PHẠM HUY HOÀNG

CODE DẠO KÍ SỰ
LẬP TRÌNH VIÊN ĐÂU PHẢI CHỈ BIẾT CODE

NHÀ XUẤT BẢN TRI THỨC

Trung 0335740004

Trung 0335740004

LỜI GIỚI THIỆU

Chào các bạn, mình là Phạm Huy Hoàng, chủ một blog IT mang tên Tôi đi code dạo.

Hiện nay, ngành IT nói chung và lập trình nói riêng đang trở thành một ngành hot, được khá nhiều bạn sinh viên lựa chọn. Tuy nhiên, so với nước ngoài, các bạn sinh viên Việt Nam chịu khá nhiều thiệt thòi vì thiếu những tấm gương và tài liệu để học hỏi. Thuở còn là sinh viên, mình cũng từng có những thắc mắc, trăn trở về kĩ thuật, về con đường nghề nghiệp, nhưng không có ai giải đáp.

Là một lập trình viên, các bạn cần học rất nhiều, nhưng không sách vở nào nói về cách tự học cho hiệu quả. Lập trình viên cần biết cách giao tiếp và làm việc nhóm, nhưng ít thầy cô nói cho các bạn biết điều này. Lập trình viên cần phải giỏi tiếng Anh, nhưng hầu như đi làm rồi các bạn mới tự nhận ra.

Không biết những điều này, bạn sẽ phải hứng chịu vô số gạch đá trên con đường nghề nghiệp. Do vậy, chúng ta cần những đầu sách định hướng nghề nghiệp và những kĩ năng phải có của người lập trình viên.

Tuy nhiên, đa phần sách cho dân IT hiện nay quá tập trung vào kĩ thuật và công nghệ (kĩ năng cứng), quên mất những kĩ năng mềm mà lập trình viên nên có. Những quyển sách trên cũng khá hàn lâm và khô cứng, khó tiếp thu.

Cuốn sách này không như thế! Vậy nó có gì hot?

Đây là cuốn sách duy nhất tập trung vào phần kĩ năng mềm mà mỗi lập trình viên cần có. Đi kèm với chúng những kĩ năng cứng được đúc kết qua kinh nghiệm bao năm làm việc của tác giả. Sách đảm bảo sẽ đưa bạn đọc từ mềm đến cứng.

Thay cho những chương sách dày cộm toàn chữ, nội dung sách được chia làm nhiều bài viết ngắn gọn, mỗi bài viết đề cập đến một khía cạnh khác nhau. Giọng văn ngắn gọn, hài hước dí dỏm, đọc không hề cứng nhắc như sách kĩ thuật mà lại rất dễ tiếp thu. Đoạn này không phải nhận xét của mình mà đó là nhận xét chung của khoảng 2000 bạn đọc ghé thăm blog mỗi ngày.

Ngành lập trình rất rộng lớn, không thể đề cập hết trong một cuốn sách. Do vậy, mình tập trung nhiều vào việc rèn luyện khả năng tự học và định hướng cho bạn đọc. Có kĩ năng tự học, có định hướng tốt, bạn sẽ dễ dàng sống sót và thăng tiến trong ngành này.

MỤC LỤC

TÓM TẮT NỘI DUNG

Nội dung cuốn sách gồm 3 phần chính:

- Phần 1 tập trung vào những kĩ năng mềm và thái độ mỗi lập trình viên cần có trên ba quãng đường: Thuở còn ngồi ghế nhà trường, khi sắp ra trường và khi bắt đầu đi làm. Tuy vậy, các bạn sinh viên có thể đọc phần "làm việc" để hiểu thêm về công việc tương lai, cũng như các bạn đã đi làm có thể đọc phần "học hành" để biết và bổ sung những kĩ năng mình còn thiếu.
- Phần 2 đi sâu hơn về những kĩ thuật lập trình từ cơ bản đến nâng cao. Đa phần những kĩ thuật này không được dạy hoặc chỉ được dạy khá sơ sài ở nhiều trường đại học, dẫn đến việc sinh viên phải tự học, tự mò mẫm khi đi làm.
- Phần 3 là những mẩu chuyện và trải nghiệm nho nhỏ của chính tác giả trong quãng thời gian làm lập trình viên ở trong và ngoài nước.

Đối tượng chính của sách là các em lớp 12 sắp chọn ngành IT, các bạn sinh viên IT, những bạn lập trình viên vừa ra trường mới đi làm, và những bạn trẻ muốn tìm hiểu về ngành IT. Do vậy, sách không tập trung quá nhiều vào kĩ thuật (ngoại trừ phần 2 nặng về kĩ năng lập trình).

Bài viết trong sách sử dụng nhiều ví dụ sinh động, ngôn từ dễ hiểu, không hàn lâm nhưng nên bạn đọc không có chuyên môn về IT cũng có thể thoải mái đọc và thưởng thức. Những từ ngữ thông dụng trong ngành IT sẽ được liệt kê phía cuối sách, giúp bạn đọc dễ tìm hiểu hơn.

PHẦN 1 – KĨ NĂNG MỀM MỀM

Đa phần các bạn sinh viên thường nghĩ rằng chỉ cần học giỏi, vững kĩ năng cứng (kĩ năng lập trình) thì sẽ dễ dàng kiếm được việc làm, thăng tiến. Đây là một suy nghĩ khá sai lầm, vì đôi khi kĩ năng mềm nhiều khi còn quan trọng hơn kĩ năng cứng rất nhiều lần.

Nếu bạn code giỏi nhưng không biết giao tiếp với trưởng nhóm và các thành viên khác, bạn sẽ không thể truyền đạt ý kiến của mình hay lãnh đạo. Nếu bạn lập trình tốt nhưng không rành tiếng Anh, không biết tự học thì kiến thức của bạn sẽ rất nhanh hết thời, làm bạn bị tụt hậu. Hoặc giả bạn có giỏi đến mấy nhưng nếu cứ mang thái độ "mình là sinh trường A, B đanh giá, giỏi hơn hẳn bọn kia!" đi xin việc, bạn sẽ rớt ngay từ vòng gửi xe, à không, gửi nón.

Do vậy, mình dành ra phần đầu cuốn sách để tập trung vào những kĩ năng mềm mà lập trình viên cần có, nên có và phải có để trở thành một lập trình viên (developer) chuyên nghiệp.

Giai đoạn 1 – Học hành

Đây là giai đoạn bạn cần tập trung học các kiến thức nền tảng trong trường, rèn luyện khả năng tự học, tiếng Anh v…v. Các bài viết trong phần này sẽ mang tính định hướng, đồng thời đề cập tới những kĩ năng nói trên.

VÀI LỜI KHUYÊN VÀ ĐỊNH HƯỚNG CHỌN TRƯỜNG CHO CÁC BẠN TRẺ

Chọn trường đại học và chọn ngành học là một ngưỡng cửa khá quan trọng của cuộc đời. Có lẽ đa phần bạn đọc của sách là sinh viên đại học, hoặc đã đi làm nên chắc sẽ không cần đọc bài này. Tuy nhiên, hãy đưa nó cho em/cháu bạn hoặc phụ huynh của các em. Bài viết sẽ giúp họ có cái nhìn đúng hơn trong việc chọn trường, giúp các em hiểu hơn về công việc mình sẽ làm trong tương lai.

Lời khuyên đầu: Học trường vừa sức

Lời khuyên đầu tiên mình muốn gửi tới các bạn và các em là: Chọn trường vừa sức mình. Vừa sức ở đây không chỉ có nghĩa là vừa sức đậu, mà còn có nghĩa là vừa sức học và cạnh tranh.

Tại sao lại chọn trường vừa sức mà không phải là trường nổi tiếng? Theo lẽ thường, chất lượng dạy và học ở của các trường nổi tiếng này khá cao, tấm bằng đại học danh tiếng cũng rất có ích khi bạn vừa ra trường xin việc hoặc muốn tiếp tục học lên cao. Tuy nhiên, ở các trường này, do chất lượng đầu vào cao nên bạn sẽ phải học hành và cạnh tranh với những bạn bè giỏi hơn (còn được gọi dưới cái tên thiên tài hay quái vật). Nếu không đủ giỏi, việc cạnh tranh với những thành phần này dễ làm bạn nản lòng thoái chí. Chưa kể, do các giáo viên đã quen với việc dạy dỗ học sinh thông minh, có thể họ sẽ giảng giải với tốc độ nhanh hơn, khó hiểu hơn, làm bạn khó theo kịp.

Ngoài ra, vào những trường giỏi, bạn rất khó để vào top đầu lớp hoặc gây ấn tượng với giáo viên (vì học sinh giỏi nhiều quá rồi). Ngày xưa, mình cũng đậu đại học BK HCM nhưng không học, một phần là do FPT có học bổng 70%, một phần là do mình không muốn bỏ quá nhiều công sức vào việc cạnh tranh học tập. Vào FPT, mình dễ dàng đứng đầu lớp, được nhiều giáo viên thương và để ý. Nhờ vậy, mình dễ dàng xin thư giới thiệu của họ khi làm đơn du học.

Lời khuyên thứ hai: Tự học đi, không ai dạy bạn đâu!

Lời khuyên thứ hai là: Kiến thức ở trong trường không đủ để bạn xin việc đi làm đâu[1]. Do đó hãy bắt đầu rèn luyện kĩ năng tự học và tìm hiểu từ bây giờ đi. Những

[1] Đọc kĩ hơn trong bài viết: Những điều trường đại học không dạy bạn"

kiến thức bạn có được khi học đại học chỉ là nền tảng thôi, khó mà áp dụng ngay vào công việc! Nếu chỉ biết những gì được dạy mà không biết tự mày mò học thêm, bạn sẽ gặp khá nhiều khó khăn khi ôm mớ nền tảng đó đi xin việc đấy!

Nhiều bạn sinh viên đi học được một, hai năm thì bắt đầu rơi vào tình trạng mất phương hướng vì cảm thấy những thứ mình học quá vô dụng, chẳng làm được gì. Thay vì chờ được thầy cô dạy, hãy tận dụng những kiến thức nền tảng đã có để tự học, sau đó áp dụng những thứ vừa học để tạo ra một sản phẩm gì đó, bạn sẽ thấy thích học ngay thôi. Còn nữa, nhớ phải tập trung trau dồi tiếng Anh nhé[2]. Học IT mà không giỏi tiếng Anh thì khó phát triển lắm đấy!

Học IT xong thì ra làm gì?

Ở Việt Nam, hầu như mọi người chỉ biết ngành IT (Information Technology – Công nghệ thông tin), chứ không biết tường tận ngành đó làm những gì. Do đó, mình sẽ giải thích một số chuyên ngành của ngành IT, về những thứ bạn sẽ học cũng như công việc bạn sẽ làm sau khi ra trường.

Hiện tại ngành IT có một số chuyên ngành sau:

- **Khoa học máy tính (Computer Science)**: Bạn sẽ học các thứ liên quan tới cách thức máy tính hoạt động. Theo như tên gọi "Khoa học", chuyên ngành này thường nặng về nghiên cứu.
- **Kĩ nghệ phần mềm (Software Engineering)**: Ngành này cũng học một số môn tương tự như CS. Tuy nhiên, chuyên ngành này thiên về thực tế và xây dựng phần mềm nên bạn được học thêm 1 số ngành như: Qui trình phát triển phần mềm, Kiểm thử phần mềm. Học ngành này bạn có thể viết ứng dụng, viết website hoặc xây dựng một hệ thống. Sinh viên tốt nghiệp cả hai ngành CS và SE đều có thể làm lập trình viên (developer).
- **Hệ thống thông tin (Information System)**: Ngành này thiên về phân tích thiết kế hệ thống dựa theo yêu cầu của khách hàng. Bạn sẽ phải học một số môn liên quan tới Thương mại điện tử, cách thức các doanh nghiệp hoạt động. Khi ra trường bạn có thể làm ở vị trí Business Analyst (BA).
- **Hệ thống nhúng (Embedded System)**: Ngành này tập trung vào việc xử lý tín hiệu số, thiết kế mạch điện, chip điện tử và linh kiện. Khi ra trường, bạn cũng là lập trình viên, nhưng lập trình cho các thiết bị hoặc mạch điện. Ngành này hơi khó và khô khan hơn ngành SE, nhưng lương trung bình cao hơn một chút.
- **Lập trình mạng (Network Engineering)**: Ngành này dạy về cơ sở hạ tầng mạng, cách lắp đặt hệ thống, v…v. Mấy bác tốt nghiệp ngành này là những

[2] Mình có chia sẻ kinh nghiệm học và thi trong bài "Tôi đã học tiếng Anh như thế nào"

người cài win dạo, bấm cáp dạo, sửa modem, quản lý server, thường gọi là IT Helpdesk. Họ là những người hùng thầm lặng, giúp hệ thống hoạt động trơn tru.

- **An toàn thông tin (Infomation Security)**: Ngành này tập trung về bảo mật, bạn sẽ được học về kiến trúc hệ thống, mã hóa, bảo mật, những phương thức hack và cách phòng chống. Ngành này phù hợp với những bạn hâm mộ các anh hacker. Ra trường, bạn có thể làm hacker mũ trắng hoặc chuyên viên bảo mật cho các công ty.

Có một số môn như Hệ điều hành, Mạng máy tính, Mã máy, Thuật toán, Cấu trúc dữ liệu…. mà sinh viên chuyên ngành nào cũng phải học. Giữa các trường đại học, chương trình học của các chuyên ngành này sẽ có đôi chút khác biệt.

Tóm tắt:

- Nên chọn trường vừa sức để bạn có thể nằm trong tốp đầu lớp
- Cần rèn luyện khả năng tự học. Trong ngành này, tự học là chính, kiến thức trong nhà trường là không đủ
- Tuỳ vào ngành học mà sinh viên IT ra trường có thể làm rất nhiều nghề: lập trình viên, quản trị mạng, an toàn thông tin,…

LẬP TRÌNH VIÊN CÓ CẦN HỌC ĐẠI HỌC HAY KHÔNG?

Trên các trang web, diễn đàn về lập trình, mình thường gặp các topic kiểu: "Cảm thấy mất định hướng khi học đại học", "Liệu em có nên bỏ đại học hay không?", "Kiến thức đại học toàn thứ vô bổ!". Các topic này cho thấy nhiều bạn sinh viên đang cảm thấy hoang mang về giá trị của việc học đại học.

Mình cũng từng là sinh viên, từng cảm thấy hoang mang như vậy.. Bài viết này sẽ phân tích những cái lợi của việc học đại học và giá trị tấm bằng đại học, giúp các bạn vững tin hơn trên con đường đã chọn.

Nguyên nhân gây hoang mang

Đa phần nguyên nhân làm các bạn sinh viên cảm thấy mất phương hướng khi học đại học là: Phải học nhiều môn học nặng và nhàm chán như Toán Lý Hóa Đại Cương, Tư tưởng Mác-Lê-Minh, … Chương trình học thì khá cũ và thiên về lý thuyết, các môn lập trình thì quanh đi quẩn lại chỉ có C, C++,Java,… làm việc với console. Trong khi đó, những kiến thức thực tế, cần thiết để đi làm thì lại ít được dạy [3]. Nhiều bạn cảm thấy mình tốn phí khá nhiều thời gian học mà chỉ được học được những điều vô bổ!!

Mặt khác, báo chí rất thích lăng xê các tấm gương "Bỏ đại học mở công ty thu nhập hàng trăm triệu" để làm lung lay tinh thần các em sinh viên. Lẽ ra báo chí nên đưa một số vụ việc như "Nghỉ đại học, lái xe ôm, làm thợ hồ. Tốt nghiệp đại học, công việc ổn định lương vài chục triệu" thì mọi người sẽ có cái nhìn khách quan hơn.

Ngoài ra, các bạn cũng đừng nghe các thánh phán: "Bill Gates và Mark Zuckerberg bỏ ĐH vẫn thành công đấy thôi!". Hai người này bỏ ĐH Harvard nhé, nếu có khả năng các bạn có thể cố gắng vào Harvard rồi bỏ để thử vận may xem sao.

Giá trị của bốn năm đại học

Thật ra, bốn năm đại học cũng không đến nỗi vô bổ như bạn tưởng tượng đâu, chúng sẽ cho bạn những điều sau:

- **Khả năng tư duy**: Các môn Toán, Lý, Hóa đại cương thoạt nhìn có vẻ vô bổ. Tính ra là chúng … vô bổ thật, vì ta cũng ít áp dụng chúng trực tiếp vào công việc. Tuy vậy, việc tập trung giải quyết những bài toán này sẽ rèn cho chúng ta khả năng tu duy và khả năng tiếp cận vấn đề. Chúng rất hữu ích trên con đường nghề nghiệp sau này.

[3] Tìm hiểu trong bài "Những điều trường đại học không dạy bạn"

- **Khả năng đọc-viết**: Học đại học, bạn sẽ phải đọc vô số tài liệu phức tạp và viết luận. Những kĩ năng đọc viết này sẽ đi theo bạn tới tận sau này khi phải viết đơn xin việc, viết mail, viết báo cáo.
- **Khả năng tự học**: Đây là một trong các kĩ năng quan trọng nhất trong ngành phần mềm. Ở đại học, các thầy cô sẽ không chỉ dẫn tận tay như cấp 3, đòi hỏi bạn phải cố gắng nhiều hơn.
- **Kiến thức nền tảng**: Chương trình học ở Đai Học khá bài bản và có cấu trúc. Hầu như trường nào cũng dạy các kiến thức về: Hệ điều hành, Quy trình phát triển phần mềm, Cấu trúc dữ liệu và Giải thuật, … Những kiến thức nền tảng này khá quan trọng. Đa phần các bạn tự học lập trình thường hay thiếu những kiến thức nền như thế này.
- **Bạn bè và quan hệ**: Bạn sẽ được làm quen và làm việc chung với rất nhiều bạn bè. Điều này rèn luyện khả năng làm việc nhóm (cũng như khả năng chịu đựng) của bạn. Sau khi ra trường, các mối quan hệ với giáo viên, bạn bè rất quan trọng. Tranh thủ kiếm gấu ở giai đoạn này luôn nhé, ra trường khó tìm lắm.
- **Cơ hội việc làm**: Bạn có thể tìm được một công việc với mức lương cao, môi trường tốt nhờ sự giới thiệu của thầy giáo hay bạn bè. Ngoài ra, các trường đại học thường có liên kết với một số công ty, giúp bạn dễ dàng tìm cơ hội thực tập và xin việc.

Ngoài ra, hoàn thành 4 năm học sẽ giúp bạn có một tấm bằng đại học trong tay. Mảnh bằng này có giá trị gì không? Hãy đọc phần dưới nhé.

Giá trị của tấm bằng ĐH

Hiện tại, đa phần các công ty phần mềm đều đòi hỏi ứng viên phải có một tấm bằng đại học hoặc cao đẳng ngành CNTT. Không cần biết khả năng của bạn thế nào, tấm bằng đại học là tấm giấy thông hành để được bạn vượt qua được vòng loại của bên nhân sự, vào vòng phỏng vấn để chứng tỏ khả năng của mình.

Nhiều bạn cũng hỏi mình: "Sao em thấy báo đăng nhiều đứa tốt nghiệp đại học vẫn thất nghiệp đấy thôi, vậy học để làm gì?". Tất nhiên, nếu bạn không có khả năng thì mảnh bằng đại học cũng chỉ là mẩu giấy lộn mà thôi. Nó chỉ giúp bạn vào được vòng phỏng vấn, còn có tìm được việc làm hay không là tùy kinh nghiệm và khả năng của bạn.

Một lời khuyên khác cho các bạn: Ngoài đại học ra, các bạn có thể lựa chọn học cao đẳng như VTC Academy, FPT Polytechnic, Aptech. Chương trình học của các cơ sở này sát với thực tế hơn, có thể đi làm được ngay. Họ cũng hỗ trợ giới thiệu cơ hội thực tập và việc làm cho sinh viên. Bạn bè, đồng nghiệp mình cũng có nhiều người tốt nghiệp cao đẳng và vẫn đi làm thoải mái.

Tóm tắt:

- Đừng vội nghe theo các tấm gương nghỉ học đại học rồi làm giàu, đó chỉ là thiểu số
- Bốn năm học đai học rèn luyện cho bạn khả năng tư duy, khả năng tự học, kiến thức nền tảng và cung cấp cơ hội việc làm
- Bằng đại học chỉ là tấm giấy thông hành khi xin việc, nếu bạn không có khả năng thực sự thì nó chỉ là tờ giấy lộn mà thôi

HAI SAI LẦM LỚN NHẤT TRONG QUÁ TRÌNH HỌC LẬP TRÌNH

Bài viết này nói về hai sai lầm lớn nhất trong quá trình học lập trình. Qua trao đổi với các bạn sinh viên, mình nhận thấy có khá nhiều bạn sinh viên mắc phải những sai lầm này (cá nhân mình hồi năm nhất năm hai cũng thế). Do đó, bài viết này sẽ cảnh tỉnh một số bạn, đồng thời chia sẻ chút kinh nghiệm để tránh các bạn đi theo vết xe đổ của mình ngày trước.

Câu nói hay gặp – trường em không dạy A, B, C ...

Dưới đây là một mẫu đối thoại giữa mình và một bạn sinh viên (giấu tên)

- Bạn: Anh ơi, học C với C++ ra trường thì làm được gì anh?
- Mình: Làm hệ thống nhúng hoặc game em nhé, lương khủng lắm đấy.
- Bạn: Em thích làm Web hoặc làm app di động cơ, C++ làm được không anh?
- Mình: Không em nhé, muốn làm web thì học HTML/CSS/JS. Sau đó có thể chuyển qua làm hybrid app mobile [4], hoặc học Android để viết app.
- Bạn: Mấy cái đó trường em không dạy anh ơi!!!
- Mình: ...

Một câu mình nói mình hay được nghe các bạn nói là: trường em chỉ dạy C, trường em chỉ dạy Java, mấy thầy cô không dạy HTML hay làm Web… Mình đã từng nói ở đầu sách, đại học chỉ cho bạn các kiến thức nền tảng về lập trình. Họ sẽ không dạy bạn cách code như thế nào, cách làm việc, cách sử dụng một ngôn ngữ hoặc framework ra sao, mà bạn sẽ phải tự dạy mình!!.

Thái độ trường không dạy nên không biết là một thái độ học tập cực kỳ sai lầm. Vấn đề không phải người ta dạy cho bạn cái gì, mà là bạn có thể học được cái gì! Thái độ ngồi chờ sung rụng, có người dạy mới học này sẽ cản trở bạn trên con đường tìm hiểu cái mới, tự cập nhật kiến thức cho bản thân. Nếu giữ thái độ này, kiến thức của bạn sẽ nhanh chóng lạc hậu, lỗi thời, gây ra nhiều khó khăn trên con đường thăng tiến của bạn.

Học tập thế nào cho đúng??

