rõ ràng cho mình, cho khách hàng rằng với anh, Agile là gì (và không là gì), Agile (trong phạm vi đó) mang lại giá trị gì và lấy đi gì...?

Giống như bác sỹ, cần rõ ràng rằng những thành phần này là gì, tổ hợp lại được gọi tên gì, chữa bệnh gì, chống chỉ định gì, tác dụng phụ gì...?

Tuyệt nhiên không thể mông lung như Agile làm nhóm hạnh phúc hơn, Agile giúp tăng năng suất, Agile giúp tăng tốc độ hoàn thành dự án... – những điều Agile nguyên thuỷ không hề đề cập.

Nguyễn Văn Hiển

Tái cấu trúc mã nguồn

Khái niệm

Mỗi người có một khái niệm tái cấu trúc mã nguồn (code refactoring) khác nhau, và khi chuyển ngữ sang tiếng việt, thì việc tìm một thuật ngữ chính xác càng khó hơn. Ở đây tôi xin chuyển nghĩa từ refactoring thành tái cấu trúc và chọn định nghĩa của *Martin Fowler:*

> *Tái cấu trúc là thay đổi ở cấu trúc bên trong mà không làm thay đổi hành vi với bên ngoài của hệ thống*

Tái cấu trúc là một quá trình cơ học, hình thức và trong nhiều trường hợp rất đơn giản để làm việc với mã của hệ thống đã tồn tại để chúng trở nên "tốt hơn". Khái niệm "tốt hơn" là một khái niệm mang tính chủ quan, và không có nghĩa là luôn làm ứng dụng chạy nhanh hơn mà thường được hiểu là theo các kỹ thuật hướng đối tượng, tăng an toàn kiểu dữ liệu, cải thiện hiệu suất, dễ đọc, dễ bảo trì và mở rộng.

Hiệu quả của tái cấu trúc

Sản xuất phần mềm sẽ không hiệu quả nếu như bạn không thể theo kịp thay đổi của thế giới. Nếu như chúng ta chỉ sản xuất ra các phần mềm trong một vài ngày thì đơn giản hơn rất nhiều. Nhưng trong thế giới này chúng ta có rất nhiều đối thủ cạnh tranh. Nên nếu bạn không tái cấu trúc phần mềm của mình, khi đối thủ có một số tính năng hữu ích mới mà bạn không cập nhật thì sản phẩm của bạn nhanh chóng bị lạc hậu. Bởi thế là một lập trình viên bạn phải đón nhận và hành động một cách thích hợp với những thay đổi. Và khi thực hiện tái cấu trúc mã là bạn đang làm điều đó.

Cải thiện thiết kế

Nếu không áp dụng tái cấu trúc khi phát triển ứng dụng, thì thiết kế sẽ ngày càng tồi đi. Vì khi phát triển ứng dụng thì ta sẽ ưu tiên cho các mục tiêu ngắn hạn (đặc biệt khi áp dụng các quy trình phát triển linh hoạt), nên mã ngày càng mất đi cấu trúc. Một trong những tên của vấn đề này gọi là technical debt (nợ kỹ thuật). Khi xảy ra vấn đề thì rất khó để quản lý và dễ bị tổn thương .Thế nên việc áp dụng tái cấu trúc sẽ giúp cho mã giữ được thiết kế tốt hơn là ưu điểm quan trọng.

Mã dễ đọc hơn

Khi lập trình là chúng ta đang giao tiếp với máy tính để yêu cầu chúng làm điều mình muốn. Nhưng còn có người khác tham gia vào quá trình này là các lập trình viên khác hay chính chúng ta trong tương lai. Chúng ta biết khi lập trình thường sẽ có người phải đọc để kiểm tra xem có vấn đề với mã đó không hoặc để mở rộng hệ thống.

Nhưng có một vấn đề là khi làm việc, lập trình viên thường không nghĩ tới những người đó trong tương lai. Vậy thì trong trường hợp này tái cấu trúc đóng vai quan trọng là giúp cải thiện thiết kế của hệ thống, từ đó cũng giúp đọc mã dễ hơn.

Lợi ích hệ quả

Từ những lợi ích cơ bản ở trên ta có thêm các lợi ích khác: do hệ thống hiện thời có một thiết kế tốt hơn và mã dễ hiểu hơn, từ đó thì việc mở rộng hệ thống dễ dàng hơn, khó bị tổn thương hơn, nên tốc độ phát triển hệ thống luôn được duy trì; mã và thiết kế dễ đọc hơn, từ đó giúp tìm ra lỗi dễ dàng hơn; vì những mục tiêu ngắn hạn lập trình viên có thể chấp nhận một lỗ hổng nào đó về công nghệ hay thiết kế mà hiện thời không gây ảnh hưởng gì tới hệ thống, nhưng khi hệ thống lớn dần thì những lỗ hổng này

được tích tụ và làm cho hệ thống dễ bị tổn thương, thể nên việc tái cấu trúc giúp nhanh chóng sửa những lỗ hổng này.

Thời điểm thực hiện

Khi thêm một chức năng mới

Khi thêm một chức năng mới, ta phải đọc lại mã để hiểu. Như vậy nếu lúc này ta thực hiện việc tái cấu trúc, mã sẽ dễ hiểu hơn, cộng với đó là này ta cũng dễ dàng hiểu mã hơn vì mình là người đã đọc và thực hiện việc tái cấu trúc. Một lý do khác là khi thực hiện việc tái cấu trúc vào thời điểm này thì thiết kế của hệ thống sẽ tốt hơn, từ đó việc mở rộng cũng dễ dàng hơn.

Khi sửa lỗi

Khi sửa lỗi ta cũng phải đọc mã, và như vậy việc tái cấu trúc làm mã dễ đọc hơn, có cấu trúc rõ ràng hơn từ đó dễ dàng phát hiện lỗi là điều cần thiết. Và bởi thế nếu bạn được gán là người sửa một lỗi nào đó thì bạn cũng thường được gán là người phải tái cấu trúc mã.

Khi rà soát mã

Nhiều tổ chức thực hiện việc rà soát mã (code review). Rà soát mã giúp cho các lập trình viên giỏi truyền lại cho các lập trình viên ít kinh nghiệm hơn, giúp cho mọi người viết mã rõ ràng hơn. Mã có thể là rất rõ ràng với tác giả, nhưng với người khác thì có thể không, bởi thế rà soát mã sẽ làm cho nhiều người đọc mã hơn. Có nhiều người đọc mã thì mã phải dễ đọc hơn và có nhiều ý tưởng hơn được trao đổi giữa các thành viên trong nhóm hơn. Bởi thế khi bạn thực hiện rà soát bạn phải đọc mã. Lần đầu bạn đọc bạn bắt đầu hiểu mã. Lần tiếp theo bạn sẽ có nhiều ý tưởng hơn để tái cấu trúc mã, từ đó bạn có thể thực hiện việc tái cấu trúc.

Các "mã bẩn" thường gặp

Chúng ta đã biết cần thực hiện tái cấu trúc khi nào, nhưng có một câu hỏi khác là mã như thế nào thì cần tái cấu trúc? Khái niệm mã bẩn (code smell) là mã có thể sinh vấn đề một cách lâu dài, sẽ giúp ta phát hiện mã cần phải tái cấu trúc. Sau đâu chúng ta sẽ liệt kê một số loại mã bẩn thường gặp:

Mã lặp

Mã lặp Là những đoạn mã xuất hiện nhiều hơn một lần trong một hoặc nhiều ứng dụng của một chủ thể. Đó là hệ quả của các hành động: sao chép mã; các chức năng tương tự được viết bởi các lập trình viên khác nhau. Hệ quả là mã trở nên dài hơn, khó hiểu hơn và khó bảo trì hơn.

Ví dụ

Đoạn mã tính giá trị trung bình của một mảng số nguyên trên C

```
extern int array1[];
extern int array2[];
int sum1 = 0;
int sum2 = 0;
int average1 = 0;
int average2 = 0;
for (int i = 0; i < 4; i++) {
    sum1 += array1[i];
}
average1 = sum1 / 4;
for (int i = 0; i < 4; i++) {
    sum2 += array2[i];
}
average2 = sum2 / 4;
```

Ta thấy hai vòng lặp for là giống nhau!

Hàm dài

Hàm dài là quá phức tạp, có lượng mã lớn. Nên khó để hiểu, triển khai, bảo trì, tái sử dụng. Nên việc tách thành các hàm nhỏ hơn là điều cần thiết.

Lớp lớn

Lớp lớn là lớp chứa quá nhiều thuộc tính và chức năng, thường là của nhiều lớp khác. Bởi thế cúng khó để đọc, bảo trì và tái sử dụng. Nên cần thực hiện các kỹ thuật cần thiết để phân bổ thành nhiều lớp khác nhau.

Hàm có nhiều tham số đầu vào

Khi một hàm có nhiều tham số đầu vào sẽ gây khó khăn để đọc, dùng và thay đổi. Với các ngôn ngữ lập trình hướng đối tượng ta có thể nhóm các tham số có liên quan vào một đối tượng để giảm số lượng tham số đầu vào.

Tính năng không phải của lớp

Là hiện tượng một phương thức không nên thuộc một lớp, nhưng do phương thức muốn sử dụng các dữ liệu của lớp đó nên lập trình viên đã gán phương thức cho lớp. Đây là một vi phạm trong lập trình hướng đối tượng. Ta cần trả phương thức đó về đúng đối tượng.

Lớp có quan hệ quá gần gũi

Đó là hiện tượng hai lớp có thể truy xuất vào các thuộc tính riêng tư của nhau một cách không cần thiết. Điều đó dẫn đến là chính các lớp đó hay lớp con của chúng có thể thay đổi các thuộc tính của lớp con lại một cách "vô thức".

Lớp quá nhỏ

Việc tạo, bảo trì và hiểu một lớp tốn tài nguyên. Vậy nếu lớp đó quá nhỏ thì ta nên xóa bỏ lớp đó đi.

Lệnh switch

Một trong những dấu hiệu tốt của lập trình hướng đối tượng là việc không dùng lệnh switch. Vấn đề của lệnh switch chủ yếu là vấn đề lặp mã. Bạn thường gặp những lệnh switch để phân bổ các chức năng ở nhiều nơi khác nhau trong cùng một ứng dụng. Và mỗi khi bạn thêm một tính năng mới, bạn phải tìm ở tất cả các lệnh này để thay đổi.

Từ chối kế thừa

Điều này xảy ra khi một lớp được thừa kế dữ liệu cũng như các tính năng của lớp cha, nhưng lại không cần phải dùng tới chúng. Chúng ta không thể cấm đoán điều, nhưng nếu điều này xảy ra thường dẫn tới vấn đề hiểu lầm và vấn đề.

Định danh quá dài hoặc quá ngắn

Các định danh cần mô tả đủ ý nghĩa để mã dễ đọc hơn và tránh gây hiểu nhầm. Bởi thế các định danh quá ngắn thường không mô tả hết ý nghĩa và gây ra nhầm lẫn. Nhưng các định danh quá dài cũng có vấn đề tương tự. Bởi thế trong trường hợp này chúng ta có thể viết tắt hay tìm định danh thay thế.

Dùng quá nhiều giá trị

Nếu trong mã có nhiều giá trị thì khi cần phải thay đổi các giá trị đó vì một lý do nào đó (ví dụ thay đổi độ chính xác) thì bạn cần phải thay đổi ở nhiều nơi. Không chỉ thế mà không phải ai và lúc nào cũng nhớ những giá trị đó có ý nghĩa gì, nên làm cho mã trở nên khó đọc hơn. Trong trường hợp này bạn có thể thay bằng cách đặt tên cho các hằng.

Các kỹ thuật tái cấu trúc *cơ bản*

Các kỹ thuật tái cấu trúc được chia thành các nhóm tùy theo tiêu chí. Ở đây chúng ta sẽ chia theo mục đích.

Bao gói các trường của lớp để ép người dùng phải dùng thông các cách truy xuất dán tiếp như các phương thức.

Việc bao gói dữ liệu giúp tăng tính an toàn của dữ liệu, che dấu dữ liệu, giúp lớp con có thể dễ dàng ghi đè các lấy dữ liệu.

Trừu tượng hóa

Các bước thực hiện

1. Tạo phương thức setter/getter cho dữ liệu.
2. Thay tất cả các tham chiếu bằng getter/ setter tương ứng.
3. Chuyển mức truy cập của dữ liệu thành private (riêng tư).
4. Kiểm tra lại lần cuối xem còn có tham chiếu nào không.
5. Biên dịch và kiểm thử lại lần cuối.

Mã chưa được bao gói:

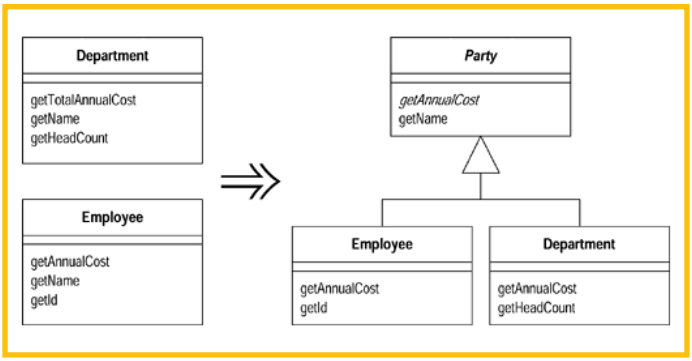
```
private int low, height;

public boolean isIn(int arg) {
    return (low >= arg)
        && (arg <= height);
}
```

Mã đã được bao gói:

```
privateint low, height;

public boolean isIn(int arg){
    return (getLow() >= arg) &&
(arg & <= getHeight());
}

public int getHeight(){
    return height;
}

public int getLow(){
    return low;
}
```

Tạo một kiểu chung cho những kiểu có chung mã.

Từ những kiểu cụ thể, tạo ra một kiểu chung chứa thuộc tính và phương thức chung cho tất cả các kiểu cụ thể. Mã bị lặp là nguyên nhân chính làm ta phải tiến hành việc tái cấu trúc này.



Việc tái cấu trúc này làm chúng ta phải viết ít mã hơn, nhiều mã được chia sẻ hơn, nên việc bảo trì, đọc cũng dễ dàng hơn.

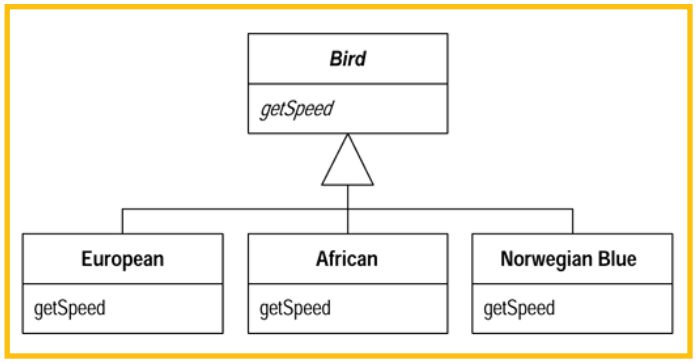
Thay các câu lệnh điều kiện bằng các sử dụng tính đa hình

Khi bạn có một câu lệnh điều kiện phụ thuộc vào loại của đối tượng. Vấn đề lớn nhất của câu lệnh điều kiện này là xuất hiện ở nhiều nơi, và mỗi lần mà bạn muốn thêm một loại mới vào thì bản phải cập nhật ở tất cả những nơi này. Khi đó tính đa hình của lập trình hướng đối tượng sẽ giúp bạn.

Cách thức tiến hành việc tái cấu trúc: Tạo một phương thức trừu tượng ở lớp cha và chuyển khối lệnh ở lệnh điều kiện vào phương thức ghi đè ở mỗi lớp con.

```
double getSpeed(){
  switch (type) {
    case EUROPEAN:
      return getBaseSpeed();
    case AFRICAN:
      return getBaseSpeed()
          - getLoadFactor() * _numberOfCoconuts
    case NORWEGIAN_BLUE:
      return _isNailed
          ? 0
          : getBaseSpeed(_voltage);
  }
  throw new RuntimeException(
      “Should be unreachable”
    );
}
```

Thiết kế sau có dùng tính đa hình



Tách phương thức

Tách từ một phương thức dài lấy một phương thức mới nhỏ hơn. Việc chia từ một phương thức dài thành nhiều phương thức nhỏ sẽ làm mã tốt hơn như: dễ hiểu, để bảo trì, dễ tái sử dụng hơn.

Các bước thực hiện

1. Tạo phương thức mới có tên phù hợp với chức năng.
2. Sao và dán đoạn mã muốn tách từ phương thức ban đầu vào phương thức mới.
3. Tìm tất cả các tham chiếu ở đoạn mã sao tới các biến của phương thức ban đầu. Các biến này sẽ là các biến cục bộ và tham số của phương thức mới.
4. Khai báo biên cục bộ cho tất cả các biến tạm ở đoạn mã sao.
5. Tìm tất cả các biến ở hàm gốc bị thay đổi giá trị ở đoạn mã sao. Nếu chỉ có một bị thay đổi thì có thể truyền giá trị đó vào bằng đối số và gán cho giá trị tương ứng. Nhưng nếu có nhiều hơn thì phải chú ý!
6. Truyền tất cả mọi biến được tham chiếu chỉ để đọc ở mã được sao vào phương thức mới như tham số.
7. Biên dịch để kiểm tra xem mọi biến cục bộ đã được xử lý.
8. Thay đoạn mã đã sao bằng phương thức mới.
9. Biên dịch và kiểm thử.