Việc học phải mang tính chất chủ động chứ không phải là bị động. Bạn phải tự tìm cái cần học và tự sắp xếp thời gian để học. Bạn nào kiến thức vững, đủ kiên nhẫn

[4] Xem thêm trong bài "Tổng quan về lập trình ứng dụng di động"

để tự học thì có thể tải ebook tiếng Anh hoặc tìm nguồn tự học trên mạng. Bạn nào kiến thức còn yếu thì có thể ra trung tâm để có người kèm cặp.

Nhà tuyển dụng chỉ cần biết bạn có kiến thức hay không và có làm được việc hay không. Họ không quan tâm là kiến thức đó bạn có được từ nhà trường, từ trung tâm hay do tự học. Thứ duy nhất bạn phải nhớ là: thầy cô hay trung tâm cũng không thể vá lỗ hổng kiến thức hay dạy cho bạn tường tận được, mà chính bạn mới là người nỗ lực hấp thu, biến kiến thức của họ thành kiến thức của mình.

Chưa kể, chương trình học ở các trường bây giờ... cũ xì, quanh đi quẩn lại chỉ có WinForm, WebForm, Java Servlet... . Nếu cứ "há miệng chờ sung, dạy gì học nấy", bạn sẽ không có đủ kĩ năng cần thiết để xin việc khi ra trường đâu nhé!

Ngoài ra, đừng nghĩ rằng chỉ học một lần cho biết là xong, nguy hiểm lắm! Công nghệ liên tục thay đổi, bạn cũng phải thường xuyên cập nhật kiến thức bản thân. Trước đây mình từng có khoảng 2 năm kinh nghiệm lập trình C#. Đầu năm nay, lúc mình xem lại thì công nghệ đã được cập nhật, làm cho kiến thức cũ của mình lỗi thời hết cả! Thay vì than trời trách đất, mình đành phải tự học để cập nhật kiến thức mới thôi.

Sai lầm thứ hai: Cẩn thận, chưa chắc học nhiều/xem nhiều là sẽ giỏi!!

Người Việt chúng ta có thói quen ghét ai ghét cả đường đi lối về. Khi đã tin tưởng hay thần tượng ai đó thì nó nói gì cũng tin; khi đã ghét thằng nào thì nó nói gì cũng sai, cũng nhảm nhí. Thái độ này dễ làm bạn tiếp nhận sai tiếp nhận thông tin sai cách!

Mình hay đọc sách Tony Buổi Sáng, có những bài viết về cách nhìn cuộc sống khá hay. Thay vì hâm mộ, nuốt từng câu từng chữ của "dượng", vẫn có những đoạn văn chém gió, những cách nhìn mà mình không đồng tình. Tuy vậy, mình vẫn chắt lọc những điều hay, những điều tác giả muốn gửi gắm, còn mấy vấn đề mình không đồng tình thì bỏ qua.

Tương tự, các bạn nên đọc sách, nghe thầy cô giảng nhưng đừng nên tin tưởng hoàn toàn những gì được nghe. Hãy tự hỏi xem: Thầy cô hay sách nói như vậy đúng hay sai, có chứng cứ gì không? Cuộc sống có câu "Không có gì là vĩnh cửu", có thể bây giờ mình cho điều đó là đúng, nhưng trong tương lại điều đó lại sai thì sao?

Đừng tin toàn bộ những gì sách nói, cũng đừng nuốt từng câu từng lời của thầy cô hay tác giả. Nghe người khác nói cái gì cũng phải nghi ngờ vả kiểm chứng. Hãy xem tác giả là một thanh niên đang chém gió với mình thông qua sách, cái gì đúng thì gật gù đồng ý, cái gì sai thì phản bác lại ngay.

Ngoài việc học nhiều, bạn còn phải biết cách lọc bỏ, chọn lựa những thông tin có ích cho bản. Những gì hay thì hãy ghi nhớ và học theo; những gì nhảm nhí thì cứ bỏ qua, coi như nó không tồn tại. Nói một cách dân dã là phải biết cách "gạn đục khơi trong" từ vô số nguồn kiến thức.

Kết luận

Sửa được hai sai lầm về thái độ học tập bị động và chọn lọc kiến thức, bạn sẽ thấy mình trở nên vô cùng tự tin. Công nghệ A/B không có trong chương trình học? Chả sao, chỉ cần tự học vài buổi là xong! Càng học nhiều, bạn sẽ càng thấy việc học cái mới trở nên rất dễ dàng và nhanh chóng.

Hi vọng, sau bài viết này, mình sẽ không còn phải nghe câu "em không biết cái ABC này, trường với thầy cô không dạy" nữa. Thay vào đó, mình hi vọng sẽ được nghe câu: "Em đang tự tìm hiểu cái ABC, anh chỉ em một số nguồn học và những điều cần lưu ý nha".

Tóm tắt:

- Đừng trong chờ vào việc nhà trường sẽ dạy cho bạn kiến thức để làm việc. Chịu khó tự học càng sớm càng tốt.
- Đọc nhiều, học nhiều là tốt, nhưng phải biết cách loại bỏ những thứ vô bổ, giữ lại những điều có ích.

ĐƯỢC GÌ MẤT GÌ KHI HỌC LẬP TRÌNH BẰNG TIẾNG VIỆT

Hiện tại, nhiều trường đại học vẫn dạy các môn lập trình bằng tiếng Việt. Hãy cùng tìm hiểu xem bạn sẽ phải chịu những thiệt thòi gì khi phải học lập trình bằng tiếng Việt nhé.

Học bằng tiếng Việt thì được gì?

Có thể nói, lập trình là một ngành khó. Không chỉ đòi hỏi suy nghĩ logic, bạn còn phải làm quen với rất nhiều khái niệm mới lạ như function, object, pointer,…. Ở những giai đoạn đầu của việc học lập trình, sử dụng tiếng Việt sẽ giúp bạn thấy dễ hiểu và dễ tiếp thu hơn. Các khái niệm như biến, mảng, con trỏ, vòng lặp,… được dịch ra tiếng Việt sẽ dễ hiểu hơn.

Với những môn phức tạp khác như Cấu trúc dữ liệu giải thuật, hướng đối tượng, … ta phải tiếp xúc với nhiều khái niệm rắc rối, các thuật toán dài dòng. Lúc này, học bằng tiếng Việt sẽ giúp tiết kiệm được thời gian, giúp ta dễ nhớ, dễ thấm hơn. Song, học lập trình tiếng Việt cũng làm bạn thiệt thòi rất nhiều.

Mất gì à? Cũng kha khá đấy!

Học lập trình tiếng Việt sẽ gây khá nhiều khó khăn khi code, đồng thời cản trở quá trình tự học và phát triển của bạn. Mình nói thật đấy, không đùa đâu! Nguồn tài liệu lập trình tiếng Việt khá ít ỏi. Hầu hết những sách tiếng Việt là sách về C, C++, và… thuật toán. Những sách về công nghệ mới không nhiều. Những cuốn sách thuộc hàng kinh điển trong giới developer thế giới như: Clean Code, Refactoring, Code Complete… đều không có tiếng Việt.

Do đã quen với việc dùng tiếng Việt nên mỗi khi gặp khó khăn các bạn thường hay Google bằng tiếng Việt để tìm câu trả lời. Tiếc thay, ở Việt Nam không có Stackoverflow mà chỉ có một vài diễn đàn lập trình và các group Facebook nên nhiều khả năng bạn sẽ không tìm được câu trả lời mình cần. Không tìm được cách giải quyết vấn đề, nhiều bạn phải mang đi hỏi lung tung. Điều này sẽ làm bạn phải dựa dẫm vào người khác và mất đi tính tự chủ.

Học bằng tiếng Việt sẽ có lợi khi bạn trả lời phỏng vấn bằng tiếng Việt. Tuy vậy, bạn cũng sẽ gặp khó khăn khi muốn vươn tầm ra thế giới như lên stackoverflow thể hiện bản thân hay đi phỏng vấn ở công ty nước ngoài (Ngày xưa vào ASWIG Solution mình cũng "bị" phỏng vấn bằng tiếng Anh đấy).

Học lập trình bằng tiếng Anh thì được chi?

Nếu đã quen việc học lập trình bằng tiếng Anh, bạn có thể tha hồ đọc sách kĩ thuật và xem video hướng dẫn trên mạng. Một chân trời mới bao la sẽ mở ra trước mắt

bạn. Mỗi khi gặp lỗi, chỉ cần đọc hiểu Exception, sau đó tra cứu Google bằng tiếng Anh chỉ mất 5 phút là ra kết quả. Không còn phải vác vấn đề đi hỏi, chờ được trả lời nữa.

Không còn phải nhờ vả người khác, bạn có thể chủ động hơn trong việc học. Do tài liệu tiếng Anh rất đầy đủ và miễn phí nên bạn có thể học bất kì thứ gì mình muốn. Có vô số trang web dạy code miễn phí bằng tiếng Anh. Bạn tha hồ tìm hiểu và vọc công nghệ mới mà không cần phải đợi tài liệu tiếng Việt. Mà cũng không cần nói đâu xa, các mẫu tuyển dụng developer có khả năng tiếng Anh thì lương lậu đều nhỉnh hơn so với developer không có tiếng Anh nhé.

Nghe có vẻ hay, nhưng mà có vẻ khó….

Phần đông sinh viên Việt Nam không giỏi tiếng Anh nên họ rất ngại. Dĩ nhiên mình cũng biết là nếu đã quen với tiếng Việt thì cũng khó có thể chuyển qua tiếng Anh ngay lập tức. Để bắt đầu, các bạn có thể tìm đọc một số series đơn giản: Head First, For Dummies. Các series này dùng ngôn ngữ dễ hiểu, nhiều hình ảnh minh họa, code đầy đủ.

Sau khi đã đọc sách quen thì các bạn có thể lên Cousera, Udemy, Pluralsight,… tìm xem các khóa học có phụ đề[5]. Vừa luyện đọc nghe tiếng Anh, vừa nâng cao kĩ năng lập trình, một công đôi ba việc luôn còn gì! Bắt đầu ngay đi nhé!

Tóm tắt:

- Học lập trình bằng tiếng Việt sẽ dễ tiếp thu hơn, nhưng sẽ gây khó khăn khi code, tìm lỗi hay tự học.
- Có thể dần làm quen với tiếng Anh bằng cách đọc các đầu sách lập trình đơn giản, sau đó tìm xem các khóa học và video.

TÔI ĐÃ HỌC TIẾNG ANH NHƯ THẾ NÀO

Có 2 điều mình muốn nói rõ trước khi bắt đầu:

- Đầu tiên, học là một quá trình lâu dài. Trừ khi bạn là siêu nhân hay thiên tài nhìn chữ là nhớ, còn lại thì học gì cũng cần thời gian để xây dựng nền móng mới giỏi dần được. Mình không phải thiên tài cũng không phải siêu nhân, do đó mình cũng phải học nhiều và học dần dần thì tiếng Anh mới khá được.
- Thứ hai, mỗi người có một kinh nghiệm/cách học khác nhau. Có người thích học tà tà mỗi ngày nửa tiếng, có người thích cày như trâu mỗi ngày 8 tiếng. Cách mình chia sẻ là cách mình thấy phù hợp với bản thân mình, các

[5] Xem bài viết "Top các trường dạy code online cho developer"

bạn thấy phần nào hợp với chính mình thì làm theo, đừng nghe và làm theo 100%. Nếu không thấy hiệu quả thì mình cũng không chịu trách nhiệm đâu.

Trong 3 phần của bài viết, mình sẽ chia sẻ phương học tập tiếng Anh và ôn luyện cho các kì thi TOEIC/IELTS của bản thân. Đây là những kinh nghiệm đã giúp mình đạt 940/990 TOEIC và 7.5/9 IELTS.

Phần 1 – Luyện thói quen học tiếng Anh

Ngày xửa ngày xưa

Hồi lớp 10-11, tiếng Anh mình cũng chỉ thuộc hàng kha khá trong lớp chứ cũng không phải là nổi trội hay xuất sắc. Đối với mình thời đó, tiếng Anh chỉ là môn học và là công cụ hỗ trợ… chơi game. Công bằng mà nói, games là công cụ dạy tiếng Anh rất tốt. Nếu muốn, bạn có thể bỏ ra 2-3 tiếng mỗi ngày để chơi games, vừa giải trí vừa học tiếng Anh luôn thể.

Thể loại game tốt nhất giúp tăng khả năng tiếng Anh là game nhập vai (JRPG của Nhật, RPG của châu Âu), bạn sẽ quen với việc đọc chữ, đọc lời thoại, xem hướng dẫn để tìm cách làm. Một số game nhập vai khá hay bạn nên thử là: Series Final Fantasy 6-10, Series Tales… Nhớ chọn những game nhiều chữ nhiều lời thoại chứ đừng chơi web game, Dota2 hay Lol Việt Hóa nhé.

Cuối năm cấp 3

Đến cuối năm lớp 12, do tìm một số bộ anime lên mình bị sa chân vào ma đạo, bắt đầu con đường cày cuốc anime và manga. Mình nhớ hồi đó là khoảng năm 2008-2009, trang vnsharing.net mới thành lập không lâu. Năm ngoái nó vừa sập vì nhiều lý do, tính ra mình cũng có nhiều bạn bè, kỉ niệm với nó, kể cũng buồn.

Thời đó, số lượng các nhóm dịch còn ít như lá mùa thu, không nhan nhản như bây giờ. Do đó, mình phải cắn răng mà mò mẫm đọc truyện/xem anime bằng tiếng Anh. Nghĩ lại cũng may chứ nếu hồi xưa có nhiều sub Việt như bây giờ chắc mình chả đụng tới tiếng Anh nữa. Theo kinh nghiệm của mình thì anime/manga cũng là một cách để học tiếng Anh khá tốt, vừa học vừa giải trí. Anime dễ xem hơn vì nhân vật nói từng câu, không phải nguyên 1 đống chữ như trong manga. Ngôn ngữ trong anime/manga thì đủ thể loại, ma pháp học đường chính trị đủ cả. Các bạn nhập môn có thể tìm xem một số bộ đơn giản như: Naruto, One Piece, Yotsuba, School Rumble, Aria, …

Nếu thích đọc manga bạn cũng có thể thử đọc tiếng Anh thay vì tiếng Việt. Thay vì mỗi ngày vào blogtruyen, manga24h, bạn hãy vào mangapark, mangafox… Vốn từ tiếng Anh của bạn sẽ lên dần dần mà không cần cố công rèn luyện học hành gì đâu.

Nhiều khi đọc quen thì bạn đọc manga cả tiếng đồng hồ mà không biết luôn ấy chứ. Bạn cũng có thể bỏ manga vào tablet và mang theo đọc. iOS có *iComic* là phần mềm đọc manga khá tốt, nếu bạn dùng Android thì có thể thử *Perfect Viewer*, ngày xưa mình dùng cả 2.

Lên đại học

Lên đại học, mình vẫn chơi game/xem anime/đọc manga như thường lệ. Do học FPT, giáo trình toàn bộ bằng tiếng Anh, sách cuốn nào cuốn nấy dày cui như từ điển nên mình cũng phải quen dần với việc đọc. Ở giai đoạn này, mình được giới thiệu một số sách khá hay về lập trình và cuộc sống nên mình bắt đầu có thói quen đọc sách. Tuy nhiên hồi đó có khi cả tháng trời mình mới đọc xong một cuốn.

Một điều khá may mắn là sách kĩ thuật cho dân lập trình chúng mình được viết bằng văn phong đơn giản, súc tích dễ hiểu, lại còn có nhiều code. Nếu đọc sách lập trình, bạn không cần đọc hết từng câu từng chữ mà chỉ cần nắm ý chính và hiểu code là được.

Tuy nhiên, đọc ebook trên vi tính rất mỏi mắt và dễ mất tập trung. Vì vậy các bạn nên tải sách vào di động hoặc tablet để mang đi mà đọc. Đây là một số phần mềm nên dùng cho cách định dạng sách:

- Định dạng PRC và MOBI: Kindle. Kindle có thể cài kèm từ điển, gặp từ khó bạn chỉ cần giữ tay vào chữ sẽ thấy giải nghĩa ngay, rất tiện.
- Định dạng EPUB: Marvin (Cả iOS và Android)
- Định dạng PDF: GoodReader

Mình đã dùng thử các phần mềm tương tự và giới thiệu cái tốt nhất cho các bạn đấy. Phần một tập trung nhiều vào Reading, vì nó là kĩ năng tiếng Anh quan trọng nhất của lập trình viên (Phải biết đọc giỏi thì bạn mới có thể đọc tài liệu dự án, tự học thông qua video và ebook[6]). Các kĩ năng còn lại sẽ được giỏi thiệu trong hai phần tiếp theo.

Phần 2 – Tôi đã tự học và thi TOEIC như thế nào

Với nhiều bạn sinh viên, TOEIC là một kì thi khá quan trọng, vì nhiều trường đại học đòi hỏi kết quả thi TOEIC vào khoảng 400-600 điểm mới cấp bằng. Nối tiếp phần trước, trong bài viết này, mình sẽ chia sẻ lại một số kinh nghiệm quá trình ôn tập, học và thi TOEIC.

[6] Không biết đọc sách gì? Hãy xem bài "Lập trình viên nên đọc sách gì?"

Chuẩn bị tinh thần và tài liệu

Một điều cần cảnh báo trước cho các bạn là: TOEIC rất dễ. Nhiều bạn cứ tưởng nó ghê gớm lắm chứ tới lúc bắt đầu ôn rồi mới thấy nó … vô cùng dễ, chắc chỉ khó hơn đề thi Anh Văn tốt nghiệp một chút thôi. Bạn bè mình nhiều người tiếng Anh không giỏi nhưng tự ôn 2-3 tháng thi toàn được 7-800 điểm trở lên cả. Vì vậy nên các bạn nên bỏ tư tưởng sợ hãi đi nhé.

Ngoài ra, nội dung TOEIC là tiếng Anh thường nhật, công sở, chỉ có thể dùng khi đi xin việc. Nếu bạn có ý định du học thì tập trung học và thi IELTS chứ đừng lấy bằng TOEIC làm gì nhé. Đa phần các bạn tự học, tự ôn thi hay gặp một trong 2 vấn đề sau:

- Không biết Google nên không biết nên học ôn thế nào hay chuẩn bị tài liệu gì.
- Biết Google nên tìm được một đống lời khuyên về học và ôn thi TOEIC, cuối cùng chết ngộp trong đống tài liệu vì… nhiều quá, không biết học cái nào.

Với trường hợp 1, các bạn có thể hỏi ý kiến hoặc kinh nghiệm từ người thân, bạn bè hoặc… ra trung tâm học. Nói gì thì nói, tự học đòi hỏi bạn phải kiên trì và chịu khó xây dựng thói quen, khá là vất vả. Ngoài trung tâm họ sẽ có một kì thi nhỏ nhằm đánh giá trình độ của bạn, từ đó tư vấn lộ trình và lớp học phù hợp. Mình không học ngoài trung tâm nên không giới thiệu được trung tâm nào tốt đâu.

Đa phần bạn bè của mình sa vào trường hợp 2, bỏ cả mấy ngày trời tải về đủ thứ sách vở, tư liệu lên đến vài GB, sau đó tẩu hỏa nhập ma vì … nhiều quá, chả biết học thế nào hay bắt đầu từ đâu. Thói quen của mình là chuẩn bị đầy đủ các tài liệu và lên lịch rồi mới bắt đầu học và ôn tập. Theo mình, bạn không cần quá nhiều sách ôn tập, chỉ cần đủ và chất lượng thôi.

Để chuẩn bị cho kì thi TOEIC mình chỉ đọc đúng 4 cuốn sách dưới đây. Bạn có thể dễ dàng tìm ebook trên mạng hoặc bản cứng trong nhà sách.

- Starter TOEIC
- Developing Skills for TOEIC Test
- Toeic Analyst
- Target TOEIC

Ôn thi TOEIC

Bạn chỉ bỏ thời ra khoảng 1 tiếng rưỡi đến 2 tiếng mỗi ngày trong vòng 2 tháng là xong ngay. Luyện thi TOEIC hay IELTS thì đều phải trải qua 3 giai đoạn:

- Giai đoạn 1: Ôn luyện lại kĩ năng tiếng Anh cơ bản nói chung như: Văn phạm, từ vựng, ngữ pháp, kĩ năng nghe đọc.

- Giai đoạn 2: Làm quen với cấu trúc đề TOEIC, các dạng câu hỏi, những điều cần lưu ý. Trong tài liệu mình chia sẻ là 3 cuốn: Starter TOEIC, Developing Skills for TOEIC Test, Toeic Analyst.
- Giai đoạn 3: Thi thử, tính thời gian và làm quen với đề. Trong cuốn Target TOEIC có 6 đề và lời giải.

Sau 3 giai đoạn ôn luyện này, bạn đã có đủ khả năng để bước vào phòng thi và làm bài rồi. Nếu còn thời gian rảnh, bạn có thể xem thêm một số mánh khoé để tăng điểm số. Tuy nhiên các bạn nên nhớ một điều là: Khả năng tiếng Anh mới là thứ quan trọng nhất giúp bạn đạt điểm cao, những mánh khoé này chỉ giúp bạn không mất điểm một cách vô duyên thôi.

Lời kết

Quá trình thi cũng không có điều gì đáng nói. Bạn tới trung tâm đăng kí, chọn ngày thi và đóng tiền. Tới hôm thi nhớ cầm theo CMND hoặc passport rồi vào phòng thi là được. Chất lượng âm thanh khá ổn, mỗi người có một tai nghe riêng nên đừng lo về phần Listening.

Phần 3 – Tôi đã tự học và thi IELTS như thế nào

Như đã nói ở phần trước, TOEIC rất dễ, còn IELTS khó hơn TOEIC nhiều lắm. Do đó quá trình ôn thi IELTS cũng lâu và lắm gian truân hơn nhiều. Ở phần 3, mình sẽ chia sẻ về quá trình ôn và thi IELTS "đầy mồ hôi xương máu" này.

Sơ lược về IELTS

IELTS là một chứng chỉ tiếng Anh thường được các trường đòi hỏi khi bạn đăng kí học đại học hoặc Cao Học. Nếu muốn du học, bạn cần chứng chỉ IELTS trên 6.5 hoặc 7 tùy theo ngành (Một số trường khác đòi hỏi TOEFL). Vì IELTS có 2 dạng là General và Academic, dạng Academic tập trung nhiều hơn vào từ vựng chuyên ngành và học tập nên khó hơn.

Cấu trúc đề IELTS gồm 4 phần: Listening, Reading, Writing, Speaking. Tổng thời gian thi 3 phần đầu khoảng 3 tiếng. Phần thi Speaking sẽ được tách riêng, mất khoảng 10-20 phút.