Ví dụ 1

```
void printOwing() {
  Enumeration e = _orders.elements();
  double outstanding = 0.0;

  // print banner
  println(“**************************”);
  println(“***** Customer Owes ******”);
  println(“**************************”);

  // calculate outstanding
  while (e.hasMoreElements()) {
    Order each = (Order) e.nextElement();
    outstanding += each.getAmount();
  }

  //print details
  println(“name:” + _name);
  println(“amount” + outstanding);
}
```

Dễ dàng tách đoạn mã hiển thị tiêu đề bằng các cắt và dán.

```
void printOwing() {
  Enumeration e = _orders.elements();
  double outstanding = 0.0;

  printBanner();

  // calculate outstanding
  while (e.hasMoreElements()) {
    Order each = (Order) e.nextElement();
    outstanding += each.getAmount();
  }

  //print details
  println(“name:” + _name);
  println(“amount” + outstanding);
}

void printBanner() {
  // print banner
  println(“**************************”);
  println(“***** Customer Owes ******”);
  println(“**************************”);
}
```

Ví dụ 2

Có biến cục bộ chỉ để đọc: Trong trường hợp này ta đơn giản là truyền chúng theo tham số. Ở ví dụ trên ta có thể tách phương thức để in ra thông tin chi tiết từ phương thức *printOwnin*

```
void printOwing() {
  Enumeration e = _orders.elements();
  double outstanding = 0.0;

  printBanner();

  // calculate outstanding
  while (e.hasMoreElements()) {
    Order each = (Order) e.nextElement();
    outstanding += each.getAmount();
  }

  printDetails(outstanding);
}

void printDetails(double outstanding) {
  println(“ name:” + _name);
  println(“ amount” + outstanding);
}
```

Và bạn có thể truyền vào số lượng biến cục bộ tùy thích.

Từ phương thức printOwing có ở trên:

```
void printOwing() {
  Enumeration e = _orders.elements();
  double outstanding = 0.0;

  printBanner();

  // calculate outstanding
  while (e.hasMoreElements()) {
    Order each = (Order) e.nextElement();
    outstanding += each.getAmount();
  }

  printDetails(outstanding);
}
```

Ta có thể tách thành:

```
void printOwing() {
  printBanner();
  double outstanding = getOutstanding();
  printDetails(outstanding);
}

double getOutstanding() {
  Enumeration e = _orders.elements();
  double outstanding = 0.0;
  while (e.hasMoreElements()) {
    Order each = (Order) e.nextElement();
    outstanding += each.getAmount();
  }
  return outstanding;
}
```

Tách lớp

Tách một phần của lớp đã tồn tại thành một lớp mới. Một lớp có kích thước tăng dần và tới trở nên khó hiểu và khó bảo trì. Lúc đó ta phải tách nhỏ lớp đó ra.

Các bước

1. Cách chia trách nhiệm của các lớp
2. Tạo lớp mới để chia sẻ trách nhiệm với lớp ban đầu
3. Tạo một liên kết từ lớp ban đầu tới lớp mới
4. Thực hiện viên di chuyển từng trường và phương thức từ lớp cũ sang lớp mới
5. Biên dịch và kiểm thử.

Ví dụ

Ta phải tách một lớp đơn giản:

```
class Person {
  private String _name;
  private String _officeAreaCode;
  private String _officeNumber;
  public String getName() {
    return _name;
  }
  public String getTelephoneNumber() {
    return _officeAreaCode + _officeNumber;
  }
  String getOfficeAreaCode() {
    return _officeAreaCode;
  }
  void setOfficeAreaCode(String arg) {
    _officeAreaCode = arg;
  }
  String getOfficeNumber() {
    return _officeNumber;
  }
  void setOfficeNumber(String arg) {
    _officeNumber = arg;
  }
}
```

Ta sẽ tách riêng một lớp chứa thông tin số điện thoại, để chia sẻ trách nhiệm với lớp ban đầu.

```
class TelephoneNumber {
}
```

Tạo một liên kết từ lớp Person tới lớp TelephoneNumber:

```
class Person {
   private TelephoneNumber _officeTelephone = new TelephoneNumber();
```

Thực hiện viên di chuyển từng trường và phương thức từ lớp cũ sang lớp mới:

```
class Person {
  private String _name;
  private TelephoneNumber _officeTelephone =
      new TelephoneNumber();
  public String getName() {
    return _name;
  }
  public String getTelephoneNumber() {
    return _officeTelephone.getTelephoneNumber();
  }
  TelephoneNumber getOfficeTelephone() {
    return _officeTelephone;
  }
}
class TelephoneNumber {
  private String _number;
  private String _areaCode;
  public String getTelephoneNumber() {
    return ("(" + _areaCode + ")" + _number);
  }
  String getAreaCode() {
    return _areaCode;
  }
  void setAreaCode(String arg) {
    _areaCode = arg;
}
  String getNumber() {
    return _number;
  }
  void setNumber(String arg) {
    _number = arg;
  }
}
```

Sau đó biên dịch và kiểm thử

Chuyển các phương thức về lớp phù hợp

Việc di chuyển phương thức giữa các lớp là công việc diễn ra thường xuyên trong tái cấu trúc. Việc di chuyển này giúp cho các lớp có kích thước phù hợp hơn, và các lớp ít phụ thuộc vào nhau mà khả năng hợp tác giữa các lớp tốt hơn.

Các bước

1. Kiểm tra có nên di chuyển các thuộc tính mà phương thức phải dùng
2. Kiểm tra xem có khai báo của phương thức ở lớp cha và lớp con
3. Khai báo phương thức ở lớp đích
4. Sao chép mã từ lớp nguồn tới lớp đích sao cho phương thức hoạt động được
5. Biên dịch lớp đích
6. Xác định tham chiếu phù hợp ở lớp nguồn
7. Chuyển phương thức ở nguồn gọi tới phương thức của lớp đích
8. Biên dịch và kiểm thử
9. Xác định xem có xóa phương thức ở lớp gốc và gọi trực tiếp tới phương thức ở lớp đích
10. Nếu xóa phương thức ở lớp gốc thì phải thay toàn bộ lời gọi từ phương thức này tới phương thức ở lớp mới
11. Biên dịch và kiểm thử

Ví dụ

Ta có lớp Account (tài khoản). Nhưng khi nhiều loại tài khoản và mỗi loại tài khoản có cách tính tiền phí khác nhau, nên ta muôn chuyển hàm _overdraftCharge vào lớp AccountType

```
class Account {
  private AccountType _type;
  private int _daysOverdrawn;
 double overdraftCharge() {
    if (_type.isPremium()) {
      double result = 10;
      if (_daysOverdrawn > 7)
        result += (_daysOverdrawn - 7) * 0.85;
      return result;
    } else {
      return _daysOverdrawn * 1.75;
    }
  }
  double bankCharge() {
    double result = 4.5;
    if (_daysOverdrawn > 0)
```

```
      result += overdraftCharge();
    return result;
  }
}
```

Thuộc tính _daysOverdrawn được dùng ở phương thức là một thuộc tính là của mỗi tài khoản, nên ta không di chuyển thuộc tính. Trong trường hợp này phương thức không được khai báo ở lớp cha cũng như lớp con. Ta sẽ khai báo phương thức ở lớp đích và sao chép mã tới lớp đích sao cho phương thức hoạt động được

```
class AccountType {
  double overdraftCharge(int daysOverdrawn) {
    if (isPremium()) {
      double result = 10;
      if (daysOverdrawn > 7)
        result += (daysOverdrawn - 7) * 0.85;
      return result;
    } else {
      return daysOverdrawn * 1.75;
    }
    //...
```

Chuyển phương thức ở nguồn gọi tới phương thức của lớp đích:

```
class Account {
  double overdraftCharge() {
    return _type.overdraftCharge(_daysOverdrawn);
  }
  // ...
```

Xóa phương thức ở lớp gốc và gọi trực tiếp tới phương thức ở lớp đích:

```
class AccountType {
  double overdraftCharge(Account account) {
    if (isPremium()) {
      double result = 10;
      if (account.getDaysOverdrawn() > 7)
        result += (account.getDaysOverdrawn() - 7) * 0.85;
      return result;
    } else {
      return account.getDaysOverdrawn() * 1.75;
    }
  }
  // ...
```

Sau đó biên dịch và kiểm thử

Chuyển các trường về lớp phù hợp

Nếu một trường (thuộc tính) nên được sử dụng ở một lớp khác phù hợp hơn hoặc khi ta thực hiện việc phân tách lớp thì thực hiện việc di chuyển các trường là điều cần thiết.

Các bước

1. Nếu trường đó là public, ta phải bao gói nó
2. Biên dịch và kiểm thử
3. Tạo trường ở có getter và setter ở lớp đích
4. Biên dịch lớp đích
5. Xác định cách tham chiếu tới lớp đích
6. Xóa trường ở lớp nguồn
7. Thay thế tham chiếu tới lớp nguồn bằng lớp đích.
8. Biên dịch và kiểm thử

Ví dụ

Ví dụ ta có lớp Account (tài khoản) và ta muốn chuyển trường _interestRate vào lớp AccountType

```
class Account {
  private AccountType _type;
  private double _interestRate;
  double interestForAmount_days(double amount,
      int days) {
    return _interestRate * amount * days / 365;
  }
}
```

Tạo trường ở có getter và setter ở lớp đích

```
class AccountType {
  private double _interestRate;
  double getInterestRate() {
    return _interestRate;
  }

  void setInterestRate(double arg) {
    _interestRate = arg;
  }
}
```

Thay thế tham chiếu tới lớp nguồn bằng lớp đích.

```
private double _interestRate;
```

```
double interestForAmount_days (
    double amount,
    int days) {
  return _type.getInterestRate()
      * amount
      * days / 365;
}
```

Sau đó biên dịch và kiểm thử.

Đổi các định danh

Đổi tên phương thức, thuộc tính, lớp để không dài, đủ mô tả ý nghĩa, và phù hợp với chuẩn.

Các bước

1. Kiểm tra xem phương thức có được triển khai ở lớp cha hay lớp con không. Nếu có thì phải thực hiện việc thay đổi này với từng triển khai này
2. Tạo phương thức với tên muốn đổi và sao chép mã của phương thức nguồn vào
3. Thay đổi các lời gọi tới phương thức cũ bằng phương thức mới
4. Xóa phương thức cũ.

Chuyển một thành phần của các lớp con lên cho lớp cha.

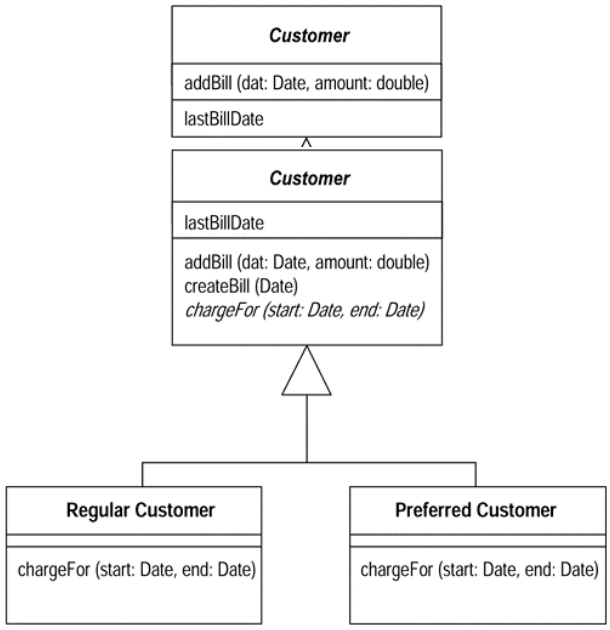
Giúp xóa bỏ mã lặp. Mặc dù việc mã lặp vẫn hoạt động bình thường, nhưng sẽ khó khăn cho nhiều việc như khó bảo trì.

Các bước

1. Kiểm tra nế các phương thức ở các lớp con có nguyên mẫu giống ở lớp cha thì sao phương thức từ một lớp con lên lớp cha
2. Xóa phương thức ở từng lớp con và kiểm thử xem có lỗi gì không tới khi chỉ còn phương thức ở lớp cha.

Ví dụ

Ví dụ ta có các lớp như hình dưới và muốn kéo phương thức chargeFor lên lớp cha.



Chuyển một thành phần của lớp cha xuống cho lớp con

Khi phương thức hoặc thuộc tính chỉ có ý nghĩa ở một lớp con cụ thể mà không phải tất cả. Khi đó thực hiện việc push down là cần thiết

Các bước

1. Khai báo phương thức ở tất cả các lớp con và sao chép thân từ lớp cha xuống lớp con.
2. Xóa phương thức ở lớp cha
3. Xóa các phương thức ở lớp con mà không cần thiết
4. Biên dịch và kiểm thử

Nguyễn Khắc Nhật

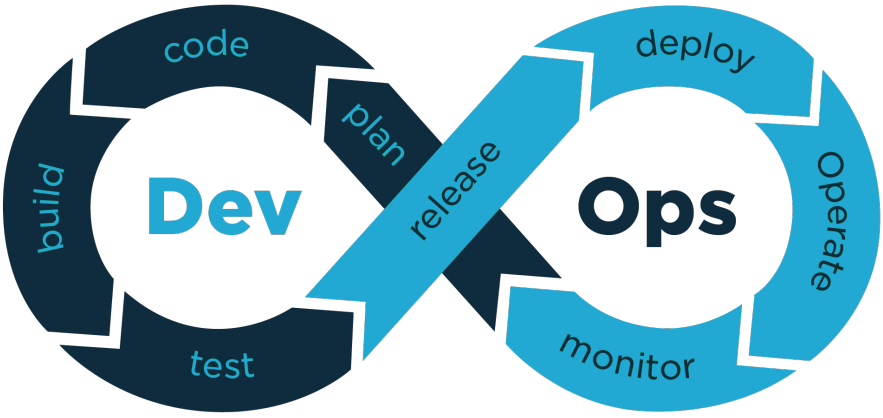
DevOps

Giải pháp phát hành phần mềm nhanh chóng

Nhanh chóng phát hành một sản phẩm mới hoặc tính năng mới ra thị trường là nhiệm vụ đầy thử thách với mọi công ty trên thế giới. Việc hóc búa nhất là làm sao để các nhóm riêng biệt: phát triển, QA và vận hành IT làm việc cùng nhau để hoàn thành công việc và phát hành sản phẩm nhanh nhất có thể.

Các quy trình và kỹ thuật đã trải qua một quá trình tiến hóa để giải quyết thử thách này. Một thập kỷ trước không ai biết đến từ DevOps, nhưng vào năm 2009, một phương pháp đã tổng hợp các quy trình để phối hợp và giao tiếp giữa nhóm phát triển, QA và vận hành IT giúp rút ngắn đáng kể thời gian đưa sản phẩm ra thị trường bắt đầu phổ biến với tên gọi DevOps.

Phương pháp DevOps là một tập hợp các kĩ thuật được thiết kế để giải quyết những vấn đề rời rạc giữa nhóm phát triển, QA và vận hành thông qua hợp tác và giao tiếp hiệu quả, kết



hợp chặt chẽ quy trình tích hợp liên tục với triển khai tự động. Giúp tạo ra môi trường để phát triển, QA và phát hành phần mềm ra thị trường nhanh chóng, ổn định.

Phương pháp Truyền thống vs DevOps

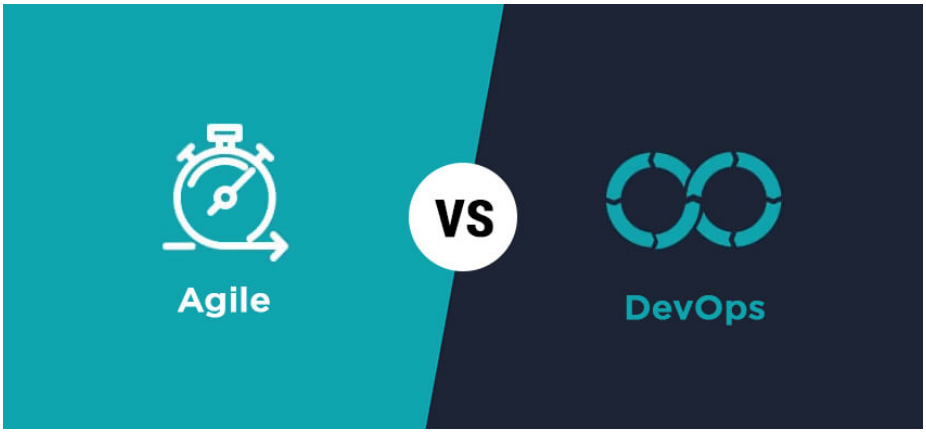
Theo phương pháp thác nước truyền thống, nhà phát triển viết mã cho các yêu cầu trên môi trường local. Khi phần mềm được build, đội QA kiểm thử phần mềm trên một môi trường tương tự như môi trường production. Cuối cùng, khi đã đáp ứng các yêu cầu, phần mềm được phát hành cho bên vận hành. Quá trình từ khi thu thập yêu cầu đến khi triển khai sản phẩm vào vận hành mất nhiều thời gian. Do hai bên phát triển và vận hành làm việc độc lập, khả năng cao là sản phẩm cuối cùng sẽ mất thêm nhiều thời gian nữa để đưa vào vận hành. Ngoài ra, sản phẩm có thể không chạy đúng như mong đợi hay gặp những trục trặc khác.