Chuẩn bị tinh thần và tài liệu

Ở bài trước, mình đã chia sẻ với các bạn rằng TOEIC rất dễ, đừng lo, cứ thả lỏng tinh thần, yên tâm mà học. Rất tiếc rằng mình phải nói điều ngược lại so với IELTS. IELTS khá khó, nằm ở một đẳng cấp cao hơn nhiều so với TOEIC vì các lý do sau:

- IELTS có cả *nghe, nói, đọc, viết* trong khi TOEIC chỉ có *nghe, đọc*. Ôn thi IELTS phải ôn cả 4 kĩ năng.

- Vốn từ của TOEIC hầu hết là tiếng Anh công sở, trong khi IELTS lại tập trung vào tiếng Anh học thuật (Academic English). Những từ dùng trong academic khá hiếm gặp và khó nhớ.
- TOEIC chỉ có trắc nghiệm, còn IELTS bạn phải trả lời kiểu tự luận, điền từ, sắp xếp. Điền từ mà chia sai thì hoặc số ít/số nhiều sẽ không có điểm.

Vì độ khó của IELTS cao hơn TOEIC nên thời gian ôn luyện và học phí của các trung tâm cũng cao hơn, 1 tháng chắc cũng tầm 1-2 triệu gì ấy. Thuở xưa mình ra APA, ACET hỏi, khóa nào cũng 17-25 triệu, mắc quá nên mình đành tự học ở nhà. Do đó, các bạn cần xác định phải kiên trì và nghiêm túc khi học thi IELTS.

Ôn thi IELTS

Như đã chia sẻ ở bài trước, việc ôn thi của mình chia làm 3 giai đoạn: Ôn kiến thức nền tảng, luyện kĩ năng thi, thi thử. Những sách mình giới thiệu đều có thể dễ dàng tìm ebook trên mạng hoặc mua ở các nhà sách lớn.

Giai đoạn 1: Ôn kiến thức nền tảng

Giai đoạn 1 này kéo dài khá lâu, khoảng 3 tháng tính từ lúc mình xác định sẽ thi IELTS. Giai đoạn này chủ yếu dùng để luyện kĩ năng tiếng Anh:

- Mình dành khoảng 1 tháng lên VOA Learning English để tập nghe lại và viết ra. VOA nói chậm nên dễ nghe hiểu, tuy nhiên bạn phải tập nghe xong 1 câu, stop và viết lại câu đó. Ban đầu việc này khá khó, về sau mình quen dần. Mỗi ngày bạn chỉ cần bỏ ra khoảng 1 tiếng, nghe và viết cỡ 3-4 clip trên VOA là được.
- Sau khi đã quen, ta có thể lên TED.com để nghe các bài nói. Các bạn nên chọn những bài nói dài 10-15 phút, nghe 1 lần không phụ đề và đoán, sau đó nghe lại có phụ đề để xác minh lại. Từ vựng các bài nói trên TED khó hơn VOA nhiều, đôi khi người nói cũng không nói giọng chuẩn, tập nghe những bài này sẽ giúp khả năng nghe tiếng Anh được cải thiện.
- Rèn lại từ vựng và phát âm: cuốn Cambridge Vocabulary for IELTS khá đủ từ vựng Academic English và mấy cuốn English Pronunciation in Use dạy cách phát âm.

Giai đoạn 2: Luyện kĩ năng thi

Ở giai đoạn này, bạn sẽ tập làm quen mới cấu trúc đề, những dạng câu hỏi thường gặp, luyện kĩ năng thi:

- Lần lượt học 2 cuốn IELTS Graduation Student Book và Study Skill. Hai cuốn này giới thiệu khá đầy đủ các dạng câu hỏi thường gặp khi thi IELTS, có bài tập để ôn luyện.

- Sau khi học xong 2 cuốn này, chuyển qua 3 cuốn Improve your IELTS Listening and Speaking, Reading, Writing.
- Phần Writing là phần khó nhất với nhiều bạn (Đề thi tiếng Anh tốt nghiệp dễ vậy mà còn nhiều người bỏ trắng). Với mình nó cũng khá khó nên mình có tham khảo thêm một số tài liệu: Writing Simon và SlidePDF by NgocBach. Do tự học nên chúng ta có phần khá thiệt thòi khi không có người chấm điểm writing hay luyện speaking cùng.
- Ngoài sách ra, các bạn nên lên trang web ielts-simon.com để đọc và làm bài tập. Tác giả Simon là người ra đề/chấm thi IELTS, các lời khuyên ông đưa ra rất hữu ích, bài tập trên trang này cũng rất hay.

Giai đoạn 3: Thi thử

Sau khi ôn luyện đầy đủ, các bạn có thể làm thử đề thi IELTS trong các quyển Cambridge IELTS Course and Practice Test. Bạn có thể ra nhà sách mua chúng với giá 90k 1 cuốn kèm đĩa, hoặc tự in ebook thì rẻ hơn.

Nếu còn thời gian rảnh, bạn có thể xem một số sách về các mánh khoé khi thi để rút kinh nghiệm nhằm tránh mất điểm vô duyên:

- Common Mistakes at IELTS Intermediate
- IELTS Target Bank 7
- Tips for IELTS by Sam McCarter

Lời kết

Nếu cảm thấy lười và không có động lực học tập, các bạn hãy đi đăng kí thi trước. Lệ phí thi IELTS là 4 triệu rưỡi, đóng tiền xong đảm bảo bạn sẽ cắm đầu vào học, không dám nghỉ vì xót tiền cho mà xem. Ngày xưa mình đăng kí thi vào cuối tháng 2, vì xót tiền nên học trâu bò cả Tết luôn. Chúc các bạn may mắn và đạt điểm cao.

Tóm tắt:

- Đừng nghĩ rằng học tiếng Anh là một việc gian khổ, hãy biến nó thành thói quen.
- Nếu muốn tăng khả năng tiếng Anh, ngoại trừ việc học, bạn có thể tập cho mình một vài thói quen giải trí bằng tiếng Anh như chơi game, đọc truyện, đọc sách, vừa vui vừa bổ.
- Để đạt kết quả cao trong các kì thi tiếng Anh, cần phân bố thời gian và thực hiện ba giai đoạn: Luyện kĩ năng tiếng Anh, tìm hiểu và làm quen cấu trúc đề, thi thử.
- Để có thêm động lực học, hãy đăng kí và đóng lệ phí thi trước. Áp lực sẽ khiến bạn tập trung học hơn.

Trung 0335740004

HỌC THUẬT TOÁN ĐỂ LÀM CÁI QUÁI GÌ?

Cấu trúc dữ liệu và thuật toán là một môn học rất thú vị nhưng cũng khá khó nhằn. Mình thường nghe các bạn sinh viên tranh luận về thuật toán. Ý kiến của các bạn được chia làm hai luồng trái chiều như sau:

- Thần thánh hóa thuật toán: Muốn lập trình giỏi phải giỏi thuật toán. Các công ty lập trình lớn toàn phỏng vấn về thuật toán còn gì!
- Coi thường thuật toán: Thuật toán là cái thứ vô dụng, mấy anh đi làm nói là có dùng bao giờ đâu.

Bài viết này sẽ giúp các bạn trả lời câu hỏi "Học thuật toán để làm cái vẹo gì?", cũng như có cái nhìn khách quan hơn về thuật toán.

Học thuật toán để... trả lời phỏng vấn

Những người bênh vực thuật toán thường bảo rằng: Các công ty lớn như Google, Amazon, Facebook rất quan tâm tới thuật toán khi phỏng vấn. Điều này là hoàn toàn có thật! Bạn sẽ phải viết code lên bảng, giải thích code và thuật toán khi phỏng vấn ở các công ty này.

Tuy vậy, không phải công ty lập trình nào cũng như Google, Amazon, Facebook. Theo kinh nghiệm của mình và bạn bè, các công ty Việt Nam tập trung nhiều vào khả năng suy nghĩ logic và giải quyết vấn đề, chỉ hỏi một số thuật toán cơ bản khi tuyển dụng (Bạn mình phỏng vấn vị trí ở công ty An. bị hỏi thuật toán dầu loang và Deep First Seatch).

Các công ty này cũng cần tuyển người làm việc được ngay và biết cách sử dụng công nghệ. Do vậy, đừng quá chăm chăm vào thuật toán, mà còn phải bỏ thời gian học hỏi và sử dụng công nghệ nữa nhé. Đừng quá tự tin là giỏi thuật toán thì học công nghệ nhanh thôi. Cần trải qua một thời gian làm việc lâu dài thì mới nắm hết được được điểm mạnh yếu, kĩ thuật và kinh nghiệm khi dùng ngôn ngữ hay công nghệ đó nhé.

Học thuật toán để giải quyết vấn đề và nâng cao tư duy

Thuật toán ở khắp nơi quanh ta. Bản thân Google mạnh mẽ như vậy là nhờ thuật toán tìm kiếm của nó. Chức năng giới thiệu sản phẩm của Amazon có được cũng nhờ thuật toán. Đến cả những tin tức hiện hằng ngày trong New Feed trên Facebook của bạn cũng do thuật toán định đoạt.

Một số lĩnh vực trong lập trình cần sử dụng rất nhiều thuật toán như: render đồ hoạ, mã hóa dữ liệu, driver thiết bị, machine learning, data mining… Mỗi lĩnh vực sẽ cần những thuật toán riêng. Phải hiểu nắm vững các thuật toán này thì bạn mới có thể làm việc trong lĩnh vực đó.

Việc giỏi thuật toán cũng giúp bạn tìm ra hướng giải quyết vấn đề nhanh hơn, viết code mạch lạc hơn. Nắm vững thuật toán và cấu trúc dữ liệu, bạn sẽ ước tính được độ phức tạp của code, đánh giá code chạy nhanh hay chậm, tốn bao nhiêu bộ nhớ, có dễ mở rộng hay hay không. Đây đều là những kĩ năng vô cùng cần thiết. Để thành một lập trình viên giỏi, bạn cần phải rành rẽ thuật toán.

Nhưng đừng quá thần thánh hóa thuật toán!

Ở Việt Nam, do nhiều kì thi tin học đều chú trọng vào phần thuật toán và giải toán nên chúng ta có xu hướng "thần thánh hóa thuật toán". Điều này dẫn đến tình trạng Việt Nam được giải tin học này nọ nhưng chẳng có phần mềm nào nổi bật cả.

Để lập trình giỏi, ta cần rành thuật toán. Tuy nhiên, giỏi thuật toán không có nghĩa là bạn sẽ thành lập trình viên giỏi. Trong một số lĩnh vực khác như phần mềm doanh nghiệp, web, mobile, phần lớn các chức năng chỉ là "thêm bớt xoá sửa". Yêu cầu của các phần mềm này thường hay thay đổi, dẫn đến việc thay đổi code. Lúc này, thuật toán hay, code chạy nhanh không quan trọng bằng việc hiểu đúng yêu cầu khách hàng (requirement), tổ chức dữ liệu, thiết kế cấu trúc code, viết code sao cho dễ đọc, dễ bảo trì.

Ngoài ra, bản chất của ngành lập trình là kế thừa để phát triển. Thay vì viết các thuật toán từ đầu, các bạn có thể sử dụng thư viện có sẵn, hoặc lấy code đã viết và chỉnh sửa lại cho phù hợp.

Ví dụ: Thay vì tự viết Tree, Stack, Linked List, các ngôn ngữ lập trình đều đã được viết sẵn, ta chỉ việc sử dụng. Đến cả những thuật toán phức tạp như nhận diện khuôn mặt, nhận diện ngôn ngữ cũng có API cả rồi. Những thư viện/API này đã được sử dụng nhiều và test nhiều, tối ưu nhiều nên sẽ ổn định và an toàn hơn code bạn tự viết.

Kết luận

Mới xem qua, thuật toán có vẻ khá khó nhằn và phức tạp. Tuy vậy, những kiến thức cơ bản về thuật toán cũng không nhiều. Quanh đi quẩn lại cũng chỉ có: Cấu trúc dữ liệu Stack, Queue, Binary Tree, Linked List, … và một số thuật toán như: Dynamic Programming, Backtrack, DFS, … Bạn hoàn toàn có thể tự học và dần nắm vững chúng.

Nếu bạn không rành thuật toán, không giải được các câu hỏi hóc búa, bạn vẫn có thể là lập trình viên tốt. Lớp mình ngày xưa có hai đứa vừa học làm freelance, làm web PHP kiêm luôn SEO, thu nhập mỗi tháng cũng khoảng 20-40 triệu. Code tốt là code đơn giản, dễ bảo trì, giải quyết được yêu cầu, đem lại giá trị cho người dùng; không phải là code dùng thuật toán cao siêu.

Vì vậy, đừng e ngại thuật toán, nhưng cũng đừng thần thánh hóa nó. Cứ xem nó là một kĩ năng, học nhiều luyện nhiều sẽ giỏi thôi. Để luyện tập, các bạn có thể nghiền ngẫm 2 cuốn sách Cracking the Coding Interview và Elements of Programming Interview để vừa học vừa ôn, sau đó lên hackerrank để thử sức thôi![7]

Tóm tắt:

- Các công ty công nghệ hàng đầu như Google, Amazon, Facebook thường yêu cầu ứng viên giỏi thuật toán. Tuy vậy, các công ty Việt Nam lại chú trọng công nghệ hơn. Các công ty đều đánh giá cao tư duy logic, khả năng giải quyết vấn đề
- Thuật toán ở khắp nơi quanh ta. Giỏi thuật toán sẽ giúp bạn viết code tốt hơn, giải quyết vấn đề nhanh hơn
- Đừng quá thần thánh hóa thuật toán! Code tốt là code đơn giản, dễ bảo trì, giải quyết được yêu cầu, đem lại giá trị cho người dùng

[7] Một số sách khác sẽ được giới thiệu trong bài "Lập trình viên nên đọc những sách gì?"

NHỮNG ĐIỀU TRƯỜNG ĐẠI HỌC KHÔNG DẠY BẠN

Như mình đã nói ở các bài viết trước, có rất nhiều điều chúng ta không được học ở trường. Là một sinh viên, bạn hãy đọc để biết những điều mình còn thiếu, sau đó tự học và bổ sung nhé.

Bài viết khá dài nên mình tạm chia ra làm 3 phần:

- Kĩ thuật lập trình
- Nâng cao giá trị bản thân
- Thành công và thăng tiến trong công việc

Phần 1 - Kĩ thuật lập trình

Ở Việt Nam, đa phần các bạn lập trình viên thường là sinh viên tốt nghiệp ngành Khoa Học Máy Tính (Computer Science). Ngành này thường nặng về khoa học và nghiên cứu. Một số bạn khác thì tốt nghiệp ngành Kĩ Nghệ Phần Mềm (Software Engineer).

Những kiến thức về hệ điều hành, thuật toán và cấu trúc dữ liệu v....v mà trường dạy là vô cùng cần thiết với các developer, mình không phủ nhận. Tuy nhiên, việc viết code, sử dụng ngôn ngữ lập trình và kiến thức thực tế lại hay bị xem nhẹ. Do đó, khi bắt đầu làm việc, đa phần các bạn sẽ thiếu những kĩ năng sau đây:

Cách đọc và viết code

Khi còn ở đại học, khi bạn viết ra code chạy đúng, chạy được, giải quyết xong bài toán tức là bạn giỏi. Trong các kì thi cũng thế. Thế nhưng, trong công việc thì khác. Code chạy đúng là yêu cầu bắt buộc, nhưng code được viết ra còn phải dễ hiểu, dễ đọc, dễ bảo trì và sửa chữa. Vì sao vậy?

Trong ngành phần mềm, code không phải chỉ viết một lần rồi bỏ đó như bài tập. Do yêu cầu của khách hàng hay thay đổi nên ta phải bảo trì, nâng cấp và sửa chữa code thường xuyên. Như mình đã chia sẻ ở bài viết trước, hãy chọn cách đơn giản, dễ hiểu, đừng chọn cách thông minh để rồi không ai hiểu.

Ngày xưa mình cũng từng là sinh viên giỏi, từng nghĩ mình code hay code giỏi... Sau khi đi làm một thời gian, đọc lại đống bùi nhùi "trông như là code" của mình mới nhận ra ngày xưa mình trẻ trâu thế nào.

Trường đại học dạy ta vô số thứ: lập trình hướng đối tượng, tính bao đóng, tính kế thừa v…v, nhưng chẳng ai dạy các bạn về các nguyên lý SOLID hay design pattern[8], những điều lập trình viên nào cũng cần nắm rõ; không ai dạy các bạn cách đặt tên hàm, tên biến, cách comment[9], cách viết API cho dễ sử dụng; không ai dạy design pattern – một thứ để phân loại lập trình viên junior và senior.

Sử dụng IDE, debug

Mình không vơ đũa cả nắm, nhưng một số trường vẫn còn sử dụng phương thức kiểm tra giấy cho các kì thi lập trình. Sau đó giáo viên sẽ đọc và do code của từng học sinh, thật là cực khổ cho cả thầy lẫn trò. Sinh viên code C, C++ trên notepad, hoặc 1 số IDE như DevC sau đó thì chạy code. Đến khi đi làm, nhiều bạn không biết sử dụng IDE như Eclipse, Visual Studio,… không biết dùng Nuget, Maven…

Mình thì may mắn hơn, thời gian code chiếm 70% thời gian học, được sử dụng Visual Studio khi học C++, dùng NetBeans khi học Java, cơ mà cũng tới năm 3 mới bắt đầu biết set breakpoint để debug chương trình. Do đó mình thấu hiểu nỗi khổ của nhiều bạn học lập trình theo kiểu học thuật, không được code nhiều.

Testing, unit test

Trong chương trình học của một số trường vẫn có môn Kiểm thử phần mềm. Tuy nhiên, nhiều sinh viên ngành CS vẫn ngáo ngơ không biết test case là gì, thế nào là black-box, white-box testing. Một số câu khoai hơn như "NUnit, JUnit, Jasmine là gì? Làm sao sử dụng mock, stub, dùng IoC?" càng ít người biết biết. Có người sẽ chu môi: Dào, tao đi code chứ có phải đi làm test đâu. Để làm một developer giỏi, phải chắc rằng code mình viết ra không lỗi. Để đảm bảo code không lỗi, phải có suy nghĩ của một tester, nghĩ ra những case để kiểm thử nó.

Agile Development

Ở trường đại học, chúng ta được học về "quy trình phát triển phần mềm", học về waterfall, agile,…. Tuy nhiên, chúng chỉ là những kiến thức nhàm chán trên giấy mà ai cũng quên ngay sau khi thi. Đến khi bắt đầu làm việc, bạn sẽ ngáo ngơ khi vào daily meeting, planning meeting, ko rõ quy trình … vì không biết Scrum, XP được áp dụng như thế nào[10] (Hồi vào FSOFT mình cũng ngáo ngơ, phải lên scrumtraining để học thêm).

---

[8] Tìm hiểu về chúng trong mục "Kĩ năng cưng cứng" nhé

[9] Hãy đọc bài viết "Luận về comment code"

[10] Scrum/ Agile sẽ được giải thích trong bài "Scurm/Agile trong thực tế"

Source code control system

Đây là một thứ khá đơn giản nên nhà trường cho rằng các bạn có thể tự học được. Hãy nhìn cách các nhóm SV năm nhất, năm 2 khổ sở làm bài tập lập trình nhóm: Mỗi người làm một phần, sau đó họp cả team ghép code lại, mất code là mất luôn (Mình cũng từng trải qua cảnh ấy, cũng may về sau đỡ hơn).

Hậu quả là các sinh viên mới ra trường phải được training lại về cách dùng SVN, dùng Git, hoặc TFS. Hồi mình mới vào FPT Software thì thấy chương trình Fresher của họ có đào tạo cái này cho sinh viên. Cuộc sống của bạn sẽ dễ thở hơn nếu bạn tự trang bị kiến thức về cách dùng Git, SVN cho mình.

Cách dùng thư viện và framework

Do bản chất của chuyên ngành Computer Science, các trường chỉ dạy một số ngôn ngữ như C++, Java để dạy các môn còn lại. Nhiều sinh viên ra trường vẫn không biết dựng một trang web như thế nào, ngôn ngữ này có framework gì hay, làm sao để hiểu và sử dụng API của một thư viện nào đó.

Các trường chỉ dạy một vài mô hình MVC, MVP, MVVP trên giấy, còn cách dùng những thư viện, framework nổi tiếng như: Struts 2, ASP MVC, Ruby on Rails, jQuery,... còn tùy vào khả năng tự học của sinh viên[11].

Phần 2 – Cách nâng cao giá trị bản thân

Nối tiếp phần 1, ở phần này mình sẽ nói chi tiết cách nâng cao giá trị bản thân – một thứ quan trọng không kém khả năng kĩ thuật.

Hơn 90% các bạn sinh viên IT mới ra trường đều muốn có một công việc ổn định, lương cao, thoải mái (Một số bạn khác muốn làm-một-cái-gì-to-tát hoặc khởi nghiệp, mở công ty riêng thì không bàn tới). Bạn cần biết một điều: Một công ty chỉ trả lương cao cho bạn khi họ thấy giá trị của bạn cao, xứng đáng với mức lương họ bỏ ra. Nếu bạn muốn thoải mái lựa chọn công việc, mức lương thoả đáng, bạn phải tự tìm cách để nâng giá bản thân lên trong mắt họ.

Dưới đây là những cách nâng cao giá trị bản thân mà trường học đã không dạy bạn:

Tìm hiểu thị trường

Mấy bạn sinh viên ngành phần mềm chắc không học về "Tài chính vi mô", nhưng hẳn ai cũng biết khái niệm cơ bản về quy luật cung-cầu. Hãy xem sức lao động của bạn

[11] Mình chia sẻ một số kinh nghiệm tự học trong bài "Cách tiếp cận 1 ngôn ngữ/công nghệ mới"

như một loại hàng hóa và tìm cách bán nó cho được giá. Khi cầu thừa, cung thiếu thì giá hàng hóa tăng. Khi cầu thiếu, cung thừa thì giá hàng hóa giảm. Ví dụ: Lương cho developer Ruby on Rails ở VN khá cao, vì cầu khá cao nhưng cung chưa đạt tới. Lương PHP có thể hơi thấp vì số lượng developer PHP quá nhiều[12].

Nếu như bạn vẫn chưa xác định được hướng đi cho mình, hãy chịu khó tìm kiếm thông tin ở các trang web như ITViec, Linkedin để xem mức lương trung bình của các ngôn ngữ là bao nhiêu, ngôn ngữ lập trình nào đang tuyển dụng nhiều … sau đó tự trang bị mình kiến thức về ngôn ngữ, framework đó. Tìm hiểu rõ về thị trường cũng sẽ giúp bạn đỡ hố khi thỏa thuận lương lúc xin việc.

Mở rộng quan hệ

Quan hệ là thứ không thể thiếu trong bất kì ngành nghề nào. Mình biết được công việc hiện tại là nhờ thằng bạn thân giới thiệu. Càng leo lên cao, bạn sẽ càng thấy tầm quan trọng của quan hệ. Quen bạn bè ở nhiều công ty sẽ giúp bạn dễ đánh giá mức lương, dễ nhảy việc, cũng như… dễ kiếm tiền. Các công ty IT hiện nay đều có chính sách reference, chỉ cần giới thiệu được một người bạn vào công ty, các bạn có thể dễ dàng bỏ túi 4-10 triệu.

Làm sao để mở rộng quan hệ? Bạn đã có sẵn một số quan hệ là bạn bè thân quen khi còn học đại học. Tới lúc đi làm, bạn sẽ quen biết nhiều người hơn. Một số mạng xã hội như Facebook, LinkedIn cũng khá hữu ích. Viết blog hoặc viết sách như mình cũng là một cách để mở rộng quan hệ đấy.

Đầu tư vào bản thân

Như đã nói ở phần "tìm hiểu thị trường", sau khi đã biết được những kĩ năng mà các nhà tuyển dụng yêu cầu, những ngôn ngữ hot hiện tại, các bạn cần bỏ thời gian để trang bị cho mình những kĩ năng đó. Hầu như các công ty nước ngoài đều có mức lương nhỉnh hơn công ty Việt Nam đôi chút, các bạn cũng nên đầu tư vào tiếng Anh.

Chúng ta cũng làm outsource khá nhiều cho các công ty phần mềm Nhật, bạn cũng có thể đầu tư cho tiếng Nhật thay vì tiếng Anh nếu muốn làm việc với các công ty Nhật. Cố gắng kiếm cái bằng TOEIC, IELTS hoặc N3, N2, phụ cấp và lương đi kèm mấy bằng này cũng kha khá[13].

Xác định hướng đi

[12] Về việc chọn ngôn ngữ để học, xem bài viết "Học ngôn ngữ lập trình nào bây giờ?"

[13] Xem lại bài viết "Tôi đã học tiếng Anh như thế nào"

Nếu cảm thấy không có hứng thú hoặc không muốn gắn bó lâu dài với việc code, các bạn cũng có thể phát triển theo hướng BA (Business Analyst) hoặc QA. Con đường phát triển cho một developer cũng khá rộng mở.

Nếu muốn tập trung vào code và technical, các bạn có thể đi theo hướng kĩ thuật: Senior Developer => Technical Lead => Software Architecture…. Nếu muốn làm việc với con người, muốn quản lý, các bạn có thể đi theo hướng quản lý: Senior Developer => Team Lead => Project Manager => Programming Manager … Mỗi hướng đi đòi hỏi những kĩ năng riêng [14].

Phương pháp tự học, tự tìm câu trả lời

Không như các ngành khác, kiến thức trong ngành IT rất nhanh hết hạn. Với ngành xây dựng, xây cầu cách đây 50 năm cũng chẳng khác gì xây cầu bây giờ. Với ngành bác sĩ, bệnh cảm cách đây 50 năm triệu chứng cũng giống bệnh cảm bây giờ. Với ngành IT, công nghệ, ngôn ngữ hoặc framework nổi tiếng cách năm 10-15 năm giờ chẳng ai dùng nữa cả (Lác đác còn mấy bác Nhật xài COBOL và VB).

Đó là lý do developer chúng ta phải học rất nhiều, học đủ thứ, học không ngừng nghỉ để không lạc hậu với giới trẻ. Mình từng gặp trường hợp 2 bác senior rớt phỏng vấn vào công ty mình dù họ đã có 5-6 năm kinh nghiệm làm việc. Lý do là họ chỉ rành WinForm và WebForm, … những công nghệ đang ngắc ngoải, và có vẻ họ cũng không muốn học thêm công nghệ mới để tự làm mới mình. Trong truyện Kim Dung, Hồng Thất Công – bang chủ Cái Bang từng nói rằng : Trong võ học, không tiến tức là lùi. Mình nghi ngờ ông từng tốt nghiệp khoa CNTT của trường nào đó, vì câu nói này cũng khá đúng với ngành IT.

Báo chí vẫn phát ngôn nhan nhản rằng: Sinh viên Việt Nam thiếu khả năng tự học. Mình không muốn biện minh gì thêm, vì mình vẫn thấy nhiều bạn nhờ giải bài tập hộ, làm đồ án hộ trên các group facebook[15]. Thiết nghĩ các trường nên hướng dẫn các sinh viên các tự học, cách nghiên cứu, cũng như rèn tính tự lập tự giác cho sinh viên[16].

Phần 3 - Thành công và thăng tiến trong công việc

[14] Giải thích rõ hơn trong bài "Con đường phát triển nghề nghiệp cho developer"

[15] Xem "Thực trạng học lập trình của các thanh niên hiện nay"

[16] Mình có hướng dẫn rất cụ thể trong bài "Cách tiếp cận ngôn ngữ/công nghệ mới"

Trong phần cuối của bài viết, mình sẽ nói về những điều mà bạn-nào-cũng-muốn-biết-nhưng-trường-học-không-dạy, đó là : Cách thành công và thăng tiến trong sự nghiệp.

Mình tổng hợp những điều này một phần từ sách vở, một phần từ pluralsight, một phần nhờ được các anh senior, PM, team leader chia sẻ. Có thể chúng không đúng 100%, nhưng biết những điều này sẽ giúp con đường nghề nghiệp của bạn bằng phẳng và "dễ thở" hơn rất nhiều.

Làm sao để phỏng vấn và xin việc?

Sinh viên vừa ra trường, muốn có tiền nuôi sống bản thân thì phải đi làm. Muốn đi làm thì phải có công việc, mà muốn có công việc thì phải đi nộp đơn xin việc. Thế nhưng nhà trường không bao giờ dạy bạn cách viết 1 CV cho đàng hoàng, cách chuẩn bị, trả lời phỏng vấn ra sao.

Có vô số những điều thuở mình muốn biết từ thời là sinh viên mà không biết hỏi ai[17]:

- Viết và trình bày CV như thế nào để thu hút nhà tuyển dụng
- Chuẩn bị gì trước khi đi phỏng vấn
- Quy trình phỏng vấn thường gặp của các công ty IT
- Những câu hỏi thường gặp khi đi phỏng vấn
- Cách thỏa thuận lương (Nhiều bạn sinh viên giỏi nhưng mới ra trường thường bị "ép giá" lúc thỏa thuận lương. Người thỏa thuận lương với bạn thường là manager, hoặc HR, họ làm chuyện đó thường ngày rồi, nên việc bạn không giỏi thỏa thuận bằng họ cũng không có gì lạ).

Văn hóa ứng xử nơi công sở

Ngành IT là một ngành khá dễ chịu. Bạn không lo mình gặp phải cảnh "ma cũ bắt nạt ma mới", phải "bưng trà rót nước" cho các cụ lão làng, hay lo chia phe phái, tụm năm tụm ba nói xấu lãnh đạo. Dân developer chúng ta tập trung nhiều vào kĩ thuật, do đó có kĩ thuật giỏi thì sẽ được tôn trọng. Tuy nhiên, cũng có một số điều không-được-học-ở-trường mà ta nên lưu ý:

- Đặc điểm của ngành phần mềm là làm việc theo nhóm, do đó các bạn đừng thu mình lại, chịu khó hòa đồng với team: Đá banh chung, chơi game chung, đi nhậu chung,…. Một khi đã thân thì làm việc chung sẽ dễ dàng hơn rất nhiều.

[17] Câu trả lời có thể tìm trong bài "Muôn nẻo đường tìm việc"

- Muốn mọi người biết mình giỏi, hãy thể hiện và chứng tỏ mình giỏi. Đây là một môn nghệ thuật khá khó, khoe nhiều các bạn sẽ mang tiếng là "nổ", là "tỏ thái độ", khoe ít người ta sẽ tưởng các bạn "không biết gì".
- Thông cảm cho các bác cấp trên. Dân lập trình chúng ta thường tưởng mình giỏi, và nghĩ rằng các cụ cấp trên "ngu chết mẹ", có biết gì về code hay lãnh đạo đâu. Thật ra câu này cũng không sai, ngày xưa manager cũng từng là coder như bạn, vì code giỏi bị đẩy lên vị trí manager, có ai dạy họ kĩ năng lãnh đạo đâu. Bị cuốn vào công việc quản lý, họ không có thời gian trau dồi kĩ năng code nữa. Với suy nghĩ "thông cảm" thay vì "coi thường", bạn sẽ dễ chia sẻ tầm nhìn với manager hơn, cơ hội thăng tiến cũng cao hơn.
- Đừng coi thường những người code dở hoặc không biết code. Đôi khi ta thấy một người code không giỏi nhưng lại nhanh thăng tiến, rồi chửi thầm "Tại sao nó code như hạch mà lên chức nhanh thế, chắc do giỏi nịnh sếp". Có khi là do họ rành văn hóa công sở và giỏi kĩ năng mềm hơn bạn đấy!

Những phương pháp thương thảo

Trong suốt quãng đời làm việc, bạn sẽ phải thương thảo vô số lần: Thương thảo với trưởng nhóm và các thành viên trong nhóm khi phân chia công việc, thương thảo với nhân sự khi thỏa thuận lương, thương thảo với manager khi muốn chuyển team, muốn được tăng lương... Do đó việc trau dồi kĩ năng thương thảo sẽ giúp bạn tiến xa hơn trên con đường nghề nghiệp. Mình từng đọc 2 cuốn sách khá hay về thương thảo là: "Đắc nhân tâm" và "Một đời thương thuyết", các bạn có thể tìm đọc.

Cách xây dựng tiếng tăm (reputation)

Câu hỏi luôn đau đáu trong đầu mỗi người đi làm, đó là: Làm sao được tăng lương thăng chức, làm sao được leo lên vị trí cao hơn? Để làm được điều đó, bạn cần xây dựng tiếng tăm trong mắt đồng nghiệp, cấp trên. Làm như thế nào ư?

> Một điều khá hay trong ngành IT là: Vì đây là một ngành nặng về kĩ thuật. do đó bạn chỉ cần giỏi tập trung trau dồi technical cho giỏi. Chỉ cần giỏi kĩ thuật, bạn sẽ được đồng nghiệp coi trọng, cấp trên tin tưởng giao phó trách nhiệm. Chỉ cần giỏi kĩ thuật, con đường sự nghiệp của bạn sẽ rộng thênh thang, bạn sẽ nhanh chóng leo lên vị trí senior, team leader,... Chỉ cần technical giỏi, lương bạn sẽ tăng vù vù, từ 500$, 1000$, 2000$, các quảng cáo tuyển dụng toàn cần người giỏi kĩ thuật l còn gì?

Các bạn vừa đọc vừa gật gù đồng ý với những điều mình viết trong đoạn trên? THẬT Á? TỈNH LẠI ĐI BẠN TRẺ À! Con đường nghề nghiệp của một lập trình viên không đơn giản bằng phẳng như vậy đâu. Chúng ta thường lầm tưởng rằng: code giỏi + kĩ thuật giỏi = lập trình viên giỏi. SAI BÉT.

Một lập trình viên cần nhiều hơn thế: kĩ năng lập trình, kĩ năng giao tiếp, kĩ năng thuyết trình, kĩ năng giải thích, … Để lấy yêu cầu từ khách hàng, bạn phải biết cách giao tiếp, biết cách hỏi, biết cách giải thích. Để thuyết phục mọi người trong nhóm làm theo mình, bạn cần rành kĩ năng chém gió, kĩ năng thuyết trình.

Đó là lý do một số người kĩ thuật không cứng nhưng vẫn lên được vị trí team leader, PM, nhờ họ giỏi "chém gió", giỏi thuyết phục và quản lý người khác. Thậm chí, kể cả một số vị trí thiên về technical như: senior dev, technical lead,… bạn vẫn sẽ phải hướng dẫn developer mới, đưa ra solution và giải thích, những kĩ năng mềm này càng không thể thiếu được.

Dưới đây là một số cách để xây dựng tiếng tăm. Một số cách mình được truyền lại, một số cách mình đã áp dụng, kết quả cũng khá ổn:

- Giúp đỡ đồng đội của bạn: Đừng vội lao vào ngỏ ý giúp đỡ, họ sẽ nghĩ là bạn ra vẻ hay nhiều chuyện. Hãy thể hiện mình biết nhiều và luôn sẵn lòng giúp đỡ, khi có khó khăn họ sẽ thử nhờ bạn giúp. Dần dần khả năng kĩ thuật thái độ làm việc của bạn sẽ làm bạn được các thành viên khác nể trọng. Họ sẽ có nhận xét tốt về bạn. và cấp trên đương nhiên sẽ để ý bạn hơn. (Lời khuyên: Hãy chân thành giúp đỡ người khác, đừng quá chú ý tới tiếng tăm, bạn sẽ có được đồng đội và bạn bè tốt. Quá tập trung vào tiếng tăm sẽ gây phản tác dụng).
- Thể hiện với leader: Hãy ước đoán công việc nhiều một tí, sau đó hoàn thành sớm thời hạn (Đừng lạm dụng chiêu này nhiều lần, sẽ phản tác dụng). Sau khi hoàn thành công việc, hãy tự tìm việc làm hoặc báo cáo leader để giao việc. Leader sẽ đánh giá cao sự nhiệt tình và thái độ làm việc của bạn.
- Giữ thái độ khiêm tốn. Người khiêm tốn thường được tôn trọng hơn. Rất nhiều bạn sinh viên mới ra trường thường giữ thái độ "ta đây giỏi", làm gì cũng hất mặt lên trời, dễ bị đánh giá là "attitude không tốt".
- Thực hiện các buổi seminar, thuyết trình, giới thiệu công nghệ: Khi bạn chia sẻ kiến thức của mình cho người khác, bạn sẽ nhận lại được kiến thức và sự nể trọng. Đó cũng là lý do mà mình viết blog, trả lời câu hỏi trên stackoverflow.

Lời kết

Mình không có ý chê trách các trường đại học khi viết bài viết này. Về bản chất, các trường chỉ dạy cho bạn kiến thức cơ bản, nó sẽ là nền tảng giúp bạn học những điều mới dễ dàng hơn. Đó cũng là lý do mà cuốn sách này ra đời, để chia sẻ cho những bạn sinh viên còn đang học, mới ra trường hoặc vừa đi làm những điều mà chỉ-đi-làm-rồi-mới-nhận-ra.

Tóm tắt:

- Những kiến thức về code, về kĩ thuật trong trường đại học là không đủ để đi làm, do đó bạn cần rèn luyện khả năng tự học, tự nghiên cứu.
- Để có mức lương cao, công việc thú vị, bạn phải biết cách tìm hiểu thị trường, nâng giá bản thân.
- Muốn thăng tiến và thành công trong công việc, bạn cần biết cách phỏng vấn, văn hóa ứng xử nơi công sở, cách xây dựng danh tiếng và thương hiệu cá nhân.

TẠO ĐỘNG LỰC HỌC TẬP VÀ LÀM VIỆC – SỨC MẠNH CỦA THÓI QUEN

Thời sinh viên, đã bao giờ bạn muốn làm bài tập, ôn thi ngay nhưng lại bị games, đi chơi, gái gú cám dỗ chưa?

Thời sinh viên, đã bao giờ bạn muốn lấy một tấm bằng ngoại ngữ, một chứng chỉ, nhưng tìm học được nửa tiếng rồi lại thôi chưa?

Lúc đi làm, đã bao giờ bạn muốn học một công nghệ mới, một ngôn ngữ mới, nhưng được một vài hôm lại thấy chán nản và muốn bỏ chưa?

Nếu câu trả lời của bạn là "Có", đừng lo, chẳng có gì xấu hổ cả, ngày xưa mình cũng từng như bạn (giờ vẫn vậy). Tuy nhiên, nhờ một vài bí quyết đơn giản, mình đã cảm thấy tự tin, dễ dàng khi học và tiếp thu kiến thức mới, có được một công việc kha khá, cũng như một số tấm bằng kha khá.

Đừng quan trọng hóa mục tiêu

Bí quyết tạo động lực học của mình không quá phức tạp.

> Hãy đặt ra những mục tiêu ngắn hạn, dài hạn cho mình. Hãy nghĩ tới niềm vui mình sẽ có khi đạt được mục tiêu, những buồn bực thất vọng khi không hoàn thành. Sau đó lên kế hoạch và thực hiện mục tiêu đó.

Bạn đang gật gù: "Ồ, ra là vậy, từ giờ mình sẽ chăm chỉ đặt mục tiêu rồi chăm chỉ làm theo". Xin lỗi, đoạn trên mình chém gió đấy, cái thứ "mục tiêu" sáo rỗng đó chẳng giúp ích cho bạn mấy đâu. Đặt mục tiêu được vài ngày là bạn sẽ lại chán và bỏ cuộc cho xem.

Vô số sách vở đã nói về tầm quan trọng "mục tiêu", cho rằng nó là kim chỉ nam thành công. Cá nhân mình lại cho rằng chúng ta đã quá đề cao nó. Đặt ra mục tiêu sẽ đặt áp lực lên bản thân bạn. Khi không đạt được mục tiêu đó, bạn sẽ dễ thấy thất vọng về chính mình. Đặt ra mục tiêu, ta nghĩ rằng mình có thể dự đoán được tương lai, trong khi "thế sự vô thường, đường đời khó đoán".

Sức mạnh của thói quen

Thật ra, cách mà mình tự tạo động lực và ép bản thân tiến bộ chính là:

> Thay vì đặt mục tiêu, dồn động lực để thực hiện, hãy xây dựng các thói quen. Chỉ cần gắng sức xây dựng được thói quen tốt, bạn sẽ dần dần đi đúng hướng.

Nếu để đam mê dẫn đường, bạn sẽ cháy hết sức mình vào một công việc nào đó. Rồi sau một tuần, một tháng, khi đam mê tắt dần, bạn sẽ thấy chán nản rồi bỏ dở. Ngược

lại, để thói quen dẫn đường, mình luôn bắt đầu bằng mọi việc những bước tiến nho nhỏ, từ từ.

Bản thân mình định lập blog từ tháng 10 năm 2015, nhưng ngần ngừ lười viết tới tháng 12 mới làm. Sau khi bắt đầu viết, mình chỉ cố gắng duy trì thói quen ra 2 bài/tuần. Đến bây giờ, mình đã hình thành được thói quen ngồi viết blog mỗi thứ 3 thứ 5. Dù nhiều khi mình không có động lực. cũng chẳng mấy hào hứng khi ngồi vào máy, mình vẫn ngồi xuống và viết, vì nó đã thành một thói quen.

Đơn giản vậy đấy! Mình dễ học công nghệ mới vì luyện được thói quen đọc sách công nghệ, xem video pluralsight. Mình tự học được tiếng Anh vì luyện được thói quen bỏ 2 tiếng mỗi ngày để học sau khi đi làm về. Khi đã thành thói quen, bạn có thể học tập, làm việc dễ dàng, không cần động lực. Điều mình thật sự muốn chia sẻ là:

> Hãy để đam mê và động lực chỉ đường cho bạn thấy điều bạn muốn làm, chứ đừng để chúng dẫn đường. Chính kỉ luật và thói quen mới là người dẫn đường cho bạn.

Dần dần, bạn sẽ xây dựng lòng tin vào bản thân – tôi có thể biến mọi chuyện thành thói quen, tôi có thể làm mọi thứ. Tin mình đi, mọi thứ chỉ mệt mỏi ở giai đoạn khởi đầu thôi. Khi tất cả đã thành một thói quen, bạn có thể làm những chuyện "cực nhọc, khó khăn" một cách vui vẻ và thích thú.

Tóm tắt:

- Đừng quan trọng hóa mục tiêu, nó sẽ đặt áp lực lên bản thân bạn
- Hãy dần dần luyện cho mình những thói quen tốt, chính chúng sẽ tiếp thêm động lực và dẫn đường cho bạn

THỰC TRẠNG HỌC LẬP TRÌNH CỦA MỘT SỐ THANH NIÊN HIỆN NAY

Lưu ý: Bài viết dùng khá nhiều từ ngữ nặng nề nên có thể các bạn sinh viên sẽ cảm thấy khá "nhột khi đọc".

Thực trạng học lập trình của các "sinh dziên Ai Ti"

Để quảng bá blog và rèn luyện khả năng kĩ thuật, mình tham gia khá nhiều group lập trình trên Facebook. Các group này thường đăng tin quảng cáo tuyển dụng công việc, hoặc có các đường link tới các bài viết vô cùng bổ ích. Các bạn lập trình viên đang đi học học hoặc mới ra trường cũng nên tham gia.

Tuy nhiên, điều khiến mình bực mình nhất là nhiều bạn lại sử dụng các group này để làm kênh… nhờ giải bài tập, fix bug, thi hộ. Hỏi vấn đề khi làm việc thì còn đỡ, đằng này mình thấy nhiều bạn còn post bài tập lên nhờ giải giùm, hoặc post đề thi lên kèm dòng chữ "Giúp em với, em đang thi". Thật lòng mà nói, mình cũng không biết biết các bạn trẻ này tốn tiền học đại học làm gì nữa.

Sự tai hại của việc nhờ làm bài hộ

Học lập trình ở Việt Nam rất chán. Năm đầu, bạn sẽ phải nhồi vào đầu mớ kiến thức đại cương nặng nề, về sau mới được học lập trình. Số môn lập trình cũng không nhiều, vì vậy bài tập và bài thi là cơ hội để các bạn rèn luyện kĩ năng, luyện tư duy lập trình. Hai trong số các kĩ năng quan trọng nhất của lập trình viên là: kĩ năng fix bug và kĩ năng tự học[18]. Các bạn trẻ, làm ơn bỏ giùm cái thói hễ code không chạy, hễ gặp bug là vác đi hỏi rồi rung đùi chờ người trả lời!

Tuy các group không phải là nơi fix bug chùa, thế nhưng một số bạn lại rất thích làm "người tốt việc tốt", sẵn sàng bỏ thời gian viết code hộ, Team Viewer fix bug hộ. Các bạn tưởng mình đang giúp người khác, nhưng thật ra làm vậy sẽ tạo thói quen ỷ lại, thui chột kĩ năng của chính những bạn được giúp.

Thời còn đi học, mình cũng biết nhiều bạn cũng không chịu làm bài, tới lúc làm đồ án, gần ra trường thì lại chẳng viết được một chức năng nào. Đây là hậu quả của việc nhờ làm bài, nhờ fix bug hộ. Mà thôi kệ, những bạn không code được ra trường thì không dành công ăn việc làm với mình được, càng đỡ phải lo. Lúc đi phỏng vấn chắc các bạn ấy cũng không kịp lên facebook hỏi hay nhờ người trả lời hộ đâu.

---

[18] Xem ở bài "Hướng dẫn tiếp cận một ngôn ngữ/công nghệ mới" và "Cách trở thành cao thủ trong việc fix bug"

Ơ, vậy không đi hỏi thì phải làm sao?