Khi đó, quy trình DevOps có một phương pháp tốt hơn để giải quyết những vấn đề trên. DevOps nhấn mạnh vào việc phối hợp và giao tiếp giữa nhóm phát triển, nhóm QA và nhóm vận hành để thực hiện việc phát triển liên tục, tích hợp liên tục, chuyển giao liên tục và giám sát các quy trình liên tục bằng cách dùng các công cụ kết nối các nhóm giúp đẩy nhanh

tốc độ phát hành. Nhờ đó công ty nhanh chóng thích ứng với những thay đổi từ yêu cầu kinh doanh.

Quan hệ giữa Agile và DevOps

DevOps ra đời một phần dựa trên khả năng phát hành sản phẩm nhanh khi công ty áp dụng Agile. Nhưng có thêm những quy trình khiến DevOps khác biệt với Agile.



Các nguyên lý của Agile chỉ áp dụng cho quá trình phát triển và QA, Agile tin tưởng vào việc tạo ra các nhóm nhỏ phát triển và phát hành phần mềm chạy tốt trong một khoảng thời gian ngắn, được gọi là Sprint. Nhóm chỉ tập trung vào Sprint và không giao tiếp với bên vận hành.

Còn DevOps lại chú trọng hơn vào việc phối hợp suôn sẻ giữa các nhóm phát triển, QA và vận hành trong suốt chu trình phát triển. Nhóm Vận hành liên tục tham gia thảo luận cùng nhóm phát triển về các mục tiêu của dự án, lộ trình phát hành và các yêu cầu kinh doanh khác. Từ ngày đầu, nhóm vận hành nên đưa ra các yêu cầu liên quan đến vận hành cho nhóm phát triển, sau đó kiểm tra lại chúng. Việc giám sát dự án liên tục cùng với giao tiếp hiệu quả và thường xuyên giúp công ty có thể nhanh chóng phát hành.

Bạn sẽ nhận được gì từ DevOps?

Hãy xem xét vài lợi ích có thể nhận được từ quy trình và văn hóa DevOps.

1. Lợi ích đầu tiên ta sẽ nhận được là DevOps giúp giảm đáng kể thời gian đưa sản phẩm ra thị trường nhờ kết nối giữa các nhóm làm việc và làm theo qui trình phát triển liên tục.
2. Nhờ các nhóm đồng bộ tốt hơn, các thành viên có được tầm nhìn rõ ràng về các công việc đang làm, do đó họ có thể thấy các vấn đề hoặc các khó khăn trước khi chúng thực sự xảy ra. Nhờ vậy thành viên nhóm có kế hoạch tốt hơn để vượt qua các vấn đề đó.
3. Nhờ sự minh bạch trong quy trình, những người phát triển sản phẩm sẽ cảm thấy mình là chủ sản phẩm. Họ thực sự sở hữu mã từ khi bắt đầu đến lúc vận hành.
4. Viện triển khai tự động sản phẩm bằng các công cụ tự động hóa trong nhiều môi trường cho phép bạn nhanh chóng thấy được các vấn đề liên quan đến môi trường. Với những phương pháp khác thì thường tốn rất nhiều thời gian để phát hiện được các vấn đề này.

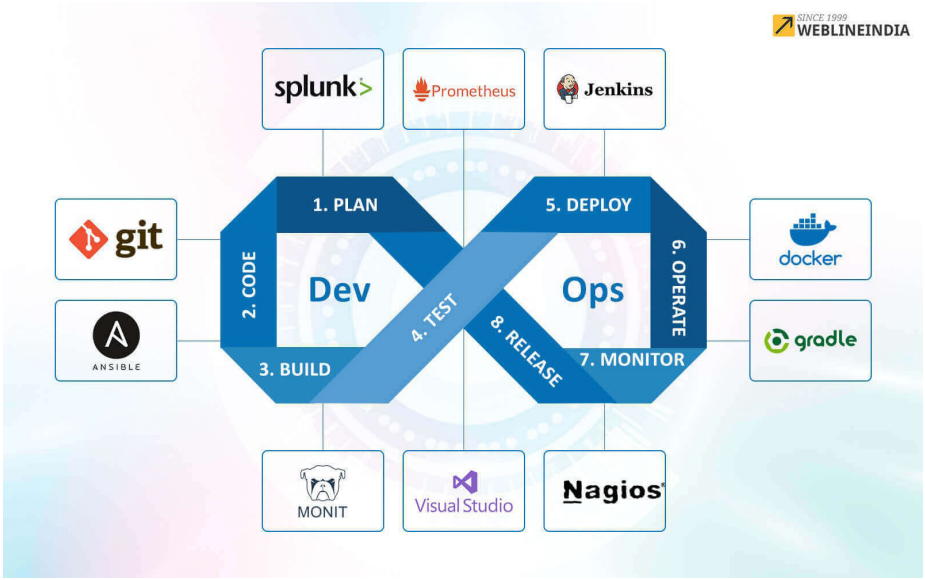
Những công cụ cho DevOps

Vì DevOps là sự cộng tác của Phát triển, QA và Vận hành, không thể có một công cụ duy nhất đáp ứng được tất cả các nhu cầu. Vì vậy cần nhiều công cụ để thực hiện thành công mỗi giai đoạn.

Hãy xem thử một vài công cụ cho mỗi giai đoạn ở dưới đây.

- Monitoring: Nagios, NewRelic, Graphite ...
- Virtualization và Containerization: Vagrant, VMware, Xen, Docker ...
- Công cụ để build, test and deployment: Jenkins, Maven, Ant, Travis, Bamboo, Teamcity...
- Quản lý cấu hình (configuration management): Puppet, Chef, Ubuntu Juju, Ansible, cfengine ...
- Orchestration: Zookeeper, Noah ...
- Cloud services: Azure, Openstack, Rackspace ...

Ngoài ra còn có các công cụ cho việc merge code, kiểm soát phiên bản,... giúp áp dụng hiệu quả các qui trình DevOps.



Các hiểu nhầm về DevOps

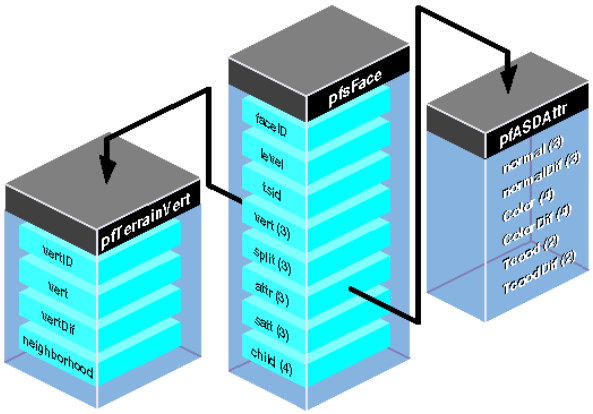
Dù DevOps đã ra đời hàng thập kỷ, nhưng nó vẫn rất mới mẻ với nhiều người. Do đó, vẫn còn nhiều hiểu nhầm.

Vài người nghĩ rằng triển khai DevOps cho phép nhà phát triển làm luôn công việc của bên vận hành. Điều này hoàn toàn không đúng. DevOps nhấn mạnh vào việc cộng tác từ cả hai nhóm, do vậy những nhà phát triển có các kỹ năng vận hành sẽ có lợi thế trong chu kỳ kinh doanh nhanh chóng hiện nay.

Cũng có rất nhiều người tin rằng có thể triển khai DevOps bằng cách dùng một bộ các công cụ cho tất cả phần việc. Điều đó không đúng; dùng vài công cụ không giúp ta đạt được DevOps, mà ta chỉ có thể đạt được các giá trị cốt lõi của nó thông qua thực sự triển khai quy trình DevOps và khôn ngoan chọn dùng các công cụ.

> *DevOps ngày càng trở lên phổ biến nhờ văn hóa phát triển liên tục, tích hợp liên tục, chuyển giao liên tục và quy trình giám sát liên tục thông qua việc cộng tác giữa Phát triển và Vận hành giúp giảm đáng kể thời gian đưa sản phẩm ra thị trường.*
>
> *Nguồn: agilebreakfast.vn*

Tìm hiểu Cấu trúc dữ liệu #1: "Chết vì thiếu hiểu biết"



Nếu bạn đã học môn "cấu trúc dữ liệu và giải thuật", hoặc đọc hết (nghiêm chỉnh) một cuốn sách về chủ đề này, thì bạn có thể bỏ qua vệt bài tôi sắp đăng lên "Tạp chí Lập trình" có tên "Tìm hiểu Cấu trúc dữ liệu" này.

Mục đích chính của vệt bài này là cung cấp các hiểu biết "chả có gì mới" về các cấu trúc dữ liệu cơ bản (bao gồm các danh sách, hàng đợi, ngăn xếp, cây, bảng băm và đồ thị) bằng cách viết giản lược và những ví dụ minh họa. Để hiểu được các bài viết trong vệt bài này, bạn cần có kiến thức cơ bản về ngôn ngữ lập trình (Java, C, C#) trước. Tôi sẽ dùng Java làm ngôn ngữ minh họa, nhưng các vấn đề được đề cập thì cố gắng trừu tượng hóa nhất có thể, vì CTDL là một vấn đề nền tảng, không phụ thuộc ngôn ngữ – một vấn đề mà mọi lập trình viên đều phải quan tâm không kể là chị ta đang sử dụng ngôn ngữ nào. Và tôi sẽ cố gắng dùng ngôn ngữ "bình dân', thay vì ngôn ngữ "học thuật" như các sách về CTDL hay dùng.

Chúng ta cùng bắt đầu với một câu chuyện thường gặp trong các tình huống lập trình. Giả sử bạn nhận được một class được viết bởi người khác (từ một API nào đó, hoặc từ một thành viên khác trong team) với một hàm quan trọng getData() có nguyên mẫu như sau:

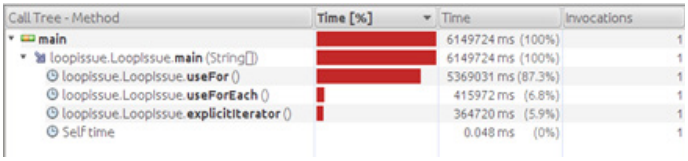
public List getData();

Hàm này trả về tất cả các dữ liệu quan trọng cho các chức năng mà bạn sẽ code ngay sau đó. Dữ liệu trả về là một danh sách (List), và bạn bắt đầu viết đoạn code đầu tiên để duyệt (traverse) toàn bộ dữ liệu trong đó như thế này:

```
//duyet danh sach
for (int i = 0; i < aList.size();
i++) {
   System.out.println(aList.
get(i));
}
```

Đơn giản, đúng không? Có bao giờ bạn đặt câu hỏi "đoạn code trên có vấn đề gì không"? Nếu chưa, thì bạn vừa được đặt câu hỏi rồi đấy. Có vấn đề gì không?

Đoạn mã nguồn đó về bản chất là "không vấn đề gì" nếu nó không rơi vào trường hợp oái oăm như sau: cái Collection mà người viết hàm getData() đã dùng để tổ chức dữ liệu là LinkedList (danh sách liên kết). Khi đó, việc dùng không đúng cách như trên có thể dẫn đến hậu quả rất tồi tệ: bạn phải ngồi chờ hàng tiếng đồng hồ để chờ chương trình kết thúc, trong khi nếu biết cách dùng cho đúng, bạn chỉ mất vài phút để duyệt hết một danh sách một triệu phần tử (thường thi khi bạn code, bạn chỉ chạy thử với vài chục phần tử nên có thể không phát hiện ra, nhưng một chương trình chạy thì vài chục nghìn cho tới một vài triệu bản ghi nằm trong một collection là chuyện ... thường ngày ở huyện), như trong hình ghi lại từ Profiler (nếu bạn chưa dùng Profiler, hãy xem qua bài "Dùng Profiler đo hiệu năng ứng dụng Java") như sau:



Hàm useFor() với đoạn code ở trên cần tới 5369031 ms (gần 90 phút) để hoàn tất việc duyệt qua tập hợp dữ liệu. Thử tưởng tượng nếu bạn code một ứng dụng web e-commerce, khách hàng sẽ bỏ bạn vì không thể kiên nhẫn đến thế. Nhìn vào hình trên, bạn thấy hàm useForEach() có thể thực hiện công việc tương tự mà chỉ cần tới 415972 ms (chưa tới 7 phút), đây là lõi của hàm useForEach():

```
//duyet danh sach
for (integer i:aList) {
   System.out.println(i);
}
```

Sự khác nhau giữa hai đoạn code chỉ ở một điểm mấu chốt: lựa chọn for hay for-each để duyệt aList. Đến đây bạn có thể đặt câu hỏi: vậy cái hàm useFor() chạy như rùa kia sai ở chỗ nào?

Câu trả lời rất đơn giản: việc duyệt theo kiểu random access (ngẫu nhiên) như trong useFor() (gọi hàm get(i) để duyệt phần tử thứ i) là sai nguyên tắc. Vì LinkedList được tổ chức đặc biệt, nên chỉ có thể được truy xuất tuần tự chứ không phải là truy xuất ngẫu nhiên thông qua chỉ số như là một danh sách dạng mảng. Để truy xuất phần tử thứ i của một danh sách liên kết, bạn sẽ phải bắt đầu từ phần tử đầu tiên (head), tuần tự đi qua các phần tử kế tiếp (thứ hai, thứ ba, v.v.) cho tới khi đến phần tử thứ i. Do đó, mỗi lần viếng thăm phần tử i, bạn tiêu tốn đúng "i" lần bước, do vậy bạn "đi" rất chậm, đặc biệt là khi "i" lớn.

Với việc dùng for-each, danh sách sẽ được dùng theo cách tối ưu với danh sách liên kết. Để hiểu rõ cơ chế duyệt theo for-each, chúng ta thử nhìn một đoạn code minh họa khác như dưới đây:

```
for (Iterator<Integer> it = aList.
iterator(); it.hasNext();) {
   System.out.println(it.next());
}
```

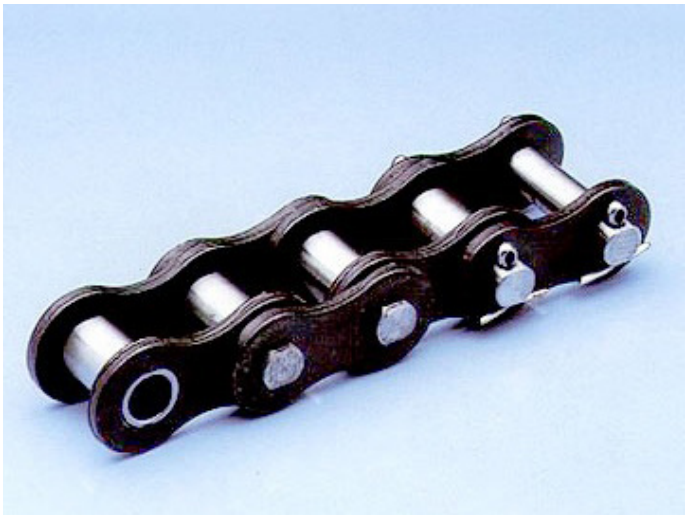
Trong Java, mỗi một tập hợp (collection) đều được tổ chức với một iterator (một đối tượng phụ trợ cho việc duyệt qua các phần tử bên trong tập hợp). Khi duyệt qua aList, iterator của một LinkedList này đã đánh dấu phần tử đang duyệt (current), như đang đặt con trỏ (cursor) ở đó vậy; do đó, khi gọi hàm it.next(), thì iterator đó không phải mất công dò từ đầu (head) cho tới phần tử thứ i, mà chỉ cần lần theo liên kết để đến với phần tử tiếp theo, tính từ vị trí đang đứng (current), mất thêm đúng một lần di chuyển. Đó chính là lý do tại sao hàm useFor() lại mất thì giờ đến vậy, trong khi hàm useForEach() và explicitIterator() thì lại rất tiết kiệm thì giờ.

Bạn thấy đấy, chỉ khác có mỗi cái lệnh lặp mà chuyện xem ra phức tạp. "Sai một li đi một dặm". Trong lập trình, đôi khi tốc độ của chương trình chỉ do một dòng code quyết định. Và vệt bài "Tìm hiểu Cấu trúc dữ liệu" sẽ cố gắng cung cấp các hiểu biết cơ bản để bạn không phải mất "dặm" nào, với các tình huống liên quan đến cấu trúc dữ liệu. Nhưng đây là câu chuyện dài kì, hãy tạm dừng ở đây đã. Ở bài tới (sẽ rất sớm thôi) chúng ta sẽ trò chuyện về các cấu trúc tuần tự: danh sách.