Như mình đã nói, kĩ năng tự học, tự tìm hiểu là một trong những kĩ năng quan trọng của lập trình viên. Trước khi hỏi thì các bạn làm ơn chịu khó Google trước giùm mình, Google tiếng Việt không ra thì Google tiếng Anh nhé. Nếu tiếng Anh yếu hay không giỏi thì …. chịu khó học giùm mình, học reading thôi cũng được, ngành này mà tiếng Anh kém thì hơi khó sống[19].

Sau này ra trường đi làm, bạn không thể nhờ người khác fix bug hay code phụ. Lấy stackoverflow ra làm ví dụ: Đây là trang hỏi đáp lớn nhất cho giới lập trình viên, các thành viên của họ đều rất ghét việc nhờ giải bài tập, nhờ fix bug hộ. Thay vì đăng câu hỏi lên group, hãy đọc kĩ tài liệu hoặc tự Google trước, đa phần các lỗi hay vấn đề bạn gặp đều có người trả lời cả rồi đấy.

Tóm tắt:

- Một bộ phận sinh viên IT hiện nay khá lười, thích đi hỏi bài tập hoặc nhờ giải hộ, sửa hộ
- Việc này sẽ làm bạn không có đủ khả năng sau khi ra trường, rất khó tìm được công việc
- Trước đi khi hỏi người khác, hãy tự đọc tài liệu hoặc Google tìm hiểu trước

[19] Xem lại bài "Tôi đã học tiếng Anh như thế nào"

Giai đoạn 2 – Ra đời

Đây là giai đoạn bạn gần ra trường, sắp đi làm. Ở giai đoạn này, do đã có kiến thức nền tảng vững từ những môn học ở trường, bạn cần tập trung tìm hiểu thêm về ngành nghề, đồng thời tự trang bị những kĩ năng cần có để xin việc.

THAY LỜI MUỐN NÓI – GỞI TỚI NHỮNG NGƯỜI THÂN YÊU CỦA MỖI LẬP TRÌNH VIÊN

Lời dẫn (cho người thân yêu của một lập trình viên nào đó)

Chào bạn.

Có thể bạn đọc được bài viết này khi vô tình xem cuốn sách "Code dạo kí sự". Hoặc có thể bạn được người yêu, bạn bè hay con cái gửi bài viết này cho đọc.

Đây là một bài viết về những người làm trong ngành lập trình. Đằng sau vẻ ngoài hào nhoáng (việc nhẹ lương cao), ngành lập trình luôn có những cái khổ riêng. Như tựa đề, bài viết này nói hộ nỗi lòng của các lập trình viên, những điều họ muốn nói mà không biết chia sẻ cùng ai.

Nếu bạn được một lập trình viên gửi bài viết này, hãy biết rằng bạn là một người thân thương quan trọng đối với lập trình viên đó, và họ đang muốn bạn biết thêm về nghề nghiệp cũng như con người của họ. Hãy cố gắng đọc hết bài viết để có thể dễ dàng thấu hiểu và cảm thông với họ hơn nhé.

Lời dẫn (cho lập trình viên)

Chào bạn.

Trong mắt nhiều người, ngành lập trình là một ngành khô khan, còn lập trình viên là những thằng ngáo ngơ thiếu tinh tế, suốt ngày chỉ biết nhìn đăm đăm vào máy tính.

Bài viết này được viết dưới góc nhìn của một lập trình viên như bạn, nhằm giúp mọi người có một cái nhìn đúng hơn về những lập trình viên như chúng mình. Do đó, bài viết chỉ sử dụng từ ngữ đơn giản dễ đọc, không cầu kỳ hoa mỹ hay rắc rối phức tạp.

Nếu được, các bạn hãy đưa bài viết này cho những người thân yêu của bạn (bạn bè, người thân hay người yêu) thay cho những lời tâm sự mà bạn muốn nói nhé. Biết đâu, sau khi đọc xong, họ sẽ hiểu và thông cảm với bạn và nghề nghiệp của bạn hơn.

Về tính cách – Có thật lập trình viên chỉ là lũ dở người, lầm lì ít nói?

Các từ "đụt, ngố" hoặc "ngáo ngơ, dở người" thường được dùng để miêu tả lập trình viên. Trong mắt đa số người, dân lập trình là những kẻ khô khan, lầm lì, suốt ngày chỉ biết cắm mặt vào máy tính. Thật vậy chăng?

Có một sự thật mà ít người biết là: Đa số lập trình viên có tính cách hướng nội (Những bác hướng ngoại thì đi làm một thời gian lên team leader hoặc quản lý hết cả rồi). Họ có đầu óc logic, hành động dựa theo lý trí, luôn muốn tìm hiểu bản chất vấn đề.

Với đầu óc logic, chúng tôi thường muốn mọi việc phải rõ ràng rành mạch. Do cả ngày chỉ biết dùng những dòng code khô khan nói chuyện với máy tính nên chúng tôi không biết nói những lời lãng mạn bóng bẩy như các soái ca trong ngôn tình, cũng như hơi thiếu sự tinh tế trong cư xử.

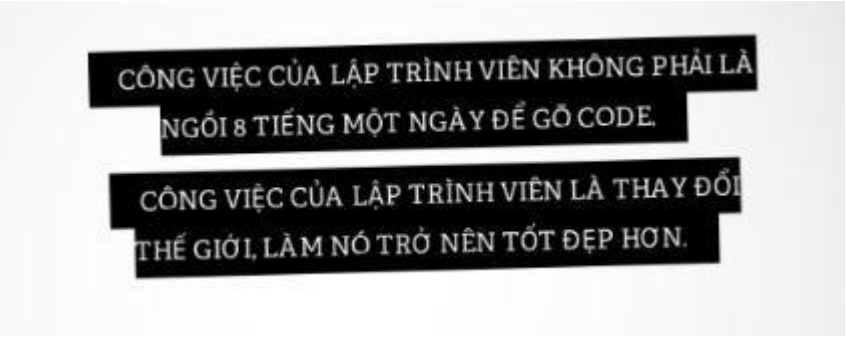
Thêm nữa, những thứ mà chúng tôi quan tâm thường có phần "khác lạ" so với đại chúng. Trong khi người khác trầm trồ về sự mạnh mẽ, tiện dụng của Facebook, Google,… chúng tôi lại tìm cách mổ xẻ cách thức hoạt động của chúng rồi tìm cách xây dựng thứ tương tự. Trong khi người đời xôn xao vì Hariwon và Ngọc Tờ-rinh lộ clip nóng 18+ thì coder chúng tôi lại xôn xao khi nghe Xamarin, .NET lộ mã nguồn mở. Do vậy, chúng tôi thường khó tìm được điểm chung với người khác khi giao tiếp, dẫn tới sự "lầm lì, ít nói" thường thấy.

Quen một lập trình viên, các bạn nữ sẽ hơi thiệt thòi khi ít nhận được những cử chỉ ga lăng, những câu tình cảm âu yếm. Tuy vậy, lập trình viên chúng tôi lại rất nghiêm túc, thật thà, chung thủy, không lăng nhăng, khi cần thì lại rất mạnh mẽ cho các bạn nữ dựa dẫm.

Về công việc – Bọn coder là lũ code dạo, suốt ngày chỉ biết gõ gõ trên máy tính?

Nhìn từ bên ngoài, công việc của một lập trình viên chúng ta chỉ là ngồi vi tính và gõ gõ 8 tiếng mỗi ngày. Cực kỳ sai lầm!!! Công việc của chúng ta là thay đổi thế giới, làm cho nó trở nên tốt đẹp hơn.

Không tin à? Tiki, Lazada bạn hay vào mua sắm, Camera360 bạn hay dùng tự sướng, game Candy Crush mà bạn chơi,… đều là sản phẩm tim óc của lập trình viên. Những thứ lập trình chúng ta làm ra giúp cho công việc thuận lợi trôi chảy hơn, cuộc sống dễ dàng thoải mái hơn.



Thật lòng mà nói, lập trình là một công việc thú vị nhưng cũng… rất rất khó, đòi hỏi rất nhiều nơ-ron thần kinh và chất xám. Đôi khi coder chúng tôi luôn bị công việc ám ảnh, đi chơi với gấu hay gia đình mà cứ nghĩ tới một tính năng chưa hoàn thành, một lỗi còn sót chưa sửa được,… tội chưa??

Một cái khổ khác của nghề lập trình là OT (làm thêm giờ)[20], những khi dự án cháy, chúng tôi phải thức thâu đêm suốt sáng hoặc ở lại công ty để hoàn thành nhiệm vụ, tiền lương không đủ bù tiền thuốc. Vì vậy, bạn nữ nào có gấu hoặc chồng làm ngành IT nhớ cảm thương và thông cảm nhé, đừng giận hờn trách móc mà tội tụi tôi lắm.

À quên, lập trình viên tụi tôi là người viết chương trình (thay đổi thế giới) chứ không phải là thợ sửa máy hay thợ cài win, do đó xin quí vị đừng nhờ vả kiểu "anh ơi cài giùm em cái Win" hoặc "cháu ơi sao bác vào trời đất xem phim mà chậm quá". Bản thân tụi tui đi làm cũng có tự cài Win đâu, phải nhờ tới mấy anh chuyên IT cơ.

Thay lời kết

Không có lập trình viên, thế giới sẽ không có Camera360 (để sống ảo), không có Google (để tìm tài liệu Copy Paste), Việt Nam sẽ không có Zalo, Mp3 Zing, webtretho hay vozforum (để vào đọc truyện và chém gió). Nhờ lập trình viên, các bạn mới có thể vi vu mà lướt Facebook để hóng hớt và chim chuột.

Ẩn sau vẻ ngoài trầm mặc ít nói là một cái đầu đầy kiến thức, ham học hỏi và rất hài hước và thú vị đấy. Đa phần các anh lập trình viên đều nhát gái, ít nói, sợ người lạ, các chị các em cứ chủ động tấn công nhé, có người yêu là coder thì không lo về chuyện lăng nhăng hay không chung thủy này nọ đâu. Lương lập trình viên cũng khá cao [21], cũng đủ nuôi thân và nuôi vợ nuôi con nữa.

Tóm tắt:

- Đa số lập trình viên thường sống hướng nội, quan tâm công nghệ nên thường bị hiểu lầm là ngố, đụt, lầm lì
- Công việc của lập trình viên khá nặng nề, dễ stress lại hay đòi hỏi nhiều thời gian, nhớ thông cảm cho họ
- Công việc của lập trình viên là thay đổi thế giới, xin đừng nhờ họ cài Win hay sửa máy tính
- Lương lập trình viên khá cao so với mặt bằng chung, họ cũng nhát gái, chung thủy, là mục tiêu lý tưởng cho các chị em

[20] Xem thêm trong bài "Mặt tối của ngành công nghiệp IT"

[21] Xem trong bài "Con đường nghề nghiệp của lập trình viên"

HỌC NGÔN NGỮ LẬP TRÌNH NÀO BÂY GIỜ?

Đây một câu hỏi mà mình thường nhận được từ các em sinh viên mới ra trường, mới vào đại học, hoặc chưa biết gì về lập trình: "Giờ mình nên học ngôn ngữ lập trình nào đây?".

Nghe đơn giản, nhưng đây là một câu hỏi có độ khó khá cao, sánh ngang với câu "Em nên làm nghề gì, học đại học nào?" của các em học sinh cấp 3. Trong phạm vi bài viết này, mình sẽ đưa ra một số dữ liệu tham khảo và lời khuyên cá nhân.

Trước khi hỏi câu này, hãy tự hỏi : Mình muốn học lập trình để làm gì?

Khi được hỏi "Giờ mình nên học ngôn ngữ lập trình nào đây?", mình luôn hỏi lại câu này "Bạn/Em muốn học lập trình để làm gì?". Trả lời được câu hỏi này, bạn đã xác định được 50% ngôn ngữ mình cần học. Dưới đây là một số câu trả lời mình hay nhận được.

1. Em vừa ra trường, trường chỉ dạy C, C++, … giờ em cần học ngôn ngữ gì để dễ kiếm việc làm, lương cao?

Thị trường việc làm IT hiện tại khá rộng, tạm chia làm 3 mảng: embedded (lập trình nhúng), web và mobile. Thị phần mảng Game khá nhỏ nên mình không nhắc đến.

- **Mảng embedded**: yêu cầu khá cao về trình độ, sử dụng ngôn ngữ lập trình C, C++, có thể dùng Java. Nếu bạn là lập trình viên C++ cứng, mức lương rất khá và mức độ cạnh tranh cũng ko nhiều.
- **Mảng mobile**: Chiếm thị phần cao nhất vẫn là app cho Android viết bằng Java, tiếp theo là app cho IOS, viết bằng Objective-C[22]. Java là một ngôn ngữ khá dễ học, độ phổ biến cũng cao, ứng dụng rộng. Với kiến thức Java bạn cũng có thể chuyển qua mảng Web.
- **Mảng web**: Để có thể trở thành lập trình viên Web, bạn phải biết lập trình front-end (Dùng HTML/CSS và ngôn ngữ JavaScript). Sau đó học một ngôn ngữ lập trình back end (C#, Java, PHP)[23].

Nếu muốn học để kiếm tiền, hãy xác định mình muốn làm mảng công việc nào, sau đó tìm các khảo sát trên mạng và nghiên cứu mức lương trung bình của developer ngôn ngữ đó.

[22] Xem kĩ hơn trong "Tổng quan về lập trình ứng dụng di động"

[23] Xem kĩ hơn trong "Kĩ năng cần có của Web Developer"

Hiện tại, có khá nhiều PHP developer nên lương trung bình của PHP developer hơi thấp so với số còn lại. Ngoài ra, một số ngôn ngữ như Rails, Python,... ít người học, developer hiếm nên biết các ngôn ngữ này sẽ có thu nhập khá hơn.

2. Mình muốn làm website hoặc ứng dụng cho người nhà, bản thân v....v.

Đây là câu trả lời mình nhận được từ một số bạn học tài chính ngân hàng, kinh tế,... Nếu bạn muốn làm một ứng dụng di động, Java là lựa chọn tốt nhất. Để tạo website, hiện tại có rất nhiều hướng dẫn tạo website bằng Joomla, Drupal, Wix,... mà ko cần kiến thức lập trình. Các bạn có thể học thêm PHP để có thể tùy biến, thêm tính năng cho trang web.

Lựa chọn thật ra không quan trọng, học một ngôn ngữ mới là chuyện đơn giản

Đọc tới đây, hẳn nhiều bạn sẽ bảo rằng mình nổ, hoặc nghĩ rằng "dám chắc thằng này không phải coder, phán như thánh". Trước khi ném đá, hãy bình tĩnh nghe mình giải thích và trình bày.

Mình cũng từng là sinh viên IT như các bạn. Môn đầu tiên về lập trình mình được học khi vào đại học là: "Cơ bản lập trình với C". Mình từng điên đầu với khai báo biến, tách hàm, điều kiện, vòng lặp, IO.... Môn tiếp theo là "Lập trình hướng đối tượng với C++". Phải thú thật C++ không phải là ngôn ngữ phù hợp để học hướng đối tượng. Mình từng nhầm lẫn trước các khái niệm "tính bao đóng, tính kế thừa".

Do đó, bản thân mình cũng biết sự khó khăn gặp phải khi học một ngôn ngữ. Tuy vậy, mình vẫn khẳng định học một ngôn ngữ mới là chuyện đơn giản.

Vì sao? Hãy tự xem lại kiến thức lập trình bạn có được khi vừa ra trường:

- Học qua 1,2 ngôn ngữ gì đó
- Cấu trúc dữ liệu và thuật toán
- Thiết kế, truy vấn cơ sở dữ liệu
- Design pattern (Có thể)
- Khả năng design

Khi mới tiếp cận lập trình, chúng ta cảm thấy khó khăn vì phải làm quen với vô số khái niệm mới. Tuy nhiên, khi đã có kiến thức cơ bản, việc tiếp cận ngôn ngữ mới trở nên rất dễ dàng. Chúng ta học gì khi học một ngôn ngữ lập trình mới? Đây là câu trả lời:

- Cách khai báo hàm, biến
- Cách khai báo vòng lặp, điều kiện if/else
- Các kiểu cấu trúc dữ liệu: list, set, tuple, ...
- IO, multi-thread, delegate, event
- IDE phù hợp, cách build, debug

- Các framework, cách sử dụng, ….

Nếu bạn đã biết cách viết for, if/else, while … trong Java, khi chuyển qua học C# hoặc javascript, cấu trúc hàm for, if/else… vẫn giữ nguyên. Kiến thức của bạn được kế thừa từ ngôn ngữ lập trình trước, do đó việc học sẽ diễn ra nhanh hơn. Hoặc khi bạn đã rõ cơ chế làm việc của ASP.NET RestAPI, việc học cách xây dựng RestAPI bằng Spring của Java cũng không quá khác biệt.

Mình từng tự học Python mất 1 tuần, và học framework Django mất khoảng 2 tuần nữa. Lý do mình học nhanh vậy là vì:

- Mình đã có kiến thức cơ bản về lập trình (class, data structure ….)
- Mình biết những gì mình cần học. Khi mới lập trình, bạn không biết mình cần học gì. Tuy nhiên nếu đã có kiến thức nói chung về lập trình, bạn sẽ biết mình tập trung học những gì, điều này tiết kiệm rất nhiều thời gian.
- Mình biết là mình làm được. Khi mình hỏi bạn bè chung ngành "Học một ngôn ngữ mới mất bao lâu", hầu hết đều trả lời "một tháng hoặc hơn". Vì thấy tốn nhiều thời gian và khó khăn như vậy nên hầu như họ rất "ngại" học ngôn ngữ mới.

Đừng sợ mình sẽ chọn nhầm ngôn ngữ, cứ học đi. Việc học một ngôn ngữ lập trình mới khi bạn đó có kiến thức cơ sở khá đơn giản, không hề khó khăn và mất thời gian như bạn nghĩ. Thêm vào đó, việc biết nhiều ngôn ngữ sẽ giúp bạn có lợi thế hơn khi xin việc.

Lời khuyên của bản thân Hoàng

Dưới đây là một số lời khuyên của mình, dựa theo kinh nghiệm cá nhân (Mình chỉ có kinh nghiệm mảng web và mobile, nên các lời khuyên có thể sẽ không áp dụng được cho mảng embedded system):

1. Đừng quá tập trung vào công nghệ: Đây là lời khuyên của thầy mình. Một số bạn rất hăm hở tìm hiểu, học những công nghệ mới. Điều này không xấu, nó còn giúp bạn học được nhiều thứ mới, dễ xin việc. Tuy nhiên, công nghệ là thứ dễ thay đổi và mau hết thời [24]. Ví dụ: Sliverlight từng làm mưa làm gió, giờ đang ngắc ngoải. WinForm thì đã lỗi thời, … thời gian bạn bỏ vào học các công nghệ này xem như lãng phí.

Chưa kể, khi một công nghệ được nâng cấp, ví dụ ASP.NET MVC 3 lên MVC 4, ta cần học thêm những điều mới trong bản 4. Việc đầu tư quá nhiều thời gian vào học công nghệ mới sẽ làm bạn dễ đuối, đôi khi là lãng phí thời gian.

[24] Đọc bài "Mặt tối của ngành công nghiệp IT"

2. Hãy đầu tư vào những thứ lâu bền: Nếu không tập trung vào công nghệ, chúng ta nên học gì? Đó là những thứ ít thay đổi, nhưng lại vô cùng cần thiết với developer:

- **Kiến thức cơ bản**: Đừng vội cười, mình biết nhiều bạn tuy đã ra trường nhưng vẫn còn lờ mờ về thuật toán, cấu trúc dữ liệu, các khái niệm pointer, delegate, multi-thread,… Đây là những kiến thức mà ta sẽ sử dụng trong suốt cuộc đời lập trình, và hầu như sẽ chẳng bao giờ thay đổi. Bỏ thời gian ra học kĩ chúng sẽ không hại gì, phải không?
- **Cách viết code**: Đặt tên biến như thế nào, tách hàm ra sao, comment như thế nào. Có một câu danh ngôn: lập trình viên dở viết code cho máy hiểu, lập trình viên giỏi viết code cho cả máy và người hiểu. Có nhiều bạn code xong, sáu tháng sau nhìn lại không hiểu mình viết gì. Khi đi làm, có nhiều giai đoạn bạn phải bảo trì, nâng cấp hệ thống, đọc những đoạn code do "thánh" viết và chửi thề "Thằng này ngu thế". Hãy code có lương tâm, nghĩ tới người sau này sẽ phải đọc, fix bug, sửa code của mình nhé (Biết đâu người đó lại là chính bạn đấy).
- **Design Pattern**: Nắm vững design pattern sẽ giúp bạn ghi điểm khi phỏng vấn. Đùa đấy, nó sẽ giúp bạn giải quyết rất nhiều vấn đề thường gặp trong khi code. Ngoài ra, design pattern sẽ giúp bạn học và hiểu 1 framework mới dễ dàng hơn, các framework thường áp dụng khá nhiều design pattern[25].
- Một số mô hình thường dùng: MVC, MVVM, MVP, mô hình Client-Server … Đây là những câu hỏi thường được hỏi trong các cuộc phỏng vấn, cũng là kiến thức cực kì quan trọng. Hiểu và nắm vững các mô hình trên sẽ làm việc lập trình trở nên thoải mái hơn nhiều.

3. Lời khuyên cuối cùng: Nếu bạn quá dư thời gian, muốn thành thạo một ngôn ngữ nào đó, mình khuyên bạn nên chọn JavaScript (mà ko phải là PHP). Vì các lý do sau đây:

- JavaScript (JS) có ứng dụng rộng khắp, số lượng framework viết cho nó rất nhiều. Ngày nay, JavaScript cũng có thể dùng để viết web, ứng dụng di động. Ngoài ra, đây là ngôn ngữ được rất nhiều lập trình viên sử dụng nên bạn dễ dàng tìm tài liệu để học (Trong khảo sát cuối 2016 của topdev, 90% developer sử dụng JS).
- Học JS không cần cài đặt nhiều. Chỉ cần mở Notepad, lưu lại lại dưới dạng HTML, mở bằng trình duyệt (hầu như máy tính nào cũng có), bạn đã có thể viết những dòng JS đầu tiên. Học C#, Java … sẽ phải cài 1 đống IDE, SDK… rất lâu và mệt mỏi.

[25] Tìm hiểu về Design Pattern trong bài "Nhập môn Design Pattern"

- JS là một ngôn ngữ hay và đẹp. Nó từng là cơn ác mộng của các developer. Ngày xưa mình rất ghét ngôn ngữ này, cộng đồng cũng có rất nhiều người ghét. Tuy nhiên, các cải tiến gần đây đã làm JavaScript có giá hơn, đẹp hơn, được nhiều người săn đón.

Tóm tắt

- Trước khi hỏi "Cần học ngôn ngữ gì?", hãy tự trả lời "Học lập trình để làm gì?"
- Đừng lo chọn sai ngôn ngữ để học, học một ngôn ngữ mới rất đơn giản
- Đừng chạy theo công nghệ, hãy tập trung vào kiến thức cơ bản và những thứ lâu bền
- Nếu quá dư thời gian, hãy tập trung vào học JavaScript

CÁCH TIẾP CẬN 1 NGÔN NGỮ/CÔNG NGHỆ MỚI

Mình đã từng nói về tầm quan trọng của việc cập nhật kiến thức ở bài viết "Những điều đại học không dạy bạn"

> Không như các ngành khác, kiến thức trong ngành IT rất nhanh hết hạn. Với ngành xây dựng, xây một cây cầu cách đây 50 năm cũng chẳng khác gì xây một cây cầu bây giờ. Với ngành y, bệnh cảm cúm cách đây 50 năm triệu chứng cũng giống bệnh cảm cúm bây giờ. Nhưng với ngành IT, công nghệ, ngôn ngữ hoặc framework nổi tiếng cách năm 10-15 năm giờ chẳng ai dùng nữa cả.

Vì vậy mình dành bài viết này để hướng dẫn các bạn cách tiếp cận một công nghệ mới. Đây là những cách mà mình tự nhận ra, tự tổng hợp trên mạng cùng với một số lời khuyên của các bậc đàn anh. Bản thân mình thấy nó khá là hữu dụng, hi vọng chúng cũng sẽ hữu dụng với các bạn.

Phần 1 - Bốn loại kiến thức

Nói về lý thuyết một chút, những kiến thức bạn cần học về một công nghệ có thể chia làm 4 loại sau (Lấy ngôn ngữ C# làm ví dụ):

- Nền tảng (Fundamentals)
- Kiến thức (Information)
- Kĩ năng (Skills)
- Nâng cao (Innovation)

1. Nền tảng (Fundamentals)

Đây là những kiến thức cơ sở nhất, là những viên gạch đặt nền móng cho kiến thức sau này (Ví dụ như: cấu trúc dữ liệu, OOP, vòng lặp, đệ qui, callback, một số mô hình MVC MVVM, cơ chế hoạt động của web, …). Vì chúng là kiến thức nền tảng, mang tính học thuật nhiều nên đôi khi khá là phi thực tế và buồn ngủ. Chắc hẳn ai cũng từng nhức đầu đau não khi nghe các thầy giảng về sự kiện, con trỏ hàm, cây nhị phân, đệ qui…

Tuy nhiên, nếu nắm vững những kiến thức nền tảng này, bạn sẽ thấy việc học và chuyển đổi qua lại giữa các ngôn ngữ khác nhau rất dễ dàng, vì chúng được xây dựng dựa trên nền tảng chung (Như bản thân mình, vì đã rõ cơ chế get/post, giao tiếp giữa client/server, mô hình MVC, mình có thể nhanh chóng học Zend của PHP, Struts2 của Java, ASP.MVC của C#).

Trường đại học chủ yếu dạy những kiến thức này, do đó đôi khi bạn sẽ thấy chương trình học khá khô khan. Hãy nhớ lại những điều mình từng học khi xem phim kiếm hiệp thời xưa: Để học được võ công thượng thừa, phải rành những chiêu thức cơ bản trước. Những chiêu thức hoa mĩ đều từ cơ bản mà ra cả. Ngoài ra, những kiến thức cơ bản này thường "sống lâu" và rất khó "hết hạn" : hàm sort qua 10, 20 năm vẫn giữ nguyên cách sort; cấu trúc dữ liệu stack, binary tree, mô hình MVC qua 10, 20 năm vẫn không hề thay đổi.

2. Kiến thức (Information)

Đây là những kiến thức bậc cao hơn, liên quan tới từng ngôn ngữ/framework chuyên biệt (Ví dụ như LINQ, Event, WinForm, WebForm, …). Những kiến thức này gắn liền với thực tế, có thể áp dụng được ngay vào làm việc. Để học nhanh, áp dụng được những kiến thức này, các bạn phải có fundamental vững. Mình từng gặp khó khăn khi viết Ajax, viết jQuery, function lồng vào nhau v…v. Khi mình hiểu ra chúng gọi là callback, mình học và viết code Ajax, jQuery dễ hơn nhiều.

Một số trường dạy nghề (APTech, Nhất Nghệ …) thường tập trung nhiều vào kiến thức dạng information mà lướt qua kiến thức cơ bản dạng fundamentals. Do đó, tuy học viên được đào tạo ra thường có kiến thức thực tiễn, có thể làm được việc ngay nhưng vấn đề chung mà các bạn này hay gặp là tuy làm được nhưng lại không hiểu cơ chế hoạt động, khi gặp lỗi ko biết nguyên nhân, không biết cách sửa. Lý do là vì kiến thức cơ bản (fundamentals) không đủ.

Một điều cần lưu ý nữa là những kiến thức dạng này khá nhanh "hết hạn", ví dụ như cách routing trong MVC 4 sẽ khác MVC 2, một số hàm trong Entity Framework 6 sẽ khác Entity Framework 4. Do đó nếu không kịp cập nhật, bạn sẽ dễ trở nên lỗi thời, vì kiến thức cũ không sử dụng được nữa.

3. Kĩ năng (Skills)

Đây là loại kiến thức đáng giá nhất (theo nghĩa đen), các công ty sẽ trả lương cho bạn nếu bạn có skills, có thể làm được việc. Kĩ năng có thể học được một phần từ trong sách vở, nhưng phần lớn bạn học được là do quá trình làm việc lâu dài, tiếp xúc nhiều với một công nghệ, giải quyết những tình huống cơ bản và phức tạp.

Ví dụ như: Infomation là việc bạn biết cơ chế routing, binding của ASP MVC. Skill là việc bạn biết áp dụng cơ chế routing, binding để tạo một trang search, insert, update. Skill phức tạp hơn là khi bạn đọc yêu cầu của khách hàng, bạn sẽ mường tượng ra cách viết front end thế nào, back end ra sao, bắt tay vào code ở đâu.

Lương ở các vị trí senior thường cao hơn, lý do là họ đã tiếp xúc với công nghệ nhiều, kĩ năng liên quan tới công nghệ đó cao hơn. Skill có dựa trên infomation, do đó nó cũng khá dễ hết hạn. Nếu bạn đã có nhiều kinh nghiệm ngôn ngữ Cobol, Basic

nhưng thị trường không cần những skill đó nữa, skill của bạn sẽ trở nên vô dụng. Hãy tập trung đầu tư làm mới skill cho mình thường xuyên nhé.

4. Thuần thục (Innovation)

Đây là cảnh giới tối cao của kiến thức, đạt tới cảnh giới này bạn sẽ được gọi là senior, master, hoặc được phong thánh (Điển hình là thánh C# Jon Skeet). Để đạt được cảnh giới này, ngoài quá trình làm việc, tiếp xúc lâu dài với công nghệ, họ còn phải bỏ thời gian đào sâu, mày mò, nghiên cứu công nghệ đó.

Ngoài những kiến thức chung, họ còn biết vô số những thứ chuyên sâu như: Code C# được biên dịch ra như thế nào, quan hệ giữa các component trong .NET, performance của Interface và Abstract class, … Bạn không cần lo lắng quá về cảnh giới này, thay vì biết sâu bạn có thể chọn cách "biết rộng", mở rộng kiến thức lên nhiều lĩnh vực chứ không quá chuyên sau vào một mảng.

Phần 2 – Ba giai đoạn tiếp cận công nghệ

Để có thể học tập dễ dàng hơn, mong các bạn hãy giữ 3 tư tưởng sau:

- Học một ngôn ngữ/công nghệ mới không khó: Mình biết có nhiều bạn rất ngại, rất sợ học cái mới, hễ nghe nói cái gì là lạ là lắc đầu nguầy nguậy, bảo "không biết". Chúng ta nên có tư tưởng là "không phải không biết mà là… chưa biết, chịu khó tìm hiểu một tí là biết thôi".
- Để học được nhiều cái mới, bạn cần phải giỏi tiếng Anh, không ngại đọc (Không cần giỏi cả 4 kĩ năng, chỉ cần giỏi reading là được). Nếu chỉ chăm chăm tìm tài liệu tiếng Việt, chỉ biết há miệng chờ hàng người ta dịch sẵn, bạn sẽ đi sau thời đại[26]. Ngoài ra, với vốn tiếng Anh kha khá, khi có bug hoặc gặp vấn đề khó giải quyết, bạn sẽ dễ Google để tìm câu trả lời hơn.
- Hạn chế hỏi linh tinh, hãy Google trước khi hỏi . Mình rất đồng tình với quan điểm "không biết phải hỏi, không giấu dốt". Tuy nhiên, dân lập trình viên nói chung rất ghét những câu hỏi ngu và thiếu suy nghĩ. Trước khi hỏi, hãy thử

[26] Xem lại bài "Được gì mất gì khi học lập trình bằng tiếng Việt"

tìm Google trước[27]. Có khi bạn hỏi chỉ mất một phút là có câu trả lời, Google để tìm câu trả lời mất tới một tiếng. Nhưng trong một tiếng đó, bạn sẽ học được rất nhiều điều liên quan khác, cả những điều bạn không biết mình cần phải hỏi.

Các bạn nhớ đọc lại phần trước để hiểu khái niệm về: fundamental, information và skill nhé. Dưới đây, mình sẽ chia sẻ quy trình mình áp dụng để học một công nghệ mới. Tất nhiên bạn có thể rút gọn qui trình tùy vào kiến thức sẵn có của bạn đối với công nghệ đó.

Tìm tài liệu học – Giai đoạn sơ khởi

Đây là bước quan trọng nhất. Nếu có người quen rành công nghệ này, bạn có thể nhờ họ chỉ từ khóa, tên sách, trang web v…v để mình có thể tự tìm hiểu. Họ cũng sẽ chỉ cho những gì cần học. Ví dụ mình muốn học thiết kế web, cần học trước về HTML, CSS, JavaScript, jQuery.

Trường hợp xui xẻo hơn, nếu không có người quen, có bạn thể vào amazon.com, đánh tên công nghệ mình muốn học, sau đó chọn khoảng hai cuốn ebook đứng đầu, sau đó tìm bản ebook và bắt đầu học. Lưu ý là chỉ cần tối đa hai cuốn, không nên tìm nhiều hơn nhé, nhiều hơn bạn sẽ lười và không đọc đâu. Mình biết có nhiều bạn tải gần cả trăm MB ebook, tưởng mình giỏi, nhiều tài liệu mà bản thân họ chẳng đọc bao giờ.

Bắt đầu học – Giai đoạn nhập môn

Tại sao mình lại khuyến khích bạn học từ sách mà không phải là học qua video, website, forum… hay gì đó. Lý do là khi xuất bản một cuốn sách, tác giả thường trau chuốt và soạn sẵn một chương trình học cho bạn. Các kiến thức được trình bày trong sách theo một cách tuần tự mạch lạc. Đây là giai đoạn các bạn bổ sung kiến thức dạng nền tảng (fundamentals) và infomation về công nghệ mình muốn học.

Sách kĩ thuật thì dĩ nhiên phải có code. Các bạn nên làm theo hướng dẫn trong sách, setup môi trường, phải code theo (Đừng đọc code rồi gật gù ờ ờ nhé, chẳng nhớ được gì đâu). Lưu ý là gõ code bằng tay để hiểu, không copy code vào rồi chạy nhé. Vừa gõ, vừa sửa code, vừa thử nghiệm, các bạn sẽ dần dần có một cái nhìn rõ ràng hơn về công nghệ mình đang học. Mình đã tự học HTML, CSS, Javascript, jQuery theo cách này, thông qua series sách Head First.

[27] Mình từng đề cập đến chuyện này trong bài "Thức trạng học lập trình của các sinh viên hiện này"

Một số series hay để nhập môn về công nghệ: xxx For Dummies, Head First xxx, một sách khác của O'Reilly. Những sách này có ngôn ngữ đơn giản, ảnh minh họa nhiều, làm việc học trở nên vui vẻ và nhẹ nhàng.

Áp dụng kiến thức – Giai đoạn nâng cao

Bạn đừng lo nếu mình đọc được một nửa cuốn sách đã chán, mình cũng từng như vậy. Nếu đọc được một nửa cuốn sách thì bạn đã có kha khá kiến thức nền tảng và một chút kiến thức chuyên sâu về công nghệ mình học. Lúc này bạn hãy tạm xa rời sách vở và thử áp dụng kiến thức mình đã học vào thực tế.

Bằng cách nào? Hãy thử viết một phần mềm nho nhỏ, hoặc một trang web to-do-list, hoặc làm một blog bằng công nghệ mình đang học. Đây là những phần mềm dễ viết, yêu cầu rõ ràng, cho bạn cơ hội để tự trải nghiệm công nghệ mình học qua việc làm các chức năng: Chức năng hiển thị, thêm bớt xóa sửa, kết nối database ,....

Nếu gặp khó khăn trong quá trình làm, bạn có thể xem lại sách, tìm cách giải quyết tương tự, hoặc lên stackoverflow để hỏi. Sau khi làm xong, hãy ráng trau chuốt source code, sau đó upload nó lên Github. Bạn vừa có code để tham khảo, hướng dẫn cho người sau, vừa có dự án để giới thiệu cho nhà tuyển dụng.

Ở giai đoạn áp dụng này, bạn sẽ học được nhiều skill, qua đó củng cố thêm fundamental và infomation. Bạn tốt của bạn ở giai đoạn này là stackoverflow hoặc 1 số sách dạng cookbook. Những sách cookbook này khá hay, chúng hướng dẫn cách dùng công nghệ để giải quyết một số yêu cầu thường gặp khi code (Cách parse 1 string sang DateTime trong C#, cách validate 1 form trong jQuery, …)

Trên đây là 3 giai đoạn các bạn sẽ trải qua trong quá trình tiếp cận công nghệ. Khi cảm thấy mình có thể giải quyết 80% mọi vấn đề gặp phải, các bạn đã đạt tới cuối giai đoạn "áp dụng". Các bạn có thể bỏ thời gian nghiên cứu chuyên sâu hơn, hoặc chuyển qua tìm hiểu công nghệ mới, tùy vào đam mê của mỗi người.

Dĩ nhiên mỗi người có một cách học khác nhau, hãy thử nghiệm để tìm cách học phù hợp với bản thân mình. Nếu chán việc đọc sách, bạn có thể tìm học qua video trên pluralsight[28] hoặc udemy. Nếu cảm thấy không nhớ kiến thức, bạn có thể lên codeschool vừa học vừa code, cũng khá trực quan và dễ hiểu.

Tóm tắt:

- Muốn nắm được một công nghệ, bạn phải nắm được nền tảng, kiến thức và rèn luyện kĩ năng.

[28] Xem bài viết sau "Top những trường dạy code online cho developer"

- Để học tốt, hãy chuẩn bị kĩ tài liệu học. Khi đọc sách hoặc xem video, nhớ code theo chứ đừng đọc suông.
- Sau khi tiếp thu kiến thức, hãy áp dụng kiến thức để xây dựng một cái gì đó. Điều này sẽ nâng cao kĩ năng và giúp bạn nhớ lâu hơn.

TOP CÁC "TRƯỜNG DẠY CODE" ONLINE CHO CÁC DEVELOPER

Là một developer, việc học một ngôn ngữ hay công nghệ mới là "chuyện thường ở huyện". Ở bài viết trước, mình đã từng chia sẻ một số hướng tiếp cận ngôn ngữ, công nghệ. Bài viết này sẽ giới thiệu một số "trường dạy code" online.

Các trường này cung cấp bài giảng online dưới dạng video (có hoặc không có phụ đề), cho phép ta code trực tiếp trên trình duyệt. Với các trường online này, bạn có thể dễ dàng tự học ở nhà hoặc ở công ty. Bảng xếp hạng này dựa theo độ nổi tiếng của chúng trên Google, cũng như đánh giá của mình khi sử dụng.

Mình sẽ giới thiệu các trường tiếng Anh lẫn Việt, nhưng ưu tiên các trường tiếng Anh hơn, vì số lượng cũng như chất lượng các khóa học ở những trường này thường cao hơn trường tiếng Việt,

Code Academy (https://www.codecademy.com/)

T

4

Đây là một trong các trường nổi tiếng nhất về lập trình Web. Toàn bộ kiến thức Code Acedemy dạy đều được cung cấp một cách miễn phí. Bạn có thể học từ HTML, CSS, JS, jQuery,... cho đến các công nghệ mới như AngularJS, React.

Giao diện của trang khá trực quan, đơn giản. Code Academy sẽ ra một số task cho người học code theo. Sau khi hoàn thành, chương trình sẽ chấm điểm và chuyển bạn tới bài sau. Theo ý kiến cá nhân mình, trang này rất phù hợp cho các bạn mới nhập môn lập trình. Khuyết điểm của Code Acedemy là số lượng các công nghệ/ngôn ngữ mà nó dạy vẫn còn hơi ít.

Pluralsight (https://www.pluralsight.com/)

Trong cộng đồng developer chuyên nghiệp, Pluralsight lại nổi tiếng hơn codeacademy nhiều. Bạn có thể học được hầu như mọi ngôn ngữ, mọi framework (C#, Java, PHP, MVC, AngularJS, …) trên Pluralsight. Tác giả các khóa học đều là những lập trình viên chuyên nghiệp và danh tiếng (MVP), chất lượng khóa học cũng rất cao. Đây là trang web ưa thích của mình.

Pluralsight có thu phí sử dụng. Giá dao động khoảng 30-50USD/tháng. Các bạn sinh viên có thể dùng tài khoản trường để được free 2 tháng. Dân đi làm như mình có thể xin ké tài khoản của công ty để vào học. Mình học được vô số thứ hữu ích từ trang này: AngularJS, định hướng sự nghiệp, Dependency Injection,….

Khuyết điểm của Pluralsight là ta chỉ học qua video mà không thực hành. Học phí cũng khá cao so với các bạn sinh viên.

Code School (https://www.codeschool.com)

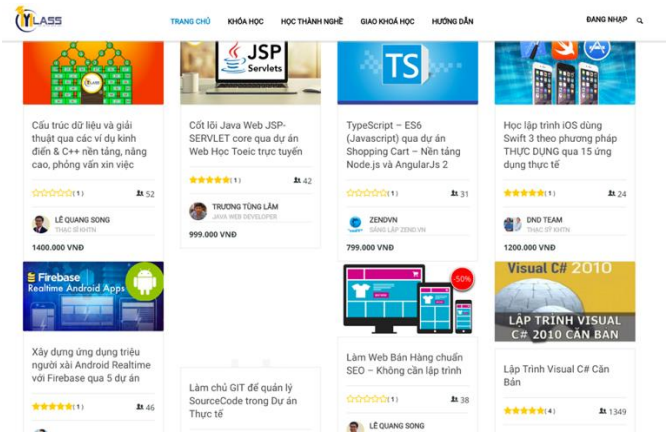
Một website khá mới mẻ và đang gây được sự chú ý trong thời gian gần đây khi vừa được Pluralsight mua lại. Điểm hay ho là nó kết hợp bài giảng video, slide PDF và hệ thống vừa code vừa chấm điểm của CodeAcademy.

Codeschool dạy khá nhiều ngôn ngữ như Ruby, Python, AngularJS, Javascript. Một số khóa học cơ bản được miễn phí, nhưng phần lớn là thu phí. Các clip dạy học trên CodeSchool cũng rất vui nhộn.

Udacity (https://www.udacity.com/)

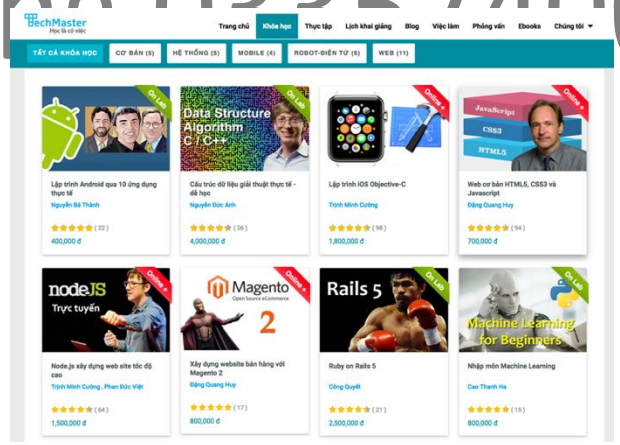
Trang web này khá nổi tiếng trong cộng đồng lập trình viên nước ngoài. Số lượng khóa học không nhiều, nhưng được tập trung thành các pathway, bổ trợ kiến thức cho nhau. Udacity còn có hệ thống nanodegree, tương đương với những chứng chỉ mini dành cho các lập trình viên.

MyClass (http://it.myclass.vn/)



Đây là một trong các hệ thống dạy lập trình nổi tiếng Việt Nam, với khoảng vài chục khóa học. Các khóa học đều được giảng dạy bằng tiếng Việt đi kèm code demo nên các bạn tiếng Anh không khá cũng có thể học được. Điểm thú vị ở itclass là có một số khóa học free toàn bộ hoặc một phần để các bạn học thử. Học phí một khóa học dao động từ 200-800k.

Techmaster (https://techmaster.vn/)



Techmaster cũng là một trang dạy lập trình khá nổi tiếng ở Việt Nam. Ngoài các khóa học online, họ còn cung cấp thêm một số khóa học offline (giảng viên trực tiếp giảng dạy), hỗ trợ thực tập cho sinh viên vừa học. Giá mỗi khóa học từ 500k-2 triệu, với các khóa có giảng viên sẽ đắt hơn.

Nhược điểm của ITClass và Techmaster là số lượng khóa học khá ít, chỉ vào khoảng 3-40 khóa, không phong phú bằng các trang nước ngoài. Do tính giá theo khóa nên bạn không thể học thoải mái như Pluralsight.

Ngoài ra, một số trang web dạy học trực tuyến cũng có vài khóa học về lập trình (edx, udemy ở nước ngoài; edumall, kyna ở Việt Nam). Các bạn có thể dễ dàng tìm và học.

Tóm tắt

- Hiện nay, có khá nhiều trường dạy lập trình online nên bạn có thể tự học tại nhà hoặc công ty
- Nên thử học các khóa học free trước để đỡ tốn phí. Trước khi học, hãy xem tóm tắt khóa học để đánh giá
- Nếu không thích xem video, hãy vào codeschool hoặc codeacademy để vừa học vừa code

KĨ NĂNG CẦN CÓ CỦA MỘT WEB DEVELOPER

Hiện nay, một lập trình viên có thể lựa chọn cho mình nhiều hướng phát triển: Lập trình nhúng (Embeded System), lập trình web, lập trình mobile (ứng dụng di động),... Trong ba hướng này, lập trình web đang là hướng mà các công ty tuyển dụng nhiều nhất. Do vậy, mình chia sẻ một số kĩ năng mà các bạn cần chuẩn bị nếu muốn theo con đường làm Web Developer.