Dưới đây là code trong bài để bạn chạy thử, hãy cho BOUNDARY nhận các giá trị khác nhau và quan sát kĩ hơn về performance đo được bởi Profiler.

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

public class LoopIssue {
    public static final int BOUNDARY = 1000000;
    public static void main(String[] args) {
      List<Integer> aList = (new LoopIssue()).getData();
      useFor(aList);
      explicitIterator(aList);
      useForEach(aList);
    }

    /**
     * Get data from a datasource, a simulation of real situations
     */

    public List<Integer> getData() {
      List<Integer> aList = new LinkedList();
      for (int i = 0; i < BOUNDARY; i++) {
          aList.add(i);
      }
      return aList;
    }

    static void useFor(List<Integer> aList) {
      //read the list
      for (int i = 0; i < aList.size(); i++) {
          System.out.println(aList.get(i));
      }
    }

    static void useForEach(List<Integer> aList) {
      //read the list
      for (Integer i : aList) {
          System.out.println(i);
      }
    }

    static void explicitIterator(List<Integer> aList) {
      //read the list
      for (Iterator<Integer> it = aList.iterator(); it.hasNext();) {
         System.out.println(it.next());
      }
    }
}
```

#2:
Bên trong danh sách liên kết



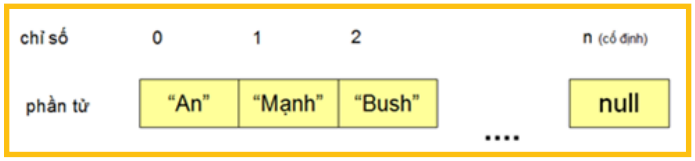
Bài trước chúng ta có nói tới LinkedList (danh sách liên kết hay "danh sách móc nối") được tổ chức đặc biệt, không giống như cấu trúc mảng. Vậy nó được tổ chức như thế nào? Bài này sẽ đi sâu vào tìm hiểu cấu trúc bên trong của danh sách liên kết, đồng thời rút ra một số kết luận cho việc sử dụng cấu trúc dữ liệu này trong xây dựng các chương trình.

Trong bài này:

- Các loại danh sách
- Nút: hạt nhân của danh sách liên kết
- Tổ chức DSLK
- Thao tác trên DSLK
- Khi nào thì dùng DSLK?

Danh sách là cấu trúc dữ liệu đơn giản, nhưng cũng có nhiều cách thức tổ chức khác nhau cho từng mục đích sử dụng. Có thể kể đến: dùng mảng tĩnh (Array), mảng động với interface dạng danh sách (ArrayList, Vector) và danh sách liên kết (LinkedList).

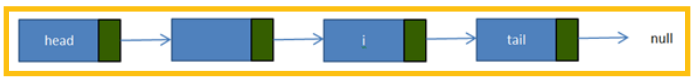
Cách đơn giản nhất để tổ chức một danh sách các phần tử là dùng mảng. Đây là CTDL đơn giản nhất có sẵn trong hầu hết các ngôn ngữ lập trình hiện đại, có khả năng lưu trữ một dãy cố định các phần tử (có thể là các giá trị primitive hoặc object).



Tuy có sẵn và dễ dùng nhưng mảng bị giới hạn số lượng phần tử chứa trong danh sách. Khi bạn khởi tạo với boundary là 20 thì Array chỉ có khả năng chứa được 20 phần tử trong mảng, không hơn. Hơn nữa, trong trường hợp muốn chèn một phần tử vào giữa mảng, có thể ta sẽ phải dịch chuyển một số lượng lớn các phần tử đứng sau. Chuyện tương tự cũng xảy ra với thao tác xoá một phần tử nào đó trong danh sách. Ưu điểm cơ bản của danh sách dạng Array (bao gồm kiểu dữ liệu mảng, và các lớp wrapper kiểu mảng là ArrayList và Vector) là nhanh, và tiện. Bạn có thể truy xuất ngẫu nhiên các phần tử để thao tác trên đối tượng đó. Tuy vậy, kiểu tổ chức này có một nhược điểm cố hữu là không cho phép bạn kéo dài danh sách ra (boundary phụ thuộc vào lần khởi tạo đầu tiên). Trong trường hợp cần đưa thêm một phần tử, cần copy Array sang một vùng nhớ mới rộng hơn, gây lãng phí bộ nhớ và CPU.

Vì các lý do đó, LinkedList (DSLK) được dùng như là một sự lựa chọn nữa để khắc phục các nhược điểm trên của Array.

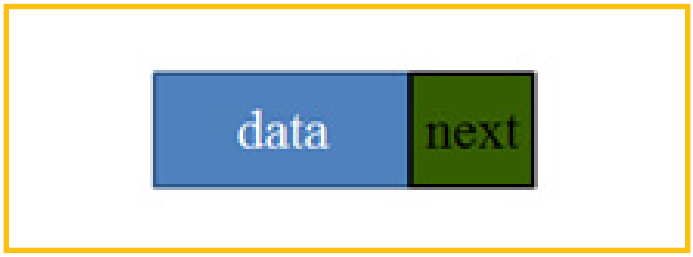
Ta có thể mường tượng DSLK như là một chuỗi (chain) các nút (node) được móc nối (link) với nhau:



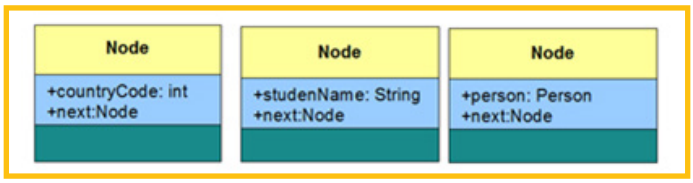
Trong đó, DSLK có một nút đặc biệt là đầu (head) – đánh dấu điểm bắt đầu; và một nút đuôi (tail)-đánh dấu điểm kết thúc của danh sách. Chúng cũng được gọi là nút đầu tiên (First), và nút cuối cùng (Last).

Nút (node) – thành phần hạt nhân của mỗi danh sách

Tổ chức của mỗi nút bao gồm hai thành phần cơ bản: phần dữ liệu (data) và phần tham chiếu tới phần tử tiếp theo (next):



Phần data có thể đơn giản là số nguyên, có thể là một string hoặc cũng có thể là một object bất kì. Phần next chính là chìa khoá để tạo lập sự liên kết giữa các phần tử trong mảng (các nút – Node). Biến next sẽ trỏ tới một nút khác (là một đối tượng kiểu Node) và xác lập sự liên kết giữa hai nút. Dưới đây là một số ví dụ về các nút:

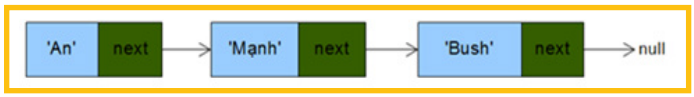


Cấu trúc đầu tiên có thể được dùng để lưu trữ mã số các quốc gia tạo thành một danh sách các mã số điện thoại quốc gia; cấu trúc thứ hai là một nút trong danh sách các tên của sinh viên; còn cái thứ ba có thể chứa thông tin chi tiết về một người (person). Đây chính là phần dữ liệu (data) như trong mô tả ở trên. Điểm chung của cả ba cấu trúc này là có phần next có kiểu chính là Node. Với cấu trúc như vậy, đối tượng Node này có thể móc nối vào một đối tượng Node khác

Thực sự thì next sẽ đóng vai trò quan trọng như thế nào?

Cấu tạo danh sách

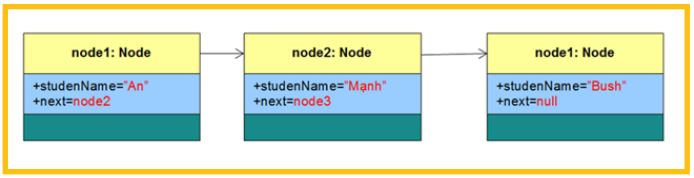
Quan sát danh sách được tạo từ các nút được mô tả ở trên trong hình dưới đây:



Bằng cách móc nối các nút lại với nhau thông qua next, xác lập được một chuỗi các đối tượng "có liên quan" được "dính" lại với nhau . Nhờ next, ta biết rằng có sinh viên 'Mạnh' đứng sau 'An' trong danh sách, và 'Bush' đứng cuối cùng. Đó là cách một danh sách được hình thành.

Trong Java, next chính là một biến tham chiếu (reference variable). Việc next trỏ tới object nào sẽ quy định object đó có mặt tiếp theo trong danh sách. Nhờ yếu tố này, danh sách mà chúng ta nhìn thấy ở trên được gọi là danh sách liên kết. Người ta còn gọi danh sách như trên là danh sách tự tham chiếu, vì mỗi phần tử của nó tự xác định phần tử tiếp theo.

Để dễ theo dõi ta sẽ xem một danh sách sinh viên được tạo thành như thế nào từ các cấu trúc Node thứ hai ở trên. Quan sát hình dưới đây:



Như vậy, với đối tượng node1, node2, node3 ta đã có thể tạo lập được một danh sách liên kết. Chìa khoá ở đây chính là dùng next để trỏ đến phần tử đứng sau trong danh sách.

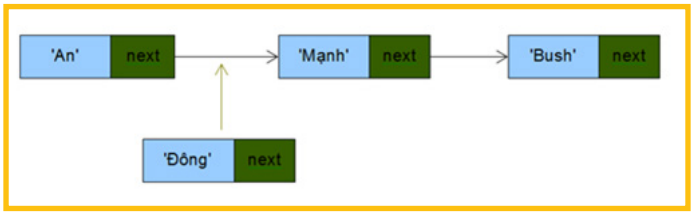
Song, vấn đề cơ bản của cấu trúc dữ liệu còn bao gồm một việc quan trọng khác: cho phép thao tác (operate) những gì trên cấu trúc dữ liệu đó. Để cài đặt một danh sách liên kết, ta cần một lớp đóng vai trò người quản lý các thành phần của nó bao gồm các chức năng cơ bản như thêm-sửa-xóa phần tử vào khỏi danh sách, chèn vào giữa, tìm phần tử trong danh sách, duyệt các phần tử, v.v.. Đó chính là lớp LinkedList mà bạn thường nhìn thấy trong các Collection Framework. Sau đây ta sẽ khảo sát chi tiết các thao tác cơ bản của DSLK.

Thao tác trên danh sách liên kết

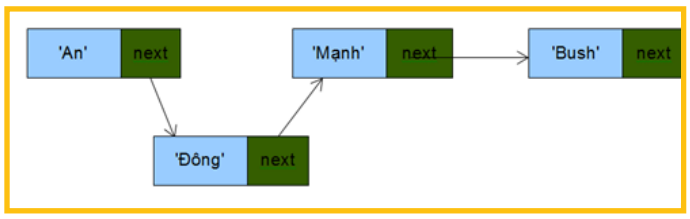
Như thấy ở trên, việc tạo lập một danh sách như trên là tương đối nhiêu khê. Tuy nhiên, như sẽ thấy dưới đây, sự phức tạp đó sẽ mang lại nhiều lợi ích rõ rệt.

Tôi muốn thêm một sinh viên vào danh sách thì sao?

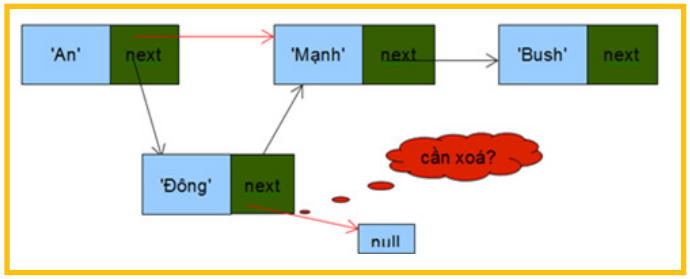
Chuyện đó hết sức đơn giản. Chỉ cần khởi tạo một node mới, đặt vào vị trí cần thiết. Ta sẽ không gặp phải vấn đề giới hạn kích thước như trong Array, cũng không lo tới việc phải dịch chuyển các phần tử trong danh sách. Việc thêm thắt chưa bao giờ dễ hơn thế!



Chỉ cần trỏ next đến nơi cần đến (ví dụ đưa next của 'An' vào 'Đông', rồi trỏ next của 'Đông' đến 'Mạnh'), thế là xếp được một sinh viên mới vào danh sách:



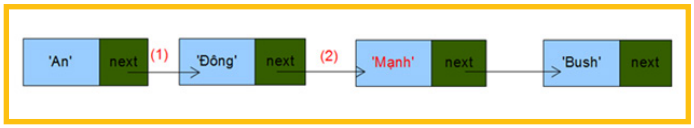
Tương tự, việc xoá một phần tử khỏi danh sách cũng đơn giản không kém:



Di chuyển các liên kết, ta sẽ đạt được mục đích. Bằng cách gán next của phần tử cần xoá tới null, gắn lại mối kết nối vừa bị đứt (như hình trên, móc 'An' vào 'Mạnh').

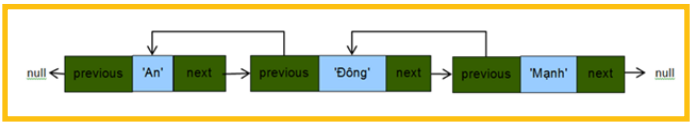
Chỉ mỗi việc duyệt DSLK là hơi khó chịu một chút

Để tìm một nút trong danh sách, ta thực hiện việc duyệt các nút trong danh sách. Từ nút đầu tiên, ta lần theo các nút tiếp theo nhờ liên kết next- chính next sẽ chỉ cho ta người tiếp theo trong danh sách. Cho tới khi next trỏ tới null, hoặc khi tìm thấy nút cần tìm, ta sẽ kết thúc việc tìm kiếm. Đây là đặc trưng cơ bản của việt duyệt tuần tự (sequential), bạn không thể truy xuất ngẫu nhiên (ngay lập tức) được, mà phải rất từ từ: bạn muốn tìm thấy "Mạnh", thì phải đi qua nhà "An", "Đông" trước đã. Và cứ lần nào tìm kiếm như thế, bạn cũng phải bắt đầu từ "head", nếu không có cách gì khác.



Để tiện lợi hơn cho việc duyệt qua danh sách, có hai cải tiến đáng kể: một là thêm một liên kết nữa cho mỗi nút: previous,

khi đó danh sách sẽ là loại DSLK đôi (Doubly Linked List) – cách này cho phép bạn tiến-lùi thoải mái hơn; hai là đặt thêm vào cấu trúc LinkedList một lớp phụ trợ (utility class) có tên Iterator để giúp việc duyệt qua danh sách tiện lợi hơn, đỡ tránh sai sót (như đề cập ở bài trước), và hỗ trợ vòng lặp for-each. Iterator là một mẫu thiết kế được sử dụng phổ biến trong thiết kế các Collection Framework do tính ưu việt của nó: trừu tượng hóa cách thức duyệt qua các phần tử trong kiểu tập hợp (collection) mà vẫn cho phép tối ưu hóa cách duyệt đặc thù cho từng kiểu CTDL. Chúng ta sẽ bàn kĩ về mẫu thiết kế (design pattern) này sau. Trong LinkedList, iterator của mỗi danh sách sẽ đánh dấu phần tử đang duyệt (current), giống như một con trỏ (cursor) để tiếp tục việc duyệt mà không cần phải quay lại duyệt từ đầu trong các lần duyệt tiếp theo, nhờ đó tiết kiệm thời gian duyệt phần tử hơn.



Lưu ý

Như bài trước có đề cập tới tình trạng dở khóc dở cười khi ta dùng hàm get(i) từ một đối tượng LinkedList. Thực ra đây là một lỗi thiết kế của JDK vì LinkedList thừa kế List nên nó thừa kế cả hàm get(i) của List (một số cấu trúc khác cũng vướng phải vấn đề này). Ta biết rằng việc truy xuất kiểu tuần tự như thế là không phù hợp trong một số trường hợp (như duyệt hết danh sách chẳng hạn). Vì thế, để cho 'lành', ta sử dụng for-each để tận dụng ưu việt của việc duyệt theo iterator mỗi khi gặp phải một collection mà không chắc nó được cấu trúc ra sao.

Mở rộng DSLK

LinkedList có thể được dùng làm Stack, Queue (như trong Java, LinkedList được cài đặt với một số hàm để phù hợp với cấu trúc hàng đợi và ngăn xếp), tuy nhiên, còn tùy sự lựa chọn cài đặt cho từng bài toán cụ thể. Ngoài ra, LinkedList cũng có thể được mở rộng để tạo ra các danh sách nhảy cóc (Skip List), và danh sách dạng vòng (Circular List) với nhiều ưu điểm phù hợp với một số bài toán đặc thù.

Cấu trúc Node có thể dùng để làm thành các phần tử của cây (Tree), đặc biệt với cây tìm kiếm nhị phân (Binary Search Tree). Ta sẽ xem xét các cấu trúc dữ liệu này sau.

Cuối cùng, vậy khi nào thì dùng LinkedList?

Căn cứ các hiểu biết về cấu trúc như trên, ta có thể rút ra một số kết luận mang tính lý thuyết về việc khi nào thì nên dùng không nên dùng danh sách liên kết.

Nên dùng DSLK:

1. Khi cần xóa chèn liên tục (trong các ứng dụng real-time chẳng hạn)
2. Khi bạn không biết collection có bao nhiêu phần tử (boundary không rõ ràng), đặc biệt lại là khi bộ nhớ hữu hạn không cho phép bạn thực hiện các biện pháp clone dữ liệu trên mảng.
3. Khi bạn không có nhu cầu truy xuất ngẫu nhiên vào phần tử nào
4. Khi bạn rất hay muốn chèn một phần tử vào giữa danh sách (như trong cài đặt các hàng đợi ưu tiên – priority queue – chẳng hạn).

Trong trường hợp khác, có thể bạn sẽ ưa thích dùng mảng (Array, ArrayList) hơn:

1. Khi cần (thường xuyên) truy xuất ngẫu nhiên
2. Số lượng phần tử trong collection là xác định
3. Khi bạn cần duyệt qua thật nhanh tất cả các phần tử trong danh sách
4. Muốn tiết kiệm bộ nhớ (vì dùng LinkedList bạn sẽ tốn một ít overhead cho việc tổ chức các liên kết next, previous và Iterator)

Chuyện về LinkedList đến đây đã có phần hơi dài, mặc dù còn nhiều cái hay để bàn tiếp. Nhưng như thế chắc cũng đủ để hiểu về cấu trúc của LinkedList rồi. Chúng ta tạm thời ngưng ở đây để cùng chuyển sang một cấu trúc dữ liệu cực kì căn bản: ngăn xếp (Stack). Hãy chú ý tới phần phụ lục dưới đây như là một bài tập để thử nghiệm, điều đó sẽ giúp bạn đào sâu hơn về LinkedList.

Phụ lục: Tự cài đặt LinkedList

Bạn có thể tự cài LinkedList để hiểu hơn về cấu trúc dữ liệu này. Dưới đây là đoạn code mẫu cài đặt hoàn chỉnh một danh sách liên kết đơn. Bạn thử mở rộng thành DoubleLinkedList và cài đặt Iterator cho nó (interface này có tại java.util) xem hiệu quả thế nào. Cũng đừng quên thí nghiệm với các thao tác tìm-thêm-sửa-xóa trên các cấu trúc bạn cài đặt nhé. Thử so sánh với ArrayList(hay Vector) xem thế nào. Nếu bạn quan tâm đến performance, thì code cái giả định của bạn đi, và để Profiler cho bạn biết kết quả.

Code: https://bitly.com.vn/WM9Bc

Các nguyên tắc thiết kế API cho Java 8

Java 8

Bất cứ ai viết mã Java đều là một nhà thiết kế API!

Mã của họ đều rồi sẽ được dùng bởi ai đó, cộng đồng, đồng nghiệp, chính họ, hay có khi tất cả. Do đó, biết các nguyên tắc cơ bản của một API tốt là rất quan trọng với tất cả các nhà phát triển Java.

Bài viết này giúp bạn học cách trở thành một lập trình viên Java tốt hơn bằng cách đưa ra các nguyên tắc giúp cho lập trình viên Java có khả năng viết các API:

- Hiển hiện một thiết kế tốt, và giấu tốt được các chi tiết triển khai
- Đảm bảo mã gọi tới API có thể sử dụng biểu thức lambda
- Đảm bảo API có thể tiến hóa một cách có kiểm soát
- Loại bỏ tất cả các khả năng gặp phải NullPointerException

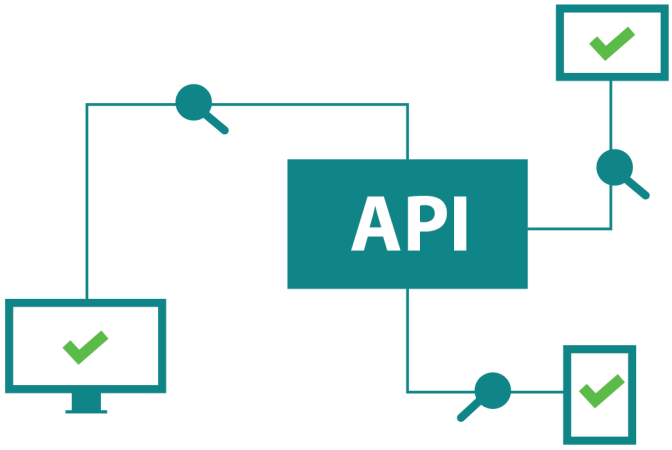
Từ triển khai được dùng thường xuyên trong bài viết này, là dịch nghĩa của thuật ngữ implement. Các thuật ngữ chuyên ngành sẽ được in nghiêng trong các trường hợp cần thiết.

Mở đầu

Thiết kế một API tốt đòi hỏi kinh nghiệm và sự cân nhắc cẩn trọng. Cần làm cho nó đúng ngay từ đầu. Một khi API đã được xuất bản nghĩa là ván đã đóng thuyền, và nhà phát triển đã tự tạo ra một cam kết vĩnh viễn không đổi với những người sử dụng nó. Do đó thiết kế API cần kết hợp hài hòa khả năng cam kết chắc chắn với sự linh hoạt nhất định trong triển khai cao.

API tốt cũng cần làm cho mã của người dùng (từ giờ sẽ được gọi là mã khách) được đẹp, thanh nhã và dễ đọc. Đồng thời, API tốt phải giấu được càng nhiều chi tiết về sự triển khai càng tốt. Và đó là nhiệm vụ của thao tác thiết kế.

Đó đều là những nhiệm vụ khó nhằn, trên thực tế, việc thiết kế nhiều khi khó hơn viết mã triển khai rất nhiều. Thiết kế cũng như một nghệ thuật, khó nắm bắt và thành thạo. Nhưng cũng như bạn không nhất thiết phải là một ca sĩ sừng sỏ mới có thể thể hiện tốt một bài hát, chỉ cần bạn hát đúng nhịp và đúng cao độ là mọi chuyện đã rất tốt rồi. Chúng ta cũng có thể áp dụng một checklist, để API dễ dàng tránh được những sai lầm nghiêm trọng nhất trong thiết kế, trở thành lập trình viên tốt hơn, tiết kiệm được nhiều thời gian và sống hạnh phúc mãi mãi.



Không dùng null để nói "không có kết quả trả về"

Xử lý null không cẩn thận (dẫn tới sự phổ biến của NullPointerException) có khi là nguồn gốc của nhiều lỗi nhất trong lịch sử các chương trình Java. Vài nhà phát triển còn cho rằng việc cung cấp khái niệm null là một trong những sai lầm tệ hại nhất của lịch sử khoa học máy tính.

May mắn, Java 8 đã giới thiệu lớp Optional như một trong những bước đầu tiên để giảm bớt vấn đề này. Theo đó, một phương thức sẽ có thể sử dụng một đối tượng Optional thay vì null để mô tả trường hợp không có kết quả trả về. Đừng bị cám dỗ bởi một chút hiệu năng để rồi ngang bướng sử dụng null. Dù sao thì Java 8 Escape Analysis cũng sẽ phân tích và tối ưu hóa hầu hết các đối tượng Optional.

Tất nhiên, nhớ tránh sử dụng cácOptional làm tham số và thuộc tính.

Nên:

```
public Optional<String>
getComment() {
  return Optional.
ofNullable(comment);
}
```

Không nên:

```
public String getComment() {
  return comment; // comment is
nullable
}
```

Không sử dụng các mảng để truyền giá trị ra và vào API

Một lỗi thiết kế khá lỗi đã được tạo ra trong Java 5, nằm ở API Enum. Chúng ta đều biết rằng lớp Enum có một phương thức gọi là values() trả về một mảng tất cả các giá trị khác nhau của Enum. Vấn đề là Java framework cần đảm bảo rằng mã khách không thể thay đổi tập các giá trị của Enum (chẳng hạn bằng cách sửa đổi đối tượng mảng), nên phương thức values() buộc phải tạo lại một đối tượng mảng mới mỗi lần được gọi. Dĩ nhiên là mất cả hiệu năng lẫn tính dễ sử dụng.

Giá mà API Enum có thể trả về một đối tượng List chỉ-đọc, khi đó nó sẽ có thể dùng lại đối tượng qua vô số lời gọi từ phía mã khách, và mã khách cũng sẽ nhận được một model dễ dùng hơn. Mảng cũng không hề thích hợp để làm tham số đầu vào cho các phương thức hiển hiện của API, bởi trừ khi bạn cẩn thận clone mảng nhận được thành đối tượng mới, nếu không sẽ chẳng thể lường được nó bị một luồng nào đó khác chỉnh sửa ngay trong trong khi phương thức đang thực thi.

Trong phần lớn trường hợp, nếu cần chuyển giao vào ra một bộ các phần tử, chúng ta nên cân nhắc viện đến cấu trúc dữ liệu Stream. Stream chỉ-đọc một cách tự nhiên (trái ngược với List - vốn đi cùng với phương thức set). Stream cho phép mã khách dễ dàng thao túng cũng như thu gom các phần tử thành một cấu trúc dữ liệu khác.

Nên:

```
public Stream<String>
comments() {
  return Stream.of(comments);
}
```

Không nên:

```
public String[] comments() {
  return comments; // Exposes
the backing array!
}
```

Và Stream kết hợp tối ưu với Java 8 Escape Analysis, cho phép API trì hoãn việc

gom các phần tử vào nó cho tới khi mã khách cần tới (chiến thuật "lazy") chừng nào nguồn dữ liệu vẫn còn hiện diện, giúp đảm bảo có tối thiểu số lượng đối tượng được tạo vào Heap.

Cân nhắc sử dụng phương thức *factory*

Tránh trao cho mã khách quyền lựa chọn lớp triển khai của một interface. Việc này khiến cho mã của API và mã khách "dính" vào nhau nhiều hơn. Khiến số lượng các lớp không được phép thay đổi hành vi với bên ngoài bị tăng lên, và phổ cam kết của API trở nên lớn hơn.

Hãy cân nhắc tạo các phương thức tĩnh chuyên dụng để giúp mã khách tạo đối tượng của lớp triển khai interface. Chẳng hạn, nếu có interface Point với hai phương thức int x() và int y(), chúng ta có thể hiển hiện một phương thức tĩnh (int x, int y) trả về một triển khai (ẩn) của interface. Để rồi nếu cả x và y đều bằng 0, trả về sẽ là một lớp triển khai tên là PointOrigoImpl không có cả trường x lẫn y, còn ngược lại thì trả về sẽ là một lớp khác mang tên PointImpl mang giá trị x và y chỉ định.

Lưu ý đảm bảo rằng các lớp triển khai nằm trong một package riêng biệt với API (chẳng hạn đặt interface Point trong package com.company.product.shape và các lớp triển khai trong package com.company.product.internal.shape).

Nên:

```
Point point = Point.of(1,2);
```

Không nên:

```
Point point = new PointImpl(1,2);
```

Ưu tiên composition với các functional interface và lambda thay vì kế thừa

Chúng ta nên tránh hoàn toàn việc cho phép kế thừa từ API. Dù gì cũng chỉ có duy nhất một lớp cha cho bất kỳ lớp Java nào, nên không phải bao giờ mã khách cũng có thể kế thừa lớp nào đó từ API. Chưa kể đến, hiển hiện một lớp trừu tượng trong API đồng nghĩa với cho phép chúng được kế thừa bởi mã khách, và đó sẽ là một cam kết rất nặng nề cho API.

Thay vào đó, nên sử dụng interface, cho nó các các phương thức tĩnh nhận các lambda làm tham số, và dựa vào các lambda đó để cấu thành một lớp triển khai nội bộ. Điều này cũng giúp mã khách dễ đọc hơn nhiều. Như trong ví dụ dưới đây, thay vì phải kế thừa một lớp public AbstractReader và ghi đè abstract void handleError(IOException ioe), sẽ tốt hơn cho mã khách nhiều nếu API hiển hiện một phương thức hay một builder trong interface Reader mà nhận vào một Consumer<IOException> và đưa chúng cho một triển khai nội bộ.

Mã khách nên:

```
Reader reader = Reader.builder()
 .withErrorHandler
(IOException::printStackTrace)
.build();
```

Không nên:

```
Reader reader = new
AbstractReader() {
@Override
public void handleError
(IOException ioe) {
ioe.printStackTrace();
  }
};
```

Đảm bảo rằng funtional interface đi kèm với annotation @FunctionalInterface

Gắn một interface với annotation @FunctionalInterface báo hiệu rằng mã khách có thể sử dụng lambda để triển khai interface, nó cũng đảm bảo interface sẽ luôn chỉ có duy nhất một phương thức trừu tượng và nhờ đó duy trì được khả năng triển khai bằng lambda trong tương lai.

Nên:

```
@FunctionalInterface
public interface CircleSegmentConstructor {
  CircleSegment apply(Point cntr, Point p, double ang);
  // abstract methods cannot be added
}
```

Không nên:

```
public interface CircleSegmentConstructor {
  CircleSegment apply(Point cntr, Point p, double ang);
  // abstract methods may be accidently added later
}
```

Tránh nạp chồng các phương thức theo tham số functional interface

Việc hai hay nhiều phương thức có cùng tên đều lấy functional interface làm tham số có thể tạo ra sự mơ hồ cho lambda ở mã khách. Lấy ví dụ, với Point có hai phương thức add(Function<Point, String> renderer) và add(Predicate<Point> logCondition), khi chúng ta gọi point.add(p -> p + " lambda") ở mã khách, compiler sẽ không thể xác định được phương thức nào cần dùng và cho ta lỗi biên dịch. Thay vào đó, nên cân nhắc sử dụng các tên phương thức khác nhau cho mỗi trường hợp sử dụng đặc thù.

Nên:

```
public interface Point {
  addRenderer(Function<Point, String> renderer);
  addLogCondition(Predicate<Point> logCondition);
}
```

Không nên:

```
public interface Point {
  add(Function<Point, String> renderer);
  add(Predicate<Point> logCondition);
}
```

Tránh định nghĩa các phương thức default trong interface

Java 8 cho phép đặt các phương thức default vào các interface và đôi khi ta có lý do để làm điều đó. Chẳng hạn để chọn một cách mặc định để triển khai phương thức cho bất kỳ lớp triển khai nào. Ngoài ra, như chúng ta đã biết, các functional interfacechỉ có khả năng chứa chính xác một phương thức trừu tượng, do đó khi cần bổ sung những phương-thức-kế-thừa-được, sử dụng phương thức default là một cách khả dĩ. Việc này rất hữu ích để phục vụ tính tương thích ngược khi nâng cấp API.

Dù vậy, lạm dụng quá có thể làm cho interface của API trông không khác gì một lớp triển khai sau khi bị nhồi nhét quá nhiều các phương-thức-không-trừu-tượng không cần thiết. Hãy cân nhắc di chuyển các phương thức chứa logic sang một lớp Util riêng biệt hay đặt chúng trong các lớp triển khai.

Nên:

```
public interface Line {
  Point start();
  Point end();
  int length();
}
```

Không nên:

```
public interface Line {
  Point start();
  Point end();
  default int length() {
    int deltaX = start().x() - end().x();
    int deltaY = start().y() - end().y();
    return (int) Math.sqrt(
        deltaX * deltaX + deltaY * deltaY
    );
  }
}
```

Validate các tham số đầu của API vào trước khi xử lý

Nhiều trường hợp ai đó kiểm tra tham số đầu vào của API một cách cẩu thả, và khi có lỗi xảy ra, nguyên nhân thực sự bị che giấu nhiều lớp rất sâu dưới stack trace. Hãy đảm bảo rằng các tham số được kiểm tra tính hợp lệ trước khi chúng được sử dụng trong các lớp triển khai. Nếu cần kiểm null, phương thức Objects.requireNonNull() rất hữu dụng. Về mặt hiệu năng, JVM có khả năng tối ưu hóa những phép kiểm tra không cần thiết.

Validate tham số cũng là một cách tốt để thi hành cam kết của API. Nếu một API được mô tả là không chấp nhận các null nhưng rồi vẫn nhận theo cách nào đó, người dùng sẽ bị bối rối.

Nên:

```
public void addToSegment(Segment segment, Point point) {
  Objects.requireNonNull(segment);
  Objects.requireNonNull(point);
  segment.add(point);
}
```

Không nên:

```
public void addToSegment(Segment segment, Point point) {
  segment.add(point);
}
```

Tổng kết

Điều quan trọng là áp dụng vào thực hành. Bạn đã biết một số quy tắc để thiết kế nên một API tốt. Hầu hết đều không khó áp dụng, và bạn luôn có thể bắt đầu bằng một quy tắc bất kỳ mà bạn cảm thấy phù hợp, sau đó mở rộng dần khả năng sang các quy tắc khác.

Nguyễn Bình Sơn

Những MÃ XẤU mà JAVA 8 có thể khử

Tới hiện tại, Java 8 đã được sử dụng trên hầu hết các ứng dụng chạy trên JVM, nhưng điều đó không có nghĩa là những gì tối tân của phiên bản này đã được khai thác triệt để. Dưới đây là một số cách viết mã già cỗi nên được cập nhật.

1. Inner class vô danh

Bất cứ khi nào bạn gặp một inner class vô danh, bạn nên cân nhắc sử dụng biểu thức lambda.

Ví dụ

```
list.sort(new Comparator() {
  public int compare (String o1, String o2) {
    return o1.length() - o2.length();
  }
});
```

...chuyển thành như sau ngắn gọn hơn nhiều:

```
list.sort((o1,o2) -> o1.length() - o2.length());
```

Dù vậy, mã ở đây vẫn không quá dễ đọc, chúng ta sẽ tiếp tục xem xét về Comparator.

2. Comparators

Comparator trong Java 8 không chỉ tối tân ở sự tương thích với các biểu thức lambda. Nó có những phương thức kết hợp với các tham chiếu phương thức, giúp cho mã trong sáng hơn rất nhiều:

```
list.sort(Comparator.comparingInt(String::length));
```

Bạn thậm chí có thể lấy các kết quả theo thứ tự ngược lại, tất cả chỉ bằng cách sử dụng thêm một phương thức hỗ trợ khác:

```
list.sort(Comparator.comparingInt(String::length).reversed());
```

3. Các class không trạng thái

Thường thì bạn sẽ hay gặp các lớp không có gì khác ngoài các phương thức tĩnh (chúng thường được đặt tên được tận cùng là "Util" hay "Helper"), chúng có nhiệm vụ gom nhóm các phương thức lại trong một không gian tên duy nhất. Không ai cần phải tạo đối tượng của các class này, và kể cả khi có làm việc đó thì các đối tượng này cũng không lưu giữ bất cứ dữ liệu riêng nào, do đó các class này được gọi là class không trạng thái (stateless).

Với Java 8, các interface có khả năng chứa các phương thức tĩnh, và trở thành lựa chọn tốt hơn so với class, bởi chúng ta không phải lo có ai đó tạo đối tượng của interface. Ví dụ kinh điển lần nữa lại là Comparator - với các phương thức tĩnh hữu dụng và mạnh mẽ của nó.

Tương tự như thế, nếu bạn gặp một class trừu tượng không trạng thái, chỉ có duy nhất những phương thức trừu tượng được thiết kế để ghi đè, lớp đó có thể được chuyển đổi thành những FunctionalInterface và sau đó được implement bằng biểu thức lambda, như Runnable chẳng hạn:

```
new Thread(new Runnable() {
    @Override
    public void run() {
      System.out.println("New thread created");
    }
  }).start();
Java 8:
new Thread(() -> {
    System.out.println("New thread created");
}).start();
```

Ví dụ

Thử xem xét cách triển khai, sử dụng annotation @FunctionalInterface cho interface:

```
@FunctionalInterface
interface Square {
  int calculate(int x);
}
```

... và implement bằng biểu thức lambda:

```
int a = 5;
Square s = (x) -> x * x;
int ans = s.calculate(a);
System.out.println(ans);
```

4. Các chỉ dẫn lặp và rẽ nhánh lồng nhau

Chúng ta có bộ API Streams được thiết kế để truy cập các collection theo cách vô cùng uyển chuyển. Khi gặp mã nguồn như dưới đây:

```
List<Field> validFields = new ArrayList<Field>();
for (Field field : fields) {
  if (meetsCriteria(field)) {
    validFields.add(field);
  }
}
return validFields;
```

...bạn phải nghĩ tới việc thay thế bằng mã sử dụng Streams. Ở đây ta dùng filter và collect:

```
return fields.stream()
  .filter(this::meetsCriteria)
  .collect(Collectors.toList());
```

Đôi khi, các vòng lặp với một lệnh rẽ nhánh ở trong có thể được tái cấu trúc thành anyMatch hay findFirst:

```
for (String current : strings) {
  if (current.equals(wanted)) {
    return true;
  }
}
return false;
```

...có thể được thay thế bởi:

```
return strings.stream()
  .anyMatch(current -> current.equals(wanted));
Và:
for (String current : strings) {
  if (current.equals(wanted)) {
    return current;
  }
}
return null;
```

...có thể sửa thành:

```
return strings.stream()
    .filter(current -> current.equals(wanted))
```

```
    .findFirst()
    .orElse(null);
```

Lưu ý rằng orElse(null) thực tế là một mã xấu, chúng ta sẽ nói đến sau.

5. Đa thao tác trên collection

Mặc dù đã cố gắng tối ưu hóa, nhưng chúng ta vẫn thường xuyên gặp trường hợp một chuỗi các thao tác được thực hiện trên một hay nhiều collection để có được kết quả mong muốn.

Ví dụ

```
// collect messages for logging
List<LogLine> lines = new ArrayList<>();
for (Message message : messages) {
  lines.add(new LogLine(message));
}
// sort
Collections.sort(lines);
// log them
for (LogLine line : lines) {
  line.log(LOG);
}
```

Chia để trị đã giúp cho mã được rõ ràng, nhưng chưa nói tới việc phải sử dụng tới comment để làm rõ ý, thì lời gọi Collections.sort vẫn gợi ý rằng chúng ta có thể sử dụng Streams tại đây. Trong thực tế, toàn bộ đoạn mã trên có thể được gộp vào một stream duy nhất:

```
messages.stream()
    .map(LogLine::new)
    .sorted()
    .forEach(logLine -> logLine.log(LOG));
```

Tái cấu trúc này không chỉ giúp mã dễ đọc hơn hay ít lại một biến trung gian, mà thực tế còn có hiệu năng cao hơn.

6. Duyệt để xóa bỏ các phần tử

Mã trước-Java-8 có thể có những đoạn mã như sau:

```
Iterator<String> iterator = strings.iterator();
while (iterator.hasNext()) {
  String current = iterator.next();
  if (current.endsWith("foo bar")) {
    iterator.remove();
  }
}
```

Giờ đây đoạn mã trên có thể được rút ngắn còn một dòng:

```
strings.removeIf(current -> current.endsWith("foo bar"));
```

Ngắn hơn, dễ đọc hơn, và cũng nhanh hơn!

7. Kiểm Null

NullPointerExceptions là điểm chí tử của các nhà phát triển Java, và bạn sẽ không lấy làm lạ khi các phép kiểm null nằm rải rác trong mã. Java 8 cho chúng ta API Optional giúp chúng ta mô tả kết quả trả về tốt hơn nhiều cũng như loại bỏ các kiểm null không cần thiết. Hãy quay lại với orElse mà chúng ta có ở tái cấu trúc số 4:

```
public static String findString (String wanted){
  List<String> strings = new ArrayList<>();
  return strings.stream()
      .filter(current -> current.equals(wanted))
      .findFirst()
      .orElse(null);
}
```

Bất kỳ lời gọi findString nào cũng sẽ phải kiểm null cho giá trị nhận được, và nếu an toàn thì làm một thao tác nào đó:

```
String foundString = findString(wantedString);
if (foundString == null) {
  return "Did not find value" + wantedString;
} else {
  return foundString;
}
```

Mã này xấu, dễ lặp và tẻ nhạt. Nếu cập nhật phương thức findString thành sử dụng Optional:

```
public static Optional<String> findString(String wanted) {
  List<String> strings = new ArrayList<>();
  return strings.stream()
      .filter(current -> current.equals(wanted))
      .findFirst();
}
```

...thì chúng ta có thể thoát khỏi trường hợp không có giá trị một cách thanh nhã hơn nhiều:

```
return findString(wantedString).orElse("Did not find value" +
wantedString);
```

> ***Mã xấu là gì?***
>
> *Mã xấu là từ được dùng để chỉ phần code mà ta cảm thấy không ổn. Đây thường là đoạn code vi phạm những quy tắc trong lập trình. Hãy xem qua ví dụ sau: bạn đang đọc một bài viết và bắt gặp một lỗi chính tả. Ngay lập tức, bạn có một cảm giác khó chịu trong người. Khi xem code, ta có những phản ứng tương tự và biết ngay chỗ đó không ổn. Đó chính là mã xấu.*
>
> ***Nguyễn Bình Sơn***

Tìm hiểu về GIẢI THUẬT Một số phương pháp sắp xếp cơ bản

1. Sắp xếp kiểu lựa chọn (Selection Sort)

Một trong những phương pháp đơn giản nhất để thực hiện sắp xếp một bảng khóa là dựa trên phép lựa chọn.

Nguyên tắc cơ bản của phương pháp sắp xếp này là "ở lượt thứ i(i=1,2,...,n) ta sẽ chọn trong dãy khoá Ki, Ki+1,...,Kn khoá nhỏ nhất và đổi chỗ nó với Ki ".

Như vậy thì rõ ràng là sau j lượt, j khoá nhỏ hơn đã lần lượt ở các vị trí thứ nhất, thứ hai,..., thứ j theo đúng thứ tự sắp xếp.

Ví dụ

Sắp xếp dãy số sau theo thứ tự tăng dần: "42, 23, 74, 11, 65, 58, 94, 36, 99, 87".

i	K_i	K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8	K_9
1	42	11	11	11	11	11	11	11	11	11
2	23	23	23	23	23	23	23	23	23	23
3	74	74	74	36	36	36	36	36	36	36
4	11	42	42	42	42	42	42	42	42	42
5	65	65	65	65	65	58	58	58	58	58
6	58	58	58	58	58	65	65	65	65	65
7	94	94	94	94	94	94	94	74	74	74
8	36	36	36	74	74	74	74	94	87	87
9	99	99	99	99	99	99	99	99	99	94
10	87	87	87	87	87	87	87	87	94	99

Sau đây là giải thuật:

Cho dãy khóa K gồm n phần tử. Giải thuật này thực hiện sắp xếp các phần tử của K theo thứ tự tăng dần dựa vào phép chọn phần tử nhỏ nhất trong mỗi lượt.

```
Procedure SELECT-SORT(K, n)
   For i:=1 to n-1 do
        Begin
        M:=i;
        For j:=i+1 to n do
        If K[j]<K[m] then m:=j;
             If m!= j then
             Begin {đổi chỗ}
                  X:=K[i];
                  K[i]:=K[m];
                  K[m]:=X;
             End
        End
   Return;
```

2. Sắp xếp chèn (Insertion Sort)

Nguyên tắc sắp xếp ở đây dựa theo kinh nghiệm của những người chơi bài. Khi có i-1 lá bài đã được sắp xếp ở trên tay, nay rút thêm lá bài thứ i nữa thì sắp xếp lại như thế nào ? Có thể so sánh lá bài mới lần lượt với lá bài thứ (i-1), thứ (i-2)... để tìm ra chỗ thích hợp và chèn nó vào chỗ đó.

Dựa trên nguyên tắc này, có thể triển khai một cách sắp xếp như sau:

Thoạt đầu K1 được coi như bảng chỉ gồm có một khoá đã sắp xếp. Xét thêm K2, so sánh nó với K1 để xác định chỗ chèn nó vào, sau đó ta sẽ có một bảng gồm 2 khoá đã được sắp xếp. Đối với K3 lại so sánh với K2, K1 và cứ tương tự như vậy với K4, K5, K6,... cuối cùng sau khi xét xong Kn thì bảng khoá đã được sắp xếp hoàn toàn.

Ta thấy ngay phương pháp này rất thuận lợi khi các khoá của dãy được đưa dần vào miền lưu trữ. Đó cũng chính là không gian nhớ dùng để sắp xếp. Có thể minh hoạ qua bảng:

i	1	2	3	4	5	6	7	8	9	10
Khoá đưa vào	42	23	74	11	65	58	94	36	99	87
1	42	23	23	11	11	11	11	11	11	11
2		42	42	23	23	23	23	23	23	23
3			74	42	42	42	42	36	36	36
4				74	65	58	58	42	42	42
5					74	65	65	58	58	58
6						74	74	65	65	65
7							94	74	74	74
8								94	94	87
9									99	94
10										99

Nhưng nếu các khóa đã có mặt ở bộ nhớ trong trước lúc sắp xếp thì sao ?

Sắp xếp vẫn có thể thực hiện được ngay tại chỗ chứ không phải chuyển sang một miền sắp xếp khác. Lúc đó các khoá cũng lần lượt được xét tới và việc xác định chỗ cho khoá mới vẫn làm tương tự, chỉ có khác là: để dành chỗ cho khoá mới nghĩa là phải dịch chuyển một số khoá lùi lại sau, ta không có sẵn chỗ trống như trường hợp nói trên (vì khoá đang xét và các khoá sẽ được xét đã chiếm các vị trí đằng sau này rồi), do đó phải đưa khoá mới này ra một chỗ nhớ phụ và sẽ đưa vào vị trí thực của nó sau khi đã đẩy các khóa cần thiết lùi lại.

Sau đây là giải thuật ứng với trường hợp này:

```
Procedure INSERT-SORT(K,n)
```

{Trong thủ tục này người ta dùng X làm ô nhớ phụ để chứa khoá mới đang được xét. Để đảm bảo cho khoá mới trong mọi trường hợp, ngay cả khi vị trí thực của nó là vị trí đầu tiên, đều được chèn vào giữa khóa nhỏ hơn nó và khoá lớn hơn nó, ở đây đưa thêm vào một khoá giả K0, có giá trị nhỏ hơn mọi khoá của bảng, và đứng trước mọi khoá đó. Ta quy ước K0 = - }.

```
K[0]= -
For i:=2 to n do
Begin
   X:=K[i];
   J:=i-1;
```

{xác định chỗ cho khoá mới được xét và dịch chuyển các khóa cần thiết }

```
While x<K[j] do
        Begin
             K[j+1]:=K[j];
             J:=j-1;
        End;
{đưa X vào đúng chỗ}
   K[j+1]:=X;
End;
Return;
```

Bảng ví dụ minh hoạ tương ứng với các lượt sắp xếp theo giải thuật này, tương tự như bảng đã nêu ở trên, chỉ có khác là không có chỗ nào trống trong miền sắp xếp cả, vì những chỗ đó đang chứa các khoá chưa được xét tới trong mỗi lượt (người đọc có thể tự lập ra bảng minh hoạ này).

3. Sắp xếp kiểu đổi chỗ (exchange sort)

Trong các phương pháp sắp xếp nêu trên, tuy kĩ thuật đổi chỗ đã được sử dụng, nhưng nó chưa trở thành một đặc điểm nổi bật. Bây giờ ta mới xét tới phương pháp mà việc đổi chỗ một cặp khoá kế cận, khi chúng ngược thứ tự, sẽ được thực hiện thường xuyên cho tới khi toàn bộ bảng các khóa đã được sắp xếp. Ý cơ bản có thể nêu như sau:

Bảng các khóa sẽ được duyệt từ đáy lên đỉnh. Dọc đường, nếu gặp hai khoá kế cận ngược thứ tự thì đổi chỗ chúng cho nhau. Như vậy trong lượt đầu khoá có giá trị nhỏ nhất sẽ chuyển dần lên đỉnh. Đến lượt thứ hai khoá có giá trị nhỏ thứ hai sẽ được chuyển lên vị trí thứ hai... Nếu hình dung dãy khoá được đặt thẳng đứng thì sau từng lượt sắp xếp, các giá trị khoá nhỏ sẽ nổi dần lên giống như các bọt nước nổi lên trong nồi nước đang sôi. Vì vậy phương pháp này thường được gọi bằng cái tên khá đặc trưng là: sắp xếp kiểu nổi bọt (bubble sort).

Giải thuật này rõ ràng còn có thể cải tiến được nhiều. Chẳng hạn, xét qua ví dụ ở trên ta thấy: sau lượt thứ 3 không phải chỉ có ba khóa 11, 23, 36 vào đúng vị trí sắp xếp của nó mà là 5 khoá. Còn sau lượt thứ 4 thì tất cả các khóa đã nằm đúng vào vị trí của nó rồi. Như vậy nghĩa là năm lượt cuối không có tác dụng gì thêm cả. Từ đó có thể thấy: nếu nhớ được vị trí của khoá được đổi chỗ cuối cùng ở mỗi lượt thì có thể coi đó là giới hạn cho việc xem xét ở lượt sau.

Ví dụ

i	K_i	lượt	1	2	3	4	5	6	7	8	9
1	42		11	11	11	11	11	11	11	11	11
2	23		42	23	23	23	23	23	23	23	23
3	74		23	42	36	36	36	36	36	36	36
4	11		74	36	42	42	42	42	42	42	42
5	65		36	74	58	58	58	58	58	58	58
6	58		65	58	74	65	65	65	65	65	65
7	94		58	65	65	74	74	74	74	74	74
8	36		94	87	87	87	87	87	87	87	87
9	99		87	94	94	94	94	94	94	94	94
10	87		99	99	99	99	99	99	99	99	99

Sau đây là giải thuật:

```
Procedure BUBBLE-SORT(K,n)
For i:=1 to n-1 do
   For j:=n down to i+1 do
        If K[j]<K[j-1] then
        Begin
             X:=K[j];
             K[j]:=K[j-1];
             K[j-1]:=X;
        End;
Return;
```

Chừng nào mà giới hạn này chính là vị trí thứ n, nghĩa là trong lượt ấy không có một phép đổi chỗ nào nữa thì sắp xếp có thể kết thúc được. Nhận xét này sẽ dẫn tới một giải thuật cải tiến hơn, chắc chắn có thể làm cho số lượt giảm đi và số lượng các phép so sánh trong mỗi lượt cũng giảm đi nữa. Người đọc hãy tự xây dựng giải thuật theo ý cải tiến này.

Nguyễn Khánh Tùng

(Tham khảo: Cấu trúc dữ liệu và giải thuật - Đỗ Xuân Lôi - NXB Đại học quốc gia Hà Nội).

Vi-rút Corona
ảnh hưởng đến ngành công nghệ như thế nào?

Mảng công nghệ đang bị ảnh hưởng rất nặng nề của vi-rút Corona, trong đó các văn phòng, cửa hàng và nhà máy phải đóng cửa ở Trung Quốc cũng như nhân sự khó quay trở lại làm việc, trong và ngoài nước.

Việc chờ đợi tình hình trở nên tốt đẹp hơn sau Tết Nguyên Đán, nhưng có vẻ như vi rút Corona đã trở nên khó kiểm soát hơn, dẫn đến tình trạng trì trệ kéo dài cho đến nay.

Không có gì là chắc chắn cho đến thời điểm này, các sự kiện bị trì hoãn, các công ty không xác định được khi nào sẽ quay trở lại kinh doanh. Sau đây là một số ảnh hưởng mà vi rút Corona đã gây ra cho toàn bộ ngành công nghiệp công nghệ tính đến thời điểm này.

Những công ty đã đóng cửa văn phòng tại Trung Quốc tính đến thời điểm này là Apple, Samsung, Microsoft, Tesla và Google.

Ngoài ra, Google cũng đóng những văn phòng gần với Hong Kong và Đài Loan. Một số văn phòng được biết là sẽ đóng cửa đến khoảng ngày 9.2.2020, nhưng đến nay thì vẫn chưa có động tĩnh mới về việc mở cửa lại.

Giám đốc nhân sự của Apple, Deirdre O'Brien, chia sẻ trong một email nội bộ rằng, văn phòng và những trung tâm liên lạc được dự đoán sẽ mở lại vào "tuần tới", trong khi các cửa hàng

bán lẻ vẫn chưa có ngày khai trương. "Vì vi rút Corona nên những vấn đề liên quan đến dọn dẹp, quy trình sức khỏe và giới hạn nội bộ xung quanh những không gian công cộng bổ sung vẫn đang được xem xét", Deirdre cho biết. Nhưng có vẻ như "tuần tới" đến thời điểm này là sau cả ngày 15.2.

Amazon chưa công bố về việc đóng cửa văn phòng tại Bắc Kinh, Thâm Quyến, Thượng hải và Quảng Châu (công ty này không có văn phòng tại Vũ Hán), nhưng yêu cầu nhân viên cần phải được cho phép trước khi di chuyển đến Trung Quốc. Các nhân viên đã du lịch vào và ra Trung Quốc cũng được phép làm việc tại nhà trong 14 ngày, để theo dõi tình trạng lây nhiễm vi rút Corona.

Facebook không có văn phòng tại Trung Quốc, nhưng cũng đã nhanh chóng cảnh báo nhân viên của mình về việc đi du lịch đến đây. LG và Razer cũng khuyên tương tự. Và khách du lịch có thể vào được Trung Quốc hay không vẫn là một câu hỏi lớn khi mà hơn 50 hãng hàng không đã cấm bay vào và ra đất nước này cho đến khi tình hình về Coronavirus được kiểm soát.

Sản xuất bị hạn chế và ra mắt bị hoãn

Trong khi nhiều sản phẩm được sản xuất tại Trung Quốc (hoặc sử dụng một phần trong các nhà cung cấp tại đây), các chuyên gia đã cảnh báo người dùng rằng sẽ có sự thiếu hụt trong một số sản phẩm điện thoại, thiết bị VR, xe hơi và các phụ kiện công nghệ khác. Foxconn và Pegatron đóng cửa ở Trung Quốc, và sẽ là nguyên nhân dẫn đến thiếu hụt iPhone và AirPod, cho đến khi các nhân viên trở lại sau dịch Covid-19.

Vào ngày 8.2 vừa qua, Trung Quốc đã không cho Foxconn mở cửa nhà máy tại Thâm Quyến với các điều kiện làm việc và sinh hoạt như hiện tại, và Foxconn cũng có ý định không mở cửa ở Trịnh Châu cho đến khi chính phủ xem xét. Nhà máy Trịnh Châu, được biết đến với tên gọi "thành phố iPhone", là nơi lắp ráp số lượng lớn các iPhone. Một số báo cáo cho thấy nhà máy tại đây sẽ làm việc lại sớm, nhưng với số lượng nhân công ít ỏi không về quê trước dịp Tết, thời điểm bùng nổ Covid-19.

Facebook cũng đã không nhận đặt hàng các thiết bị Oculus Quest VR, do sự chậm trễ trong các hoạt động nhà máy. Công ty đã chia sẻ: "Cũng giống như những công ty khác, chúng tôi biết đang có các ảnh hưởng lớn lên kế hoạch sản xuất thiết bị do vi rút Corona. Chúng tôi đang rất cẩn trọng trong việc đảm bảo sự an toàn của các nhân viên, đối tác sản xuất và khách hàng, và quản lý tình huống một cách chặt chẽ. Chúng tôi đang làm việc để phục hồi sớm nhất có thể".

Quest đã ngưng nhận đặt hàng từ trước dịp lễ, các nhà bán lẻ cho biết các thiết bị sẽ sớm có lại vào đầu tháng 2, nhưng cuối cùng cũng phải dời ngày bán ra vào 10.3 cho phiên bản 64GB.

Và đối với ROG Phone II, Asus cũng thông báo cho khách hàng rằng thiết bị sẽ không xuất hiện cho đến khi có thêm thông tin về vi rút Corona.

Với việc đóng cửa nhà máy, Tesla cũng chậm trễ trong việc bán ra Model 3, mặc dù các dự đoán cho thấy chỉ trễ hơn so với dự định khoảng 1 đến 1 tuần rưỡi. Nintendo Switch cũng không phải là ngoại lệ.

Xáo động sự kiện

Sự ảnh hưởng của vi rút Corona mạnh hơn tưởng tượng, nhiều công ty công nghệ cũng đã rút lui khỏi các sự kiện quốc tế như Mobile World Congress (MWC) ở Barcelona. Sony, Amazon, Nvidia, LG Electronics, ZTE và Ericsson đã có động thái chấm dứt hoạt động của mình tại sự kiện. Nhưng trong một bất ngờ mới nhất, toàn bộ sự kiện MWC 2020 sẽ không diễn ra vào ngày 24/2 tới.

Giải đấu Overwatch League của Blizzard trong tháng 2 và 3 cũng đã bị hủy. Vòng loại giải thế giới của PUBG vào tháng 4 tới đây ở Berlin cũng bị trì hoãn cho đến khi có thêm thông báo mới về Covid-19.

Vi rút Corona hiện đã có số lượng người nhiễm và chết cũng như chữa khỏi tăng cao mỗi ngày. Vì thế, tình hình hiện tại của các hãng công nghệ đều bị đình trệ, và hầu như không ai biết trước được sự trì trệ này sẽ kéo dài tới bao lâu.

Coronavirus *cũng được gọi là virus corona, là một nhóm gồm các loại virus thuộc phân họ Coronavirinae trong Họ Coronaviridae, theo Bộ Nidovirales. Coronavirus gây bệnh ở các loài động vật có vú, bao gồm cả con người và chim. Ở người, virus gây nhiễm trùng đường hô hấp thường là nhẹ nhưng trong trường hợp ít gặp có thể gây tử vong.*

Vào tháng 12 năm 2019, một loại coronavirus mới là SARS-CoV-2 gây nên dịch bệnh viêm phổi, bắt nguồn từ thành phố Vũ Hán của Trung Quốc và lan rộng sang một số nơi khác.

Tôn Bảo - Báo Thanh niên

> ***CodeGym Bob*** *là một nền tảng hỗ trợ các lập trình viên học, luyện tập thuật toán và kỹ năng giải quyết vấn đề. Các bài luyện tập trong CodeGym Bob được thiết kế để dẫn dắt và nâng kỹ năng của lập trình viên theo từng cấp độ. Với hàng trăm thành viên đang luyện tập trên nền tảng này, CodeGym Bob đang khẳng định tính hữu ích của mình đối với các lập trình viên vừa mới bước vào nghề. Nền tảng CodeGym Bob được cung cấp hoàn toàn miễn phí.*

CodeGym Bob

Ứng dụng luyện thuật toán dành cho người mới bắt đầu

Các tính năng chính của CodeGym Bob bao gồm:

- Bài học: Chuỗi các bài học được chia làm 11 mức độ nối tiếp nhau, giúp cho người học đi từ những khái niệm cơ bản nhất cho đến những bài toán giải quyết vấn đề.
- Luyện tập: Chuỗi các bài luyện tập giúp học viên thực hành kỹ năng giải quyết vấn đề và áp dụng các thuật toán thông dụng trong những bài toán thường gặp.
- Bài kiểm tra: Bài kiểm tra ngắn là bước cuối cùng để học viên hoàn thành các nội dung luyện tập, giúp học viên biết được tình trạng của mình sau một thời gian luyện tập.

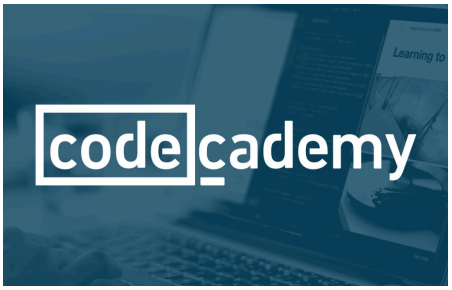
Hiện nay, CodeGym Bob đã hỗ trợ các ngôn ngữ lập trình phổ biến là JavaScript, PHP và Java.

Giao diện màn hình của ứng dụng CodeGym Bob

Nguồn: CodeGym Việt Nam

TRANG WEB HỌC LẬP TRÌNH ONLINE TỐT NHẤT HIỆN NAY

Trang web học lập trình online đang trở thành một trong những nơi học lập trình rất tốt cho nhiều bạn. Nơi mà các bạn có thể tìm thấy rất nhiều thứ hay về lập trình. Hơn nữa tất cả các kiến thức đều được chia sẻ một cách miễn phí và các kiến thức cập nhật thương xuyên. Trong bài viết này tôi sẽ chia sẻ với các bạn một vài trang web học lập trình online tốt nhất hiện nay.



Code Academy

Trang web đầu tiên mà chúng tôi muốn giới thiệu với các bạn đó chính là CodeAcademy. Đây là một trang web học lập trình rất phù hợp cho những bạn mới bắt đầu tìm hiểu về lập trình. Bạn cũng có thể tự học lập trình ở trang web này với những thứ cơ bản và miễn phí như HTML & CSS , JavaScript , jQuery , PHP , Python và Ruby.

Có hơn 300 giờ nội dung trực tuyến có sẵn miễn phí cho hàng triệu cá nhân trên toàn cầu. Các khóa học có sẵn sẽ có thể giúp bất kỳ người nào muốn học về lập trình. Và mỗi khóa học đều có thời gian dự kiến để bạn hoàn thành. Điều này giúp bạn có thể lên kế hoạch và mục tiêu học tập cho mình một cách tốt hơn,

Tuy nhiên những bài học ở trang web này chỉ dừng lại ở mức cơ bản. Chính vì vậy mà nó phù hợp với những người mới với lập trình. Còn nếu bạn đã học chuyên sâu và muốn tìm những gì nó nâng cao thì nên đến với các trang web lập trình khác.

Coursera

Coursera được thành lập từ năm 2012. Hiện nay trang web này đang cũng cấp hơn 1000 khóa học từ 119 tổ chức khác nhau. Các khóa học về lập trình của Coursera thì không hoàn toàn miễn phí có một số khóa học lập trình bạn phải trả phí để học. Các khóa học kéo dài từ 4 đến 6 tuần và thường có giá từ khoảng $ 29 đến $ 99. Nó không chỉ trực tuyến mà còn có sẵn cho tất cả mọi người có quyền truy cập vào máy tính. Nhưng bù lại bạn sẽ được nhận chứng chỉ cho khóa học đó.

Trang web vẫn có rất nhiều các khóa học miễn phí dành cho dân lập trình. Và các khóa học này đến từ các trường đại học như Đại học Washington, Stanford, Đại học Toronto và Vanderbilt. Mỗi khóa học trên Coursera đều được giảng dạy bởi các tổ chức giáo dục tốt nhất thế giới. Và được tạo thành từ các bài giảng video, diễn đàn thảo luận cộng đồng. Chính vì vậy đây là một trang web học lập trình online bạn nên tìm hiểu và tham gia để học lập trình tốt hơn.

W3school

W3Schools là trang web học lập trình online đã quá nổi tiếng với đân lập trình. Đây là một trang web giáo dục để tìm hiểu các công nghệ web trực tuyến. Nội dung bao gồm các hướng dẫn và tài liệu tham khảo liên quan đến HTML , CSS , JavaScript ,JSON , PHP , Python. Ngoài ra con có về AngularJS, SQL , Bootstrap , Node.js, jQuery , XQuery , AJAX và XML. Bạn có thể tìm thấy bất cứ thứ gì trên trang web này về lập trình.

Nó phù hợp với cả người mới bắt đầu và những người đã có kinh nghiệm. Các kiến thức trong W3schools được sắp xếp rất hợp lý. Phần chia theo từng mảng kiến thức từ cơ bản đến nâng cao thành từng bài. Vì vậy mà đây là một trong những trang we không thể bỏ quá với bất kỳ dân lập trình nào.

edX

EdX là một nền tảng học trực tuyến hàng đầu. Nó được thành lập vào năm 2012 bởi MIT và Harvard. Trang web này hoàn toàn là nguồn mở thay vì lợi nhuận. EdX có hơn 90 đối tác trên khắp thế giới và bao gồm các tổ chức phi lợi nhuận, trường đại học và tổ chức hàng đầu thế giới. Vì vậy bạn sẽ được tìm hiểu các kiến thức về lập trình tiên tiến nhất. Học sinh có thể chọn các khóa học của họ từ khoảng 60 trường. Những người đăng ký các khóa học được đặt ở khắp nơi trên thế giới. EdX được thành lập và quản lý bởi các trường đại học và cao đẳng.

Udemy

Được thành lập vào năm 2010, Udemy là một nền tảng học tập trực tuyến có thể được sử dụng như một cách để cải thiện hoặc học các kỹ năng công việc. Mặc dù có những khóa học bạn phải trả tiền. Nhưng có rất nhiều khóa học lập trình miễn phí, được dạy qua các bài học video. Chẳng hạn như khóa học có tên là Lập trình cho Doanh nhân – HTML & CSS hoặc Giới thiệu về Lập trình Python.

Đây là một thị trường toàn cầu để dạy và học trực tuyến. Nơi sinh viên sẽ thành thạo các kỹ năng mới và đạt được mục tiêu của mình. Thông qua việc chọn từ một thư viện gồm hơn 45.000 khóa học. Bạn sẽ được giảng dạy bởi các giảng viên là chuyên gia trong lĩnh vực của họ. Bạn có thể tham khảo và chọn các khóa học ở đây để tham khảo. Và bạn nên chọn những khóa học được nhiều học viên đánh giá 5 sao. Bởi nó có thể là những khóa học chất lượng và phù hợp với bạn.

Khan Academy

Khan Academy là một trong những tổ chức học trực tuyến miễn phí ban đầu. Với các hướng dẫn bằng video, bạn có thể tìm hiểu cách lập trình bản vẽ, hoạt hình và trò chơi bằng JavaScript và Treatment-JS. Hoặc tìm hiểu cách tạo trang web bằng HTML và CSS. Trang web này cung cấp giáo dục dưới dạng

các bài tập thực hành video hướng dẫn.