Kĩ năng Front-end

Nói một các đơn giản, front-end là những gì người dùng nhìn thấy và tương tác. Nó là "mặt tiền" của một trang web. Nếu bạn thích thiết kế, gần gũi với người dùng thì bạn có thể tập trung phát triển những kĩ năng front-end và trở thành một front-end developer. Những kĩ năng bạn cần có bao gồm:

- HTML/CSS/Javascript cơ bản (Đừng nghĩ js dễ nhé, khó lắm đấy).
- Một số thư viện/framework nổi tiếng: Bootstrap, jQuery, AngularJS, EmberJS.
- Kĩ năng thiết kế, sử dụng Photoshop. Kiến thức và kinh nghiệm về UI/UX [29].
- LESS, SASS (stylesheet language).
- Sử dụng NodeJS, npm, grunt, ... để optimize, minimize HTML/CSS/JS.
- Kiến thức về Ajax, cách thiết kế giao diện responsive .....

Vai trò của front-end trong một sản phẩm là khá quan trọng, vì giao diện là thứ đập vào mắt người dùng đầu tiên. Front-end developer không chỉ thiết kế giao diện đẹp, mà còn phải rõ ràng và dễ sử dụng, giúp người dùng làm việc mình muốn một cách đơn giản, nhanh gọn (Google là một ví dụ).

Một số sách hay để nâng cao kĩ năng front-end:

- Series Head First, The Missing Manual (Head First HTML & CSS, jQuery The Missing Manual ...)
- Don't Make Me Think
- The Design of Everyday Things

Kĩ năng Back-end

Back-end là những thứ người dùng không nhìn thấy nhưng giúp cho hệ thống hoạt động trơn tru. Dữ liệu của người dùng, những thuật toán phân tích ... đều nằm ở

[29] Xem bài viết "Tổng quan về UI/UX trong ngành lập trình"

back-end. Nếu front-end là lớp sơn, lớp vỏ của một ngôi nhà thì back-end chính là giàn giáo, xương sườn của ngôi nhà đó. Những kĩ năng bạn cần có bao gồm:

- Ngôn ngữ server-side để viết back-end: C#, Java, Python, PHP. Dĩ nhiên là phải bao gồm kiến thức về những web framework đi kèm các ngôn ngữ này như ASP.NET MVC, Spring, Django, Rails,…
- Kiến thức về database SQL: MS SQL Server, MySQL, … Gần đây một số database NoSQL đang khá thịnh hành: Neo4j, MongoDB, …
- Kiến thức về web nói chung, mô hình client/server, cách viết Web Service, cách đăng nhập và phân quyền.
- Kiến thức về 1 số CMS: WordPress, Joomla, Umbraco, ….

Kiến thức phần back-end rất nhiều và phức tạp, do đó một back-end developer chỉ nên tập trung vào 2-3 ngôn ngữ chính, đừng ráng học quá nhiều kẻo "tấu hỏa nhập ma". Code phần back-end thường rất nhiều và "khủng", do đó cần có cấu trúc tốt để có thể dễ cải tiến và mở rộng. Back-end developer có thể trau dồi kiến thức để leo lên vị trí System Analyst hoặc Software Architecture.

Một số sách hay cho back-end developer:

- Clean Code
- Code Complete
- Head First Design Pattern
- Sách chuyên sâu về ngôn ngữ/framework: C# in Depth, Pro .NET 4.5, Spring in Action, …

Kĩ năng phân tích thiết kế

Hiện nay, ranh giới giữa front-end và back-end trong lập trình web khá mong manh. Đa phần các web developer ngày nay phát triển theo hướng full-stack: thông thạo về back-end và có kha khá kiến thức về front-end. Biết cả front-end và back-end, bạn sẽ biết được một trang web hoạt động như thế nào từ đầu tới cuối.

Lập trình viên web cũng có thể "lấn sân" qua mảng mobile nhờ sự giúp sức của một số framework như React-Native, Cordova, Ionic, Xamarin, … Để tăng giá trị của bản thân, ngoài kĩ năng cứng, bạn cần trau dồi kĩ năng phân tích, giải quyết vấn đề: Khách hàng cần gì ở trang web, lượng truy cập là bao nhiêu, làm sao để tăng tốc độ tải của Web? Nhà tuyển dụng sẽ đánh giá kĩ năng này của bạn khi phỏng vấn đấy.

Một số sách nên tham khảo:

- The Pragmatic Programmer: From Journeyman to Master
- The Passionate Programmer: Creating a Remarkable Career in Software Development
- Getting Real

- Cracking the Coding Interview: 150 Programming Questions and Solutions

Tóm tắt

- Để trở thành lập trình viên web, bạn cần có kĩ năng front-end, back-end và phân tích thiết kế
- Front-end là những gì người dùng nhìn thấy. Bạn cần biết về UI/UX và HTML, CSS, JS
- Back-end là những thứ phía sau giúp hệ thống hoạt động trơn tru. Bạn cần biết về database và một ngôn ngữ như: C#, Java,, PHP…
- Có thể tập trung vào một mảng hoặc phát triển theo hướng full-stack, thông thạo cả front-end và back-end

TỔNG QUAN VỀ LẬP TRÌNH ỨNG DỤNG DI ĐỘNG

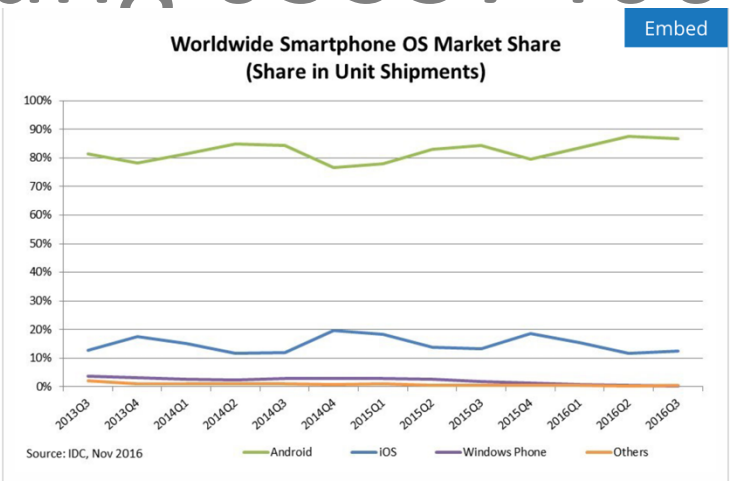
Trong khoảng thời gian gần đây, lập trình di động cũng đang trở thành một ngành hot. Các mẩu tin tuyển dụng gần nhất mình đọc thường tuyển Android developer và iOS developer với mức lương khá cao, không thua kém gì lập trình web hay lập trình hệ thống nhúng.

Ngoài ra, nếu biết cách lập trình ứng dụng di động, bạn cũng có thể làm freelance hoặc tự phát triển ứng dụng và kiếm tiền thông qua ứng dụng của mình. (Tương tự như Flappy Bird của Nguyễn Hà Đông).

Bài viết này sẽ giúp các bạn có cái nhìn tổng quan về thị trường ứng dụng di động hiện nay, cũng như giới thiệu một số ngôn ngữ/công nghệ các bạn cần biết nếu muốn đi theo con đường này.

Phần 1 - Thế chân vạc trong trận chiến Mobile

Trên thị trường ứng dụng di động hiện nay, 3 hệ điều hành chiếm thị phần cao nhất là : Android, iOS và Window Phone, tiếp sau là một số hệ điều hành khác như BlackBerry... Trong phạm vi bài viết, mình chỉ phân tích về 3 hệ điều hành đứng đầu là Android, iOS và Windows Phone nhé.



Thị phần các hệ điều hành năm 2016 (Nguồn: http://www.idc.com/promo/smartphone-market-share/os)

Android – Kẻ chiếm miếng bánh lớn nhất

Theo biểu đồ, ta dễ nhận ra Android luôn chiếm hơn 80% thị phần của mảng di động. Ứng dụng Android được viết bằng ngôn ngữ Java, do đó các bạn lập trình viên Java có thể dễ dàng chuyển hướng qua mảng này.

Lập trình viên Android đang là mục tiêu được các công ty săn đón. Các mẩu tin tuyển dụng Android developer chiếm tỉ trọng lớn nhất trong số các tin tuyển dụng của mảng mobile. Thuở còn làm đồ án tốt nghiệp, có một anh trong nhóm mình chưa biết gì về Android. Mình và ổng tự học và làm trong 2 tháng làm đồ án, vừa xong đồ án thì anh ấy đi phỏng vấn vị trí lập trình Android trong lập công ty và được nhận luôn.

Android có khá nhiều device với đủ kích cỡ màn hình cùng với vô số phiên bản được cập nhật thường xuyên. Điều này gây khá nhiều khó khăn cho lập trình viên khi viết app : Cần phải test nhiều, đảm bảo ứng dụng tương thích với nhiều device, không bị lỗi giao diện, v…v.

Nếu bạn muốn đi theo con đường viết ứng dụng kiếm tiền, đưa ứng dụng lên Google Store, bạn sẽ phải mua một tài khoản Android Developer. Phí tài khoản này là 25$/năm.

iOS – Vị vua không ngai

Theo biểu đồ, iOS chỉ chiếm 20% thị phần, chỉ bằng 1/4 so với Android. Tuy nhiên, các khảo sát lại cho thấy doanh thu của ứng dụng trên Apple Store lại cao hơn Google Play Store. Nguyên nhân là do người dùng iOS chơi sang hơn, chịu khó bỏ tiền mua ứng dụng hơn so với người dùng Android.

Số lượng tuyển dụng iOS ít hơn Android, tuy nhiên lương cho lập trình viên iOS lại nhỉnh hơn bên Android chút đỉnh. Lý do không phải vì iOS tốt hơn Android, mà chỉ đơn thuần là qui luật cung cầu: Lập trình viên iOS hiếm hơn lập trình viên Android nên họ có giá cao hơn.

Để tiếp cận iOS, bạn cần máy ảo hoặc máy Mac để cài hệ điều hành MacOS. Ứng dụng iOS được viết bằng ngôn ngữ Objective-C (Giống C nhưng có thêm OOP) hoặc Swift. Việc code và debug trên iOS cũng không phức tạp mấy. Bạn phải cài đặt Xcode, và tạo tài khoản Apple Developer mới có thể test ứng dụng và đưa ứng dụng lên Apple Store. Bộ phận kiểm duyệt của Apple Store cũng khắt khe hơn Google Play Store, nhiều khi bạn phải chờ khá lâu để ứng dụng của mình được duyệt.

Nếu làm ở công ty, bạn sẽ được cũng cấp tài khoản Apple Developer cũng như device để test. Nếu muốn tự viết, bạn sẽ phải tự trả 100$/năm cho tài khoản Apple Developer, và mất thêm một khoản kha khá để mua thiết bị (iPhone, iPad) về test.

Windows Phone – Kẻ sinh sau đẻ muộn

Windows Phone đã chậm chân khi gia nhập thị trường di động, nơi Android và iOS đã làm mưa làm gió khá lâu. Mặc dù Microsoft đã có một số chính sách hỗ trợ

developer nhưng hệ thống ứng dụng trên Window App Store vẫn còn khá nghèo nàn và nhàm chán.

Thú thật, mình chả thấy công ty nào tuyển lập trình viên Windows Phone cả. Hầu như các công ty đều o bế cho ứng dụng trên Android, iOS trước rồi mới đến Windows Phone. Vì Windows Phone được viết bằng ngôn ngữ C# kết hợp với XAML, các lập trình viên C# có thể thử sức ở mảng này.

Cá nhân mình từng code cả Android lẫn Windows Phone thì thấy Windows Phone dễ code hơn, debug nhanh và tiện hơn. Với Android, nếu không có device, ta phải debug trên máy ảo, chạy rất chậm nếu máy bạn yếu, máy ảo của Window Phone lại rất mượt và nhanh.

Nếu muốn viết app kiếm tiền, mình nghỉ các bạn đừng vội thử Window Phone vì hiện tại số lượng người dùng Window Phone rất ít. Tuy vậy, nó cũng có một số ưu điểm:

- Apple Store và Play Store đã có rất nhiều ứng dụng, tính cạnh tranh rất cao. Ngược lại, bạn ít khi gặp phải sự cạnh tranh trên Window Store.
- Microsoft đưa ra khá nhiều chính sách hỗ trợ Windows Phone, có thể trong tương lai sẽ thu hút nhiều người dùng hơn.
- Account Window Phone Developer có giá rất rẻ, chỉ có 19$ và dùng mãi mãi.

Phần 2 – Các hướng phát triển ứng dụng di động

Hiện nay, có 3 hướng chính để phát triển một ứng dụng di động, đó là: Web App, Native App và Hybrid App. Mỗi hướng có những ưu nhược điểm riêng và cần những kĩ năng riêng biệt

Web App

Hướng Mobile Web thường được áp dụng khi các bạn đã có sẵn một website đang hoạt động. Ta sẽ tạo thêm một trang web riêng cho mobile, sử dụng HTML, CSS, một số framework hỗ trợ mobile và responsive (Bootstrap, jQuery Mobile, Materialize). Người dùng sẽ truy cập trang web dành cho mobile để sử dụng ứng dụng.

Các xử lý khác liên quan đến backend như database sẽ được thực hiện phía trên server. Với một số công nghệ như Ionic, một web app có thể trông y hệt một ứng dụng di động thật sự.

Ưu điểm

- Chỉ cần có kiến thức về web là viết được
- Viết một lần, chạy được trên mọi hệ điều hành
- Người dùng không cần phải cài app, có thể vào thẳng trang web
- Không cần phải thông qua App Store, tiết kiệm tiền
- Dễ nâng cấp (Chỉ việc nâng cấp web là xong)

Nhược điểm

- Với một số máy đời cũ, Web App sẽ bị lỗi giao diện, hiển thị sai, hoặc không chạy được vì lỗi JavaScript
- Performance chậm
- Không thể tận dụng được các tính năng của di động: Push notification, chụp hình, nghiêng máy, định vị GPS…

Kĩ năng cần có

- Kiến thức HTML, CSS, Javascript cơ bản
- Kiến thức về một số framework responsive/mobile như: jQuery Mobile, Bootstrap, …
- Một số framework JavaScript để viết Single Page Application: AngularJS, EmberJS, …

Native App

Viết Native App nghĩa là lập trình viên sẽ sử dụng SDK mà nhà sản xuất cung cấp (Android thì dùng Android SDK, iOS thì dùng Swift và xCode) để lập trình ra một ứng

dụng, build ứng dụng đó thành file cài đặt và gửi lên App Store để kiểm duyệt. Người dùng sẽ phải tìm ứng dụng trên App Store, tải về máy và chạy.

Đây là hướng phát triển được áp dụng nhiều nhất, điển hình là game Flappy Bird của Nguyễn Hà Đông. Với những ứng dụng game, xử lý ảnh, cần tính toán nhiều, Native App là lựa chọn tốt nhất.

Với những hệ thống lớn, cần đồng bộ dữ liệu, ta vẫn phải viết phần back-end trên server. Server sẽ đưa ra một số API. Native app lấy dữ liệu về máy, truyền dữ liệu lên server thông qua các API này.

Ưu điểm

- Tận dụng được toàn bộ những tính năng của device: Chụp ảnh, nghiêng máy, rung, GPS, notification
- Có thể chạy được offline
- Performance nhanh
- Là lựa chọn duy nhất cho các ứng dụng game, xử lý hình ảnh hay video …

Khuyết điểm

- Cần cài đặt khá nhiều thứ để code (Eclipse, XCode, Android SDK, …)
- Với mỗi hệ điều hành, ta phải viết một ứng dụng riêng. Khó đảm bảo sự đồng bộ giữa các ứng dụng (1 button trên Android sẽ khác 1 button trên iOS)
- Cần phải đưa app lên App Store, mỗi lần nâng cấp thì người dùng phải update lại app

Kĩ năng cần có

- Ngôn ngữ lập trình: Java cho Android, Objective-C hoặc Swift cho iOS, C# cho Windows Phone
- Kiến thức chuyên sâu về SDK: View, Action, Adapter trong Android …
- Cách xây dựng Web Serivce, Restful API, cách gọi API từ device, …

Hybrid App

Hybrid App kết hợp những ưu điểm của Mobile Web và Native App. Ta xây dựng một ứng dụng bằng HTML, CSS, Javascript, chạy trên WebView của mobile. Tuy nhiên, Hybrid App vẫn có thể tận dụng những tính năng của device như chụp hình, GPS, rung, ….

Hybrid App sẽ được viết dựa trên một cross-platform framework: Cordova, Phonegap,…. Ta sẽ gọi những chức năng của mobile thông qua API mà framework này cung cấp, dưới dạng JavaScript. Bạn chỉ cần viết một lần, những framework này

sẽ tự động build ứng dụng này ra các file cài đặt cho Android, iOS và Windows Phone. Một số ứng dụng không quá nặng về xử lý, cần tận dụng chức năng của device sẽ chọn hướng phát triển này.

Ưu điểm

- Chỉ cần biết HTML, CSS, JS (Thế nên mình mới khuyên các bạn nên học Javascript)
- Viết một lần, chạy được trên nhiều hệ điều hành
- Tận dụng được các chức năng của device

Khuyết điểm

- Không ổn định, khó debug. Framework sẽ dịch code của bạn thành code native, việc sửa lỗi ứng dụng khá khó vì bạn không biết code sẽ được dịch ra như thế nào
- Performance chậm
- Cần cài đặt nhiều thứ (Titanium, Cordova đều bắt phải cài đặt SDK này nọ thì mới build ứng dụng được)

Kiến thức cần biết

- HTML, CSS, Javscript cơ bản
- Cách dùng một số framework CSS, Javascript: jQuery Mobile, Ionic Framework, AngularJS, Bootstrap, ...
- Kiến thức về các cross-platform framework: Cordova, Phonegap
- Cách xây dựng Web Serivce, Restful API, cách gọi API từ device, ... (Hybrid app cũng sẽ kết nối với server thông qua API như Native App)

Tóm tắt

- Trong thị trường lập trình di động hiện nay có 3 hệ điều hành chính là: Android, iOS và Window Phone.
- Android chiếm thị phần lớn nhất, nhưng doanh thu cao nhất thuộc về iOS
- Có 3 hướng chính để phát triển ứng dụng di động: Web App, Native App, Hybrid App

MUÔN NẺO ĐƯỜNG TÌM VIỆC

Không như các ngành nghề khác, ngành lập trình là một ngành khá dễ xin việc ở Việt Nam. Chỉ cần có khả năng code kha khá, các bạn sinh viên có thể dễ dàng xin việc với mức lương tạm ổn, không cần phải quen biết, lót tay hay con ông cháu cha gì cả.

Bài viết này chia sẻ kinh nghiệm mình có được trong quá trình xin việc, từ lúc viết CV cho đến lúc phỏng vấn. Hi vọng chúng sẽ có ích cho bạn trên bước đường tìm việc.

Phần 1: Tìm việc và viết CV

Tìm việc ở đâu?

Các trang web dưới đây là những trang web về việc làm lớn nhất Việt Nam hiện tại. Đây là những kênh trung gian giữa các nhà tuyển dụng và người tìm việc, do đó mình khuyên các bạn đang tìm việc hãy tạo một CV online cho mỗi trang này:

- IT Việc: https://itviec.com/
- Linkedin: https://www.linkedin.com/
- Vietnamworks: http://topit.vietnamworks.com/
- Jobtown: https://jobtown.co/
- Một số group lập trình trên facebook cũng thường có các mẩu tin tuyển dụng.

Lâu lâu vẫn có nhận được điện thoại mời phỏng vấn của nhân sự các công ty. Mình không cần nộp CV hay ứng tuyển vì họ đã xem CV trên mạng rồi. Cá nhân mình khuyến khích các bạn nên trau chuốt CV trên LinkedIn, chức năng connect của trang này cho phép bạn kết nối với nhiều người (đồng nghiệp, cấp trên), mở rộng mạng lưới nghề nghiệp của bạn.

Viết và gửi CV

1. Nội dung

Do đã có vô số các bài viết hướng dẫn viết CV trên mạng rồi, mình sẽ không nhắc lại. Nếu bạn chưa biết trình bày CV như thế nào, cần những thông tin gì, hãy tìm một CV mẫu cho vị trí của mình (vd bạn là junior developer hãy xem CV mẫu của 1 junor dev, tương tự với junior QC), chỉnh sửa lại cho phù hợp. Format CV của linkedin cũng khá ổn; chỉ cần bê các phần từ trên linkedin về, chỉnh sửa một chút là bạn đã có một CV khá chuẩn.

Một điều cần lưu ý: Hãy chỉnh sửa CV cho phù hợp với vị trí bạn ứng tuyển. Giả sử bạn biết cả C#, Java, Python, nhưng công việc bạn ứng tuyển đòi hỏi C# MVC, hãy để các kĩ năng và dự án liên quan tới C# lên đầu trong CV.

2. Cách trình bày

CV không cần phải đẹp lộng lẫy nhưng phải rõ ràng và ngắn gọn. Cỡ chữ nên phù hợp là 12-14, sử dụng font cơ bản như Times New Roman hoặc Arial. Các phần đề mục nên viết rõ ràng, in đậm, nhìn lướt qua có thể đọc được. Thường thường bên tuyển dụng cũng không quá bắt bẻ về mặt hình thức, nhưng các bạn nên chịu khó canh thẳng hàng thẳng lối, đồng thời soát kĩ các lỗi chính tả. Những điều nhỏ nhặt này sẽ thể hiện tính chuyên nghiệp và sự cẩn thận của bạn.

Các bạn cũng nên xuất file CV ra dưới dạng PDF để máy nào cũng mở được. CV nên đặt tên theo format "**[Tên vị trí] Tên bạn**", giúp nhà tuyển dụng dễ đọc và phân loại (Hoặc bạn đặt tên theo format mà nhà tuyển dụng đưa ra trong quảng cáo tuyển dụng).

3. Một số sai lầm hay gặp

- **Ảo tưởng sức mạnh**: Không biết vô tình hay cố ý mà trong phần kĩ năng, nhiều bạn tự đánh giá trình độ của mình là Intermediate, Expert, hoặc Master, dù chỉ mới ra trường, chỉ mới làm 1,2 cái đồ án chứ chưa làm dự án lớn nào. Nếu đã từng làm qua một công nghệ nào, các bạn nên ghi thời gian đã tiếp xúc và mức độ nắm vững công nghệ đó là đủ. Trình độ bạn ở mức nào thì người phỏng vấn sẽ tự xác định, bạn có nói mình là Master họ cũng không tin đâu.
- **Gây chú ý không đúng cách**: Dùng font màu mè lạ mắt để đập vào mắt nhà tuyển dụng. Cách này thường gây tác dụng ngược. Họ có thể quăng CV của bạn vào sọt rác không thương tiếc.
- **Chém gió**: Mình từng gặp trường hợp có bạn chém gió về số năm đi làm và công nghệ mình biết trong CV. Tới lúc phỏng vấn, bị vặn hỏi thì bạn bảo là: mình làm part time, công nghệ ABC gì đó mình hiểu nhưng chưa làm bao giờ … Nhiêu đó là quá đủ để rớt đài rồi nhỉ? Hãy nhớ rằng mục đích của CV chỉ là giúp bạn qua được vòng gửi xe, được mời phỏng vấn thôi, chứ không giúp bạn có được việc làm ngay đâu nhé.