Có hàng triệu sinh viên từ khắp nơi trên thế giới. Với những câu chuyện độc đáo của họ được học thông qua Khan Academy. Các tài nguyên có thể được dịch sang hơn 36 ngôn ngữ. Khi bạn đến với trang web này bạn không chỉ được học kiến thức. Mà bạn còn có thể tìm kiếm được các kinh nghiệm về lập trình cho bản thân mình.

Code Avengers

Code Avengers cung cấp các khóa học lập trình thú vị. Nó hướng dẫn bạn cách viết mã trò chơi, ứng dụng và trang web bằng JavaScript, HTML và CSS. Mỗi khóa học chỉ mất 12 giờ để hoàn thành. Nó có sẵn bằng tiếng Anh, tiếng Nga, tiếng Hà Lan, tiếng Tây Ban Nha. Ngoài ra còn có tiếng Ý, tiếng Thổ Nhĩ Kỳ và tiếng Bồ Đào Nha.

Đây là một trang web vừa học vừa giải trí rất tốt cho dân lập trình. Bạn sẽ được tham gia các thử thách về code, các câu đố và trò chơi về lập trình. Tại đây bạ sẽ có cơ hội giao lưu với rất nhiều các học viên khác. Bạn sẽ có thêm nhiều kinh nghiệm và nhiều người bạn để chia sẻ kiến thức với mình hơn.

HTML5 Rocks

HTML5 Rocks ra mắt vào năm 2010, bao gồm rất nhiều các tài nguyên về HTML5 phiên bản mới nhất. Nó là nguồn mở. Đây là trang web phù hợp với những người có kinh nghiệm hơn. Nó không phải là một nơi tuyệt vời cho người mới bắt đầu. Người dùng có thể tìm kiếm thông qua trang web để tìm kiếm chính xác những gì họ đang tìm. Hoặc họ có thể duyệt qua tất cả các hướng dẫn. Các hướng dẫn được tác giả bởi nhiều cá nhân chuyên về các lĩnh vực phát triển web và ngôn ngữ mã khác nhau.

Với những trang web lập trình online trên mong rằng các bạn có thể tìm ra cho mình những khóa học phù hợp với bản thân. Hơn hết đây còn là môi trường tốt để bạn có thể tìm kiếm tài liệu và làm quen với những người bạn mới. Đó có thể tạo ra nhiều cơ hội cho bạn hơn trong học tập và làm việc.

Chúc các bạn học tập tốt.

5 DỰ ÁN "LÀM ĐỂ HỌC"
ĐỂ HOÀN THIỆN HƠN NĂNG LỰC LẬP TRÌNH

Là một kỹ sư phần mềm, có thể vào một thời điểm nào đó, bạn sẽ cảm thấy chán công việc vì những thứ nhàm chán lặp lại mỗi ngày. Bạn cảm thấy rằng mình không còn được học hỏi gì nhiều từ môi trường làm việc, từ sản phẩm đang xây dựng ở công ty, từ đồng nghiệp, từ sếp,.. Có thể, bạn đang thiếu đi không gian học hỏi cho chính mình.

Nếu vẫn còn khao khát học hỏi và nâng cao kỹ năng nghề nghiệp thì bạn có thể chọn giải pháp thực hiện một dự án thú vị nào đó bên cạnh công việc. Đó nên là một dự án thử thách so với năng lực hiện tại của bạn.

Hãy chọn một dự án mà bạn yêu thích. Như Paul Graham, một chuyên gia - nhà khởi nghiệp nổi tiếng từng viết "Bạn không thể làm bất cứ điều gì thực sự tốt trừ khi bạn yêu nó, và nếu bạn thích vọc vạch, chắc chắn bạn sẽ làm điều đó trên dự án của riêng mình".

Hãy bắt tay vào thực hiện ngay nhé! Dưới đây là một số gợi ý mà bạn có thể tham khảo. Thoạt nhìn, những ý tưởng này có thể quen thuộc. Nhưng nếu bạn nghiêm túc tìm hiểu, chắc chắn bạn sẽ tìm thấy nhiều kiến thức thú vị ẩn sau những thứ "quen thuộc" như thế!

Dự án 1: Xây dựng web framework

Web framework là gì?

Web framework là bộ khung cơ bản để chúng ta có thể xây dựng nên các ứng dụng web hiệu quả hơn. Một framework có thể chứa bên trong nhiều thư viện và những cơ chế cần thiết cho ứng dụng web. Ví dụ: xử lý dữ liệu được trích xuất từ HTTP request, cơ chế điều hướng từ đường dẫn đến mã xử lý cụ thể, cơ chế kết nối và làm việc với những cơ sở dữ liệu thông dụng, cách cấu trúc và bố trí mã nguồn sao cho dễ hiểu và dễ bổ sung tính năng mới, cơ chế logging phục vụ debug...

Các lập trình viên ứng dụng web thường dựa vào framework hiện đại nào đó để họ có thể tập trung nhiều hơn vào việc xây dựng tính năng người dùng, thay vì phải phát triển lại những tính năng kỹ thuật lặp đi lặp lại của các ứng dụng.

Việc xây dựng một web framework sẽ giúp bạn có cái nhìn tổng quan về kiến trúc của một ứng dụng web. Bạn sẽ học được nhiều kiến thức về xử lý các yêu cầu HTTP và phản hồi lại kết quả cho trình duyệt, thiết kế thư viện logging, xử lý lỗi, học các design pattern, kết nối cơ sở dữ liệu, chuyển đổi dữ liệu...

Làm thế nào để thực hiện dự án?

Hãy bắt đầu với việc liệt kê những tính năng được hỗ trợ của một framework bạn đang dùng hoặc phổ biến trong một ngôn ngữ lập trình nào đó. Từ đó, bạn sẽ có cái nhìn tổng quát về những gì cần có trong framework. Bạn có thể dựa vào trải nghiệm của mình về các framework trước đây để liệt kê ra được những gì quan trọng nhất. Đó sẽ là bản thiết kế cơ bản và thú vị cho framework của riêng bạn.

Rất nhiều lập trình viên trên thế giới thích tự tạo một framework cho riêng mình để học hỏi; nếu may mắn, framework đó có thể là công cụ hỗ trợ đắc lực cho công việc của họ và lan tỏa đến các cộng đồng xung quanh.

Chỉ với một vài từ khóa theo mẫu "how to build a web framework in {{ngôn ngữ lập trình bạn thích}}", bạn sẽ tìm được hàng trăm bài viết về chủ đề này.

Dự án 2: Xây dựng web server

Web server là gì?

Web Server là một thành phần cơ bản trong mọi hệ thống liên quan tới ứng dụng web. Chắc hẳn những cái tên như Apache, Nginx,.. không quá xa lạ với các bạn lập trình ứng dụng web. Có thể bạn đã làm việc qua hàng chục dự án web khác nhau nhưng sẽ rất hiếm có dịp để bạn nhúng tay vào mã nguồn của thành phần cốt lõi đó.

Đây chắc chắn là một dự án hay ho vì qua đó bạn có thể học được rất nhiều về lập trình. Bạn sẽ phải học về giao thức truyền tải, lập trình mạng, phân luồng và rất nhiều kỹ thuật khác để có thể tự phát triển được một web server. Bạn sẽ hiểu được các thức mà web server nhận request từ trình duyệt và chuyển hướng tới ứng dụng của bạn để thực hiện những nghiệp vụ phức tạp bên trong.

Làm thế nào để thực hiện dự án?

Hãy bắt đầu bằng việc tìm hiểu TCP - một giao thức cơ bản mà đa phần các ứng dụng giao tiếp qua mạng đều sử dụng. Tiếp theo là giao thức HTTP. Giao thức này dựa trên TCP, được tạo ra để phục vụ cho việc truyền tải nội dung web. Bạn có thể xây dựng một web server đơn giản chỉ xử lý các yêu cầu HTTP qua hướng dẫn trong bài viết này "HTTP Server: Everything you need to know to Build a simple HTTP server from scratch".

Một ứng dụng web hiện đại thường được xây dựng dựa vào một ngôn ngữ lập trình hiện đại cùng với một framework "xịn sò" nào đó. Nếu bạn đang xây dựng các ứng dụng web bằng Java thì có thể đã từng nghe qua Spring, Struts,... Nếu bạn sử dụng Python thì có thể biết tới Django, Flask, Pyramid,...

Để web server của bạn có thể chuyển các yêu cầu HTTP này sang ứng dụng web dựa trên các framework hiện đại thì chúng ta cần tìm hiểu cơ chế giao tiếp trên các nền tảng cụ thể. Cụ thể hơn, nếu bạn muốn xây dựng web server cho các ứng dụng được viết bằng Java thì bạn cần tìm hiểu đặc tả Servlet (Servlet API). Nếu muốn làm việc với Python thì áp dụng một cơ chế khá phổ biến là Web Server Gateway Interface (WSGI).

Dự án 3:
Xây dựng chương trình quản lý cơ sở dữ liệu đơn giản

Cơ sở dữ liệu là gì?

Cơ sở dữ liệu (CSDL) là một hình thức lưu trữ dữ liệu phổ biến trong thế giới lập trình ứng dụng. Trình quản trị CSDL là dạng phần mềm phía máy chủ, cho phép các ứng dụng hoặc người dùng kết nối và thực hiện các thao tác với dữ liệu.

Là một lập trình viên, chắc chắn bạn đã từng nghe qua những cái tên như MySQL, MS SQL Server, Oracle SQL Server,... Đây là những hệ quản trị cơ sở dữ liệu nổi tiếng và được sử dụng phổ biến trên thế giới.

Tuy nhiên, không nhiều lập trình viên thực sự hiểu cơ sở dữ liệu làm việc như thế nào.

Ví dụ:

- Dữ liệu được lưu với định dạng gì?
- Vì sao chỉ có duy nhất một khóa chính cho mỗi bảng?
- Index được định dạng như thế nào?
- ...và nhiều câu hỏi sâu về bản chất khác nữa.

Việc xây dựng một chương trình quản trị cơ sở dữ liệu đơn giản sẽ giúp bạn học được cách thức lưu trữ dữ liệu, cơ chế lưu trữ index và khoá chính trong bảng, phân tích cú pháp ngôn ngữ truy vấn (SQL) thành những thao tác xử lý dữ liệu cụ thể, và nhiều kiến thức nền tảng khác.

Làm thế nào để thực hiện dự án?

Nếu chưa thành thạo về SQL, bạn có thể tìm hiểu nhanh qua một khoá học ngắn trên Internet, hoặc trên trang web W3Schools SQL. Tại đây, bạn sẽ được ôn tập và tìm hiểu những câu lệnh SQL cơ bản để có thể thực hiện được dự án.

Sau khi nắm được SQL cơ bản, bạn có thể tham khảo tài liệu tại đây: Let's Build a Simple Database để được hướng dẫn xây dựng những tính năng cơ bản cho trình quản lý cơ sở dữ liệu.

Dự án 4:
Xây dựng hệ thống Linux từ mã nguồn

Hệ điều hành nhân Linux là gì?

Linux là một nhân (kernel) hệ điều hành mã nguồn mở được phát triển từ những năm 1990. Khởi nguồn là một dự án "làm cho vui" của "Linus Torvald". Sau đó, nó đã phát triển thành một thành phần không thể thiếu của dòng hệ điều hành nguồn mở tương-tự-Unix.

Đây là một trong số những dự án rất nhiều kỹ sư trên thế giới yêu thích. Việc tự xây dựng lại một hệ điều hành với nhân Linux sẽ giúp bạn hiểu rõ các thành phần cấu thành nên một hệ điều hành. Đây là là cơ hội để bạn tìm hiểu cách hệ thống Linux hoạt động từ trong ra ngoài, cách mọi thứ hoạt động cùng nhau và phụ thuộc vào nhau.

Hơn thế nữa, điều hay nhất mà trải nghiệm học tập này có thể mang lại là khả năng tùy chỉnh hệ thống Linux để phù hợp với nhu cầu riêng của bạn. Nó cho phép bạn có nhiều quyền kiểm soát hệ thống hơn mà không cần dựa vào các bản phân phối có sẵn. Bạn có thể tạo được các hệ thống Linux rất nhỏ gọn phục vụ những mục đích riêng.

Làm thế nào để thực hiện dự án?

Nếu bạn chưa từ sử dụng một hệ điều hành nhân Linux nào thì hãy cài một bản phân phối (hay còn gọi là phiên bản) phổ biến như Ubuntu, Fedora/Red Hat,... Sau khi hiểu về cấu trúc hệ thống của Linux,

bạn có thể hoàn toàn tự xây dựng nên một bản hệ điều hành của riêng bạn. Có rất nhiều hướng dẫn trên Internet về chủ đề này. Một trong số đó là trang linux-fromscratch.org.

Dự án 5: Xây dựng một công cụ kiểm thử tự động

Công cụ kiểm thử tự động là gì?

Công cụ kiểm thử tự động là những phần mềm giúp thực hiện các thao tác người dùng trên giao diện ứng dụng một cách tự động. Các nhóm xây dựng phần mềm thường áp dụng những công cụ này để nâng cao năng suất trong việc kiểm thử giao diện và tìm lỗi chức năng của ứng dụng.

Trên thực tế, có rất nhiều công nghệ xây dựng ứng dụng như web, di động, nhúng,... Mỗi loại công nghệ sẽ có những giải pháp khác nhau.

Đây là một dự án khá phức tạp. Bạn sẽ phải đọc khá nhiều tài liệu API về các nền tảng, tìm cách nhúng mã vào trình duyệt hoặc các ứng dụng đích để đọc thông tin về những thành phần đang được hiển thị trên giao diện. Thế nhưng, bạn sẽ học được nhiều về cách thức hệ điều hành xây dựng và quản lý giao diện ứng dụng, cách thức trình duyệt làm việc với mã HTML của trang web,...

Làm thế nào để thực hiện dự án?

Nếu bạn muốn xây dựng công cụ tự động cho ứng dụng web, bạn có thể tìm hiểu cơ chế thực thi của công cụ Selenium (một framework tự động hoá và mã nguồn mở cho web).

Nếu bạn xây dựng một ứng dụng desktop trên nền tảng .Net của Microsoft, bạn có thể tìm hiểu qua Microsoft UI Automation, một framework của Microsoft giúp xây dựng các hệ thống hỗ trợ tương tác với giao diện ứng dụng.

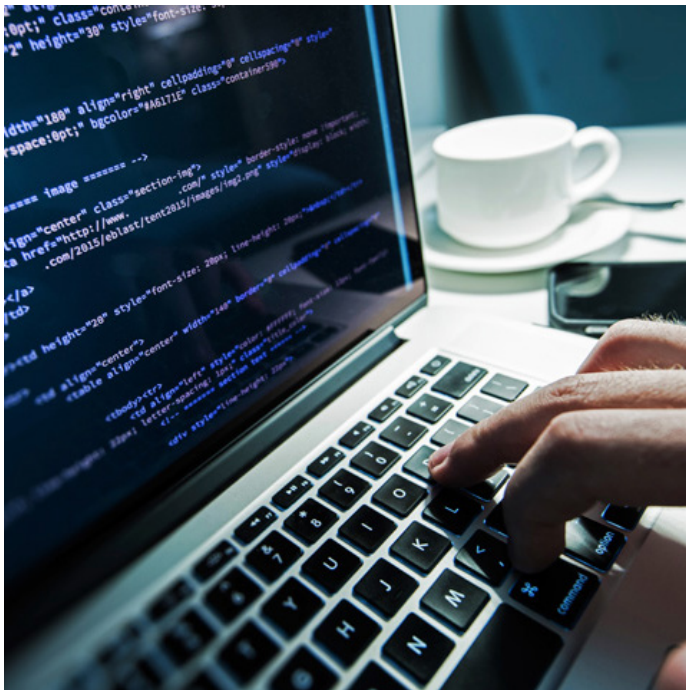
Tổng kết

Trên đây là 5 gợi ý những dự án mà các bạn có thể thực hiện để học thêm những điều thú vị, nâng cao kiến thức và kỹ năng cho sự nghiệp lập trình. Thế giới bên ngoài kia vẫn còn nhiều dự án thú vị khác nữa. Không bao giờ có giới hạn cho việc thử nghiệm và học hỏi. Miễn sao điều đó giúp xây dựng những kỹ năng có giá trị cho nghề nghiệp của bạn. Một khi đã chọn một ý tưởng, hãy thực hiện nó với lòng đam mê và sự nghiêm túc.

Việc duy trì thực hiện dự án liên tục trong một thời gian dài có thể là thử thách đối với nhiều người. Thông thường, chúng ta sẽ không thấy được thành quả ngay lập tức. Một thủ thuật để vượt qua được vấn đề này là hãy bắt đầu dự án với một thiết kế đơn giản. Dành một khung thời gian nhất định trong ngày để bổ sung tính năng. Bạn nên lên lịch tái cấu trúc sau một khoản thời gian nhất định. Và quan trọng là, đừng bao giờ ngại thử nghiệm những cái mới.

Chúc bạn sẽ luôn học được những điều mới mẻ!

Đặng Huy Hòa

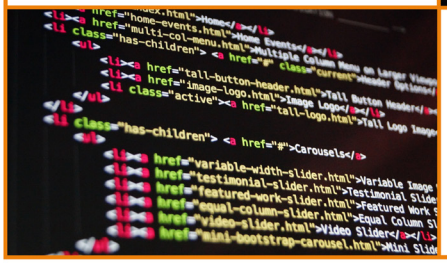


DEV CƯỜI

ĐOẠN CODE TÔI ĐƯỢC HỌC TRÊN TRƯỜNG	ĐOẠN CODE TÔI PHẢI VIẾT LÚC ĐI LÀM
	

Sự khác biệt giữa "bài tập thầy giao" và "bài tập sếp giao"

CÁCH MỌI NGƯỜI CHẠY ▶



◀ CÁCH LẬP TRÌNH VIÊN CHẠY

Người bình thường chạy và Dev chạy