Phần 2: Quy trình phỏng vấn

Sau khi đọc xong phần 1, hi vọng các bạn đã chuẩn bị được cho mình một mẫu CV rõ ràng mạch lạc. Nếu mọi chuyện đều ổn, khoảng vài ngày đến 2 tuần sau khi gửi CV, bạn sẽ được một/nhiều công ty gọi điện thoại hoặc gửi email mời phỏng vấn. Sau khi

nhận điện thoại, hãy kiểm tra hộp mail, sau đó gửi mail xác nhận rằng mình sẽ có mặt tại công ty lúc X giờ, ngày Y để thực hiện phỏng vấn nhé.

Một số công ty còn có thêm vòng interview qua điện thoại. Một số công ty lớn (Fsoft, Harvey Nash, …) có cả entry test – bài thi đầu vào dành cho ứng viên, bao gồm: Thi tiếng Anh, kiến thức lập trình cơ bản,… các bạn nên chú ý.

Những việc cần làm trước khi đi phỏng vấn

- **Tìm đường đến nơi phỏng vấn**: Nếu có thể, hay đi ngang nơi phỏng vấn trước đó một ngày. Điều này giúp bạn ước lượng được khoảng thời gian đi. Tới hôm phỏng vấn, bạn sẽ khá hồi hộp, lo lắng; biết trước đường tới nơi phỏng vấn sẽ giúp bạn tự tin hơn so với việc chờ tới hôm phỏng vấn mới tìm.
- **Tìm hiểu công ty bạn sắp phỏng vấn:** Đây là điều quan trọng nhất mà khá nhiều bạn bỏ sót. Hãy lên trang web của công ty đó, xem và ghi nhớ những thông tin quan trọng như: Công ty tập trung vào lĩnh vực nào, môi trường làm việc ra sao, công ty coi trọng những giá trị nào của nhân viên… Biết những điều này, bạn sẽ có cái nhìn tổng quan về công việc mình sắp làm cũng như ghi điểm với nhà tuyển dụng khi trả lời.
- **Xem xét giá cả thị trường, yêu cầu lương hợp lý**: Có một thực tế phũ phàng là nếu bạn không đòi hỏi mức lương, nhà tuyển dụng sẽ trả cho bạn mức lương thấp nhất có thể (Ai cũng muốn tiết kiệm mà). Mức lương có thể được ghi rõ trong mẫu quảng cáo việc làm hoặc không. Bạn có thể tham khảo mức lương từ những người quen làm công ty đó, hoặc người quen có vị trí tương đương vị trí bạn muốn apply (ví dụ bạn muốn apply vị trí senior dev, bạn có thể hỏi người quen làm senior dev để biết khoảng lương).
- **Ôn lại kiến thức**: Xem lại kiến thức cơ bản về lập trình, OOP, tiếp theo là những kiến thức liên quan đến phần mà nhà tuyển dụng yêu cầu (C#, Java, SQL, MVC, Struts, …). Bạn có thể Google *C# interview questions* hoặc những thứ tương tự để tìm những câu hỏi phỏng vấn hay được hỏi. Bạn nên dành 1-2 tuần cho việc này.

Quy trình phỏng vấn

Như mình đã nói ở đầu bài, quy trình phỏng vấn ở các công ty thường khác nhau, tuy nhiên nó thường bao gồm các bước sau:

1. Phone interview hoặc entry test

Đây là vòng để sàng lọc ứng viên, tiết kiệm thời gian cho nhà tuyển dụng. Bạn sẽ bị hỏi một số kiến thức cơ bản về ngôn ngữ, framework mình ghi trong CV (C#, MVC, …). Qua buổi phỏng vấn ngắn (20-30p), người phỏng vấn sẽ quyết định có nên mời

bạn đến cty phỏng vấn hay không. Một số công ty khác tổ chức entry test, thay vì phỏng vấn thì bạn sẽ làm một bài test nho nhỏ về lập trình.

2. Face interview – Vòng quan trọng nhất

Nếu mọi chuyện êm xuôi, bạn sẽ tới công ty để phỏng vấn. Hãy nhớ rằng đây cũng là cơ hội để bạn phỏng vấn ngược lại nhà tuyển dụng. Bạn có thể quan sát công ty để có cái nhìn tổng quan về không khí làm việc và môi trường làm việc. Dưới đây là những câu hỏi bạn sẽ được hỏi khi phỏng vấn:

- Giới thiệu bản thân? Bạn có thể nói sơ về số năm kinh nghiệm, sở thích về công nghệ, vị trí muốn làm (Ngắn gọn thôi nhé).
- Hãy nói về 1 dự án bạn đã làm? Bạn làm vai trò gì? Người phỏng vấn sẽ hỏi khá kĩ về cấu trúc của dự án, công việc bạn làm, khó khăn bạn gặp phải, cũng như cách bạn xử lý chúng. Họ sẽ đánh giá được nhiều điều từ bạn qua câu hỏi này.
- Một loạt các câu hỏi technical liên quan tới những gì bạn ghi trong CV. Bạn sẽ được hỏi từ back-end (Database, SQL, Entity Framework, LINQ, Delegate), cho tới front-end (HTML, CSS, jQuery, AngularJS,…), và những điều liên quan tới framework (MVC routing, model mapping, …). Ngoài những câu lý thuyết, có thể bạn sẽ được yêu cầu giải quyết vấn đề. Ví dụ: Anh muốn submit một form bằng Ajax phải thế nào?
- Một loạt câu hỏi liên quan đến OOP, kiểu dữ liệu, thuật toán v..v. Đây là những câu hỏi về kiến thức cơ sở, dùng để đánh giá kiến thức nền tảng của bạn, không liên quan đến ngôn ngữ hay framework. Có thể bạn sẽ bị bắt viết code trên giấy hoặc trên máy để giải một số câu hỏi đánh giá trình độ. Với các bạn sinh viên thì người phỏng vấn sẽ tập trung vào phần này hơn là công nghệ.
- Một số câu hỏi cá nhân để không khí bớt căng thẳng: Bạn có sở thích gì? Bạn có điểm yếu điểm mạnh gì? Cứ trả lời thành thật nhé.

Cuối buổi phỏng vấn, bạn sẽ được hỏi câu cuối cùng: Bạn có câu hỏi gì không? Theo kinh nghiệm, các bạn có thể hỏi về những điều sau đây:

- Công ty mong chờ điều gì ở mỗi nhân viên?
- Hỏi người phỏng vấn một ngày làm việc của họ hoặc thành viên trong nhóm như thế nào?
- Môi trường làm việc ra sao, có bắt làm thêm giờ không?
- Chính sách review tăng lương như thế nào?
- Ngoài lương ra công ty có những ưu đãi nào? Công ty có tổ chức seminar hay chính sách gì để giúp nhân viên phát triển không?

Những câu hỏi này sẽ thể hiện bạn có tinh thần làm việc nghiêm túc, biết suy nghĩ đến tương lai. Chúng cũng giúp bạn hiểu thêm về công ty mà mình sắp làm việc.

Người phỏng vấn bạn có thể là Project Manager hoặc Team Leader của một dự án. Họ không chỉ đánh giá bạn qua khả năng kĩ thuật mà còn qua thái độ giao tiếp và trả lời câu hỏi. Nếu gặp câu hỏi bạn không biết, hãy thành thật bảo rằng mình không biết nhưng thể hiện thái độ sẵn sàng học hỏi, bạn vẫn sẽ được đánh giá tốt nhé.

3. Sau buổi phỏng vấn

Sau buổi phỏng vấn, nhớ gửi một email cảm ơn cho người đã phỏng vấn mình (nếu không có email trực tiếp, hãy cảm ơn bên nhân sự, nhờ chuyển lời cảm ơn đến người đó). Đây là một điều nho nhỏ, hiệu quả lại lớn mà phần đông các bạn thường "quên" không làm.

Nếu mọi chuyện ổn thỏa, bạn sẽ tiếp tục vòng phỏng vấn với quản lý hoặc bên nhân sự để đàm phán về lương bổng cũng như thời gian bạn có thể bắt đầu đi làm. Sau khoảng 2-10 ngày làm việc, bạn sẽ nhận được điện thoại hoặc email thông báo kết quả phỏng vấn.

Nếu được nhận vào làm việc, bạn sẽ nhận được một offer letter. Trong offer letter này có ghi rõ về lương bổng, số ngày phép, số giờ làm việc. Các bạn nhớ đọc kĩ, nếu có gì thắc mắc thì nhớ hỏi lại chứ đừng vội vàng kí. Bạn cũng nên chuẩn bị đầy đủ giấy tờ để nộp cho phòng nhân sự vào ngày đầu tiên đi làm của mình nhé. Chúc các bạn may mắn!

Tóm tắt

- Ngành IT rất dễ tìm việc, một số kênh thường dùng: itviec, linkedin, vietnamwork.
- CV là thứ đầu tiên gây ấn tượng với nhà tuyển dụng. Hãy trình bày rõ ràng, nội dung chân thật, không thổi phồng khả năng của bản thân.
- Trước khi phỏng vấn, hãy chịu khó tìm hiểu công ty, tìm hiểu thị trường và ôn lại kiến thức.
- Luôn tỏ thái độ thân thiện hòa nhã khi phỏng vấn. Sau khi kết thúc phỏng vấn, nhớ gửi email cảm ơn.

Phần 3 – Làm việc

Sau khi tốt nghiệp ra trường và vượt qua vòng phỏng vấn gian khổ, bạn sẽ được nhận vào một công ty phần mềm. Bắt đầu trong một môi trường mới, bạn sẽ khá bỡ ngỡ khi lần đầu tiếp xúc với dự án thực tế, tuân theo qui trình, làm việc theo nhóm. Hẳn bạn cũng sẽ có vài câu hỏi về nghề nghiệp, về tương lai. Những câu hỏi đó sẽ được trả lời trong phần này.

CON ĐƯỜNG PHÁT TRIỂN SỰ NGHIỆP (CAREER PATH) CHO DEVELOPER

Các bạn sinh viên còn đang học hoặc mới ra trường sẽ khó hình dung được về những vị trí và chức danh trong ngành lập trình. Bài viết này sẽ giải đáp một số thắc mắc của các bạn như:

- Mới đi làm em có chức danh gì, công việc thế nào?
- Code lâu thì lên được chức gì, lương có cao không?
- Em chỉ thích code thôi, không thích làm trưởng nhóm, em nên định hướng thế nào.

Hiểu rõ con đường nghề nghiệp của ngành developer, các bạn sẽ dễ định hình phát triển tương lai của bản thân, cũng như dồn sức vào con đường mình đã chọn.

Mình chỉ liệt kê con đường nghề nghiệp của một developer vì bản thân mình cũng là developer. Con đường của 1 tester (QA engineer) cũng có một số chức danh tương tự nhưng lên cao sẽ khác. Các chức vụ sẽ được miêu tả theo thứ tự từ thấp lên cao nhé.

Lưu ý: Mức lương đề cập trong bài dựa theo mức lương và quảng cáo tuyển dụng của các công ty ở TP. HCM, ở các tỉnh thành khác sẽ có một số chênh lệch.

Fresher/Junior Developer

Các bạn sinh viên đi thực tập hoặc mới ra trường thường được chức danh này. Kinh nghiệm của Junior Developer thường vào khoảng 6 tháng – 1 năm. Mức lương thì tùy vào khả năng của bạn, thường dao động trong khoảng 300-500$.

Do chưa có kinh nghiệm nên fresher/junior thường được các công ty đào tạo lại, vì vậy khi phỏng vấn vị trí fresher các công ty thường chỉ chú trọng khả năng suy nghĩ logic, kiến thức lập trình cơ bản và tiềm năng phát triển của bạn. Cá nhân mình thấy chương trình đào tạo Fresher của FSOFT cũng khá tốt, có dạy nhiều thứ mà bạn sẽ tiếp xúc khi làm việc.

Do chưa có kinh nghiệm nên mọi người thường không đòi hỏi ở bạn quá cao. Công việc của một junior thường là tìm hiểu dự án hiện tại, code các module đơn giản, fix bug với sự trợ giúp và review của senior. Ở giai đoạn junior, các bạn hãy cố gắng tranh thủ học cách code, cách thức làm việc và kinh nghiệm của các bác senior đi trước.

Developer

Code được một thời gian khoảng 1-2 năm, các bạn sẽ được gọi là Developer (Nhiều bác lên thẳng vị trí Team Leader hoặc Senior tùy vào công ty). Ở giai đoạn này, bạn đã làm qua một số project, khá rành về một số công nghệ. Mức lương của developer vào khoảng 600-900$.

Phỏng vấn cho developer dĩ nhiên là khó hơn junior. Người phỏng vấn sẽ hỏi bạn về những dự án bạn đã làm, các khó khăn bạn đã gặp phải và cách giải quyết? Ngoài ra, buổi phỏng vấn sẽ tập trung vào những công nghệ bạn đã ghi trong CV. Vì developer đã có kinh nghiệm, khi đi làm các bạn sẽ không còn "được" các anh senior kèm cặp, và cũng khó mà lấy danh nghĩa junior để hỏi, nhờ vả hay mắc lỗi nữa.

Ở giai đoạn này, bạn đã được code một số module phức tạp hơn, tham gia họp, code review, thảo luận với khách hàng v…v. Đây là giai đoạn để bạn dồn nén kiến thức, kinh nghiệm và gây dựng danh tiếng để lên nấc tiếp theo trong bậc thang nghề nghiệp.

Lý thuyết là vậy nhưng thực tế, thuở làm ở FSOFT mình ở vị trí junior được khoảng một tháng rồi nhảy vào làm các công việc của developer, nhận cả những việc khó chứ không đùn đẩy gì, nhờ vậy cũng mình học hỏi được khá nhiều.

Quản lý hay kĩ thuật?

Ở giai đoạn sau, bạn đã có thể xác định con đường cho mình. Nếu muốn tập trung vào code và kĩ thuật, bạn có thể đi theo hướng kĩ thuật: Senior Developer => Technical Lead => Software Architecture. Nếu muốn làm việc với quy trình và con người, bạn nên đi theo hướng quản lý: Team Lead => Project Manager => Manager.

Ở giai đoạn đầu, lằn ranh giữa 2 con đường này khá mờ nhạt, nhưng càng lên cao lại càng trở nên rõ ràng. Các bạn có thể xem bảng tóm tắt sau:

Hướng quản lý **(Management)**	**Hướng kĩ thuật** **(Technical)**
Team Leader Bạn trở thành trưởng một nhóm nho nhỏ khoảng 3-6 thành viên. Ngoại trừ code, bạn còn phải họp hành với cấp trên, báo cáo với khách hàng, quản lý cấp dưới. Ở giai đoạn này, bạn sẽ dần học thêm một số kĩ năng lãnh đạo, kĩ	**Senior Developer** Sau một thời gian làm việc, bạn nắm vững, hiểu sâu và rộng nhiều công nghệ và qui trình. Ở vị trí này, ngoại trừ khả năng code "thần thánh", bạn còn phải biết đưa ra design và solution. Ngoài ra, bạn còn phải

năng quản lý v…v. Ở một số cong ty nhỏ, developer lâu năm và có kinh nghiệm sẽ lên team leader. Bạn vẫn còn khá nhiều thời gian code, code giỏi có thể sẽ làm thành viên trong team tôn trọng hơn. Mức lương cho team leader thường vào khoảng 1000-1500$	hướng dẫn chỉ bảo các em junior mới vào, cũng như tham gia code review v…v. Đôi khi senior dev cũng kiêm luôn vị trí team leader, do đó bạn cũng cần một chút kĩ năng diễn đạt và lãnh đạo. Mức lương cho Senior Developer cũng vào khoảng 1000-1500$ (hoặc hơn).
Project Manager Lên đến vị trí này, bạn sẽ có rất ít hoặc hầu như không có thời gian code. Đa phần thời gian của bạn dùng để đọc báo, lướt voz, lướt webtretho …. Đùa đấy, công việc chính của bạn bây giờ là quản lý cấp dưới, báo cáo với khách hàng và cấp trên về tiến độ dự án, lâu lâu bạn còn phải đi phỏng vấn một số ứng viên để tuyển thành viên cho dự án nữa. Bạn là người quyết định sự thành bại của một dự án, do đó nếu dự án thành công bạn sẽ được thưởng một khoản bonus kha khá tùy theo chính sách công ty. Mức lương cho PM vào khoảng 1000-2000$.	**Technical Leader** Bạn cần hiểu biết về công nghệ sâu và rộng vì chính bạn là người lựa chọn công nghệ, qui trình… cho một dự án. Những quyết định lớn về thiết kế, cấu trúc code … sẽ do bạn chịu trách nhiệm. Ở giai đoạn này, ngoài việc technical "cứng", bạn còn phải giỏi thuyết trình, hướng dẫn, giải thích… Vì sao á? Khi đưa ra vấn đề thì phải giải thích hợp lý, thành viên khác nó mới hiểu, nể và làm theo chứ. Mức lương cho vị trí này vào khoảng 1500-2500$.
Manager/Director Chúc mừng, ở vị trí này bạn đã được gọi là sếp, cấp trên, lãnh đạo,… Lúc này bạn sẽ không có thời gian mà code. Việc của bạn là họp hành, giao việc, phỏng vấn, trao đổi với các bộ phận, phòng ban, xử lý việc hành chính… Bạn còn phải đề ra định hướng phát triển của bộ phận mình quản lý, xây dựng qui trình làm việc, tuyển dụng,… Chế độ đãi ngộ cho vị trí	**Software Architect** Muốn đạt chức danh này, ít nhất bạn phải có khá nhiều năm trong ngành. (Nhìn ai mặt mũi trẻ măng mà vỗ ngực tự xưng SA thì đừng tin). Công việc của bạn khá gian khổ: Từ một yêu cầu "mơ hồ" của khách hàng, bạn phải làm việc với BA để đánh giá solution, làm việc với PM để xây dựng một team, làm việc với Technical Lead để thiết kế, đưa ra các quyết

này chênh lệch khá lớn tùy công ty và cấp bậc manager nên mình không có con số cụ thể.	định quan trọng về kiến trúc hệ thống. Vị trí này mặc dù không có quyền quản lý, nhưng lại có khá nhiều quyền lực ngầm.

Ngoài những con đường trên, các bạn có thể đi theo hướng Sales, Kĩ sư cầu nối (BrSE), Business Analyist. Bạn không cần phải quyết định con đường nghề nghiệp từ quá sớm. Sau khi đi làm một thời gian bạn sẽ tự nhận ra điểm mạnh, điểm yếu của mình và chọn được con đường phù hợp thôi.

Tóm tắt

- Có khá nhiều lựa chọn nghề nghiệp cho các bạn học ngành IT: developer, tester, BA.
- Mới đi làm, bạn sẽ được gọi là fresher hoặc junior. Làm việc một thời gian bạn "lên chức" developer, dĩ nhiên là cũng lên lương.
- Bạn có thể phát triển theo hướng quản lý và kĩ thuật, tùy vào khả năng và sở thích của bản thân.

MẶT TỐI CỦA NGÀNH CÔNG NGHIỆP IT

Ngành lập trình kể ra cũng có khá nhiều cái sướng: Dễ xin việc, công việc thú vị, tiếp xúc nhiều cái mới, mức lương khá. Tuy vậy, nó có không ít mặt tối mà chỉ những người có thâm niên và tiếp xúc lâu với nghề mới trải nghiệm và nhận ra được.

Bài viết này lấy cảm hứng từ khóa học cùng tên trên pluralsight: Technology Career Dark Side, nhằm giúp bạn đọc có cái nhìn khách quan hơn về ngành IT, cũng như tự rút ra cách "sống sót" cho bản thân mình.

Công nghệ liên tục thay đổi

Trong ngành lập trình, công nghệ là thứ liên tục thay đổi. Những công nghệ mới liên tục ra đời thay thế công nghệ cũ, làm kiến thức rất dễ lỗi thời vì. Do đó, người lập trình viên phải liên tục học hỏi, nếu không họ sẽ trở nên lạc hậu.

Nguyên nhân sâu xa đằng sau chuyện này chính là Tiền. Tại sao mỗi năm FIFA và PES đều ra phiên bản mới? Để bán lấy tiền. Tại sao mỗi năm iPhone lại ra phiên bản mới? Để hút máu người dùng, đế kiếm tiền. Đó cũng là lý do các hãng công nghệ liên tục đưa ra các sản phẩm/công nghệ mới: C# thay đổi từ 2.0 tới 5.0, Windows mỗi 2-3 năm lại ra bản mới, Visual Studio và SQL Server cũng tương tự.

Nhìn chung, sự thay đổi này có mặt tích cực của nó: Các framework/thư viện mới có nhiều tính năng hơn, giúp việc code nhanh và dễ dàng hơn. Tuy nhiên, điều đó cũng đi kèm không ích phiền toái: Mỗi phiên bản lại có chút ít cập nhật và thay đổi, làm việc nâng cấp hay tích hợp rất mệt mỏi và mất thời gian. Phiền toái lớn nhất chính là: Tốn công sức, thời gian mà lập trình viên đã bỏ ra để học ngôn ngữ đó.

Đôi khi một công nghệ chết đi, không được tiếp tục phát triển hay hỗ trợ nữa (VB6, Silverlight). Thử tưởng tượng bây giờ MS SQL không được phát triển tiếp xem, một đống dự án sử dụng MS SQL sẽ lao đao. Đó là lý do các ngôn ngữ/công nghệ như Java, C#, PHP, MySQL vẫn được ưa chuộng so với những thứ mới mẻ như NodeJS, MongoDB, … vì chúng có tuổi đời lâu hơn, đáng tin tưởng hơn.

Vòng đời của một công nghệ: Một công nghệ mới ra đời, được nhiều công ty sử dụng, nhu cầu tuyển dụng cao nên nhiều người theo học. Qua năm tháng, công nghệ đó chết dần, không ai tuyển nữa nên ít người học. Tuy nhiên, ứng dụng của các công ty được xây dựng bằng các công nghệ cũ, họ vẫn cần developer để bảo trì hay nâng cấp ứng dụng. Đó là lý do các developer Cobol, VB6, Fortran vẫn rất được giá (Các công ty phần mềm Nhật có nhiều hệ thống lớn và siêu lớn xây dựng bằng các công nghệ này, toàn outsource cho dân VN bảo trì).

Hackathon và Stackoverflow không "tốt đẹp" như bạn nghĩ

Hackathon là một cuộc thi code nho nhỏ, thường tổ chức vào hai ngày cuối tuần. Các lập trình viên sẽ lập thành một đội khoảng 3-5 người để xây dựng một sản phẩm. Giải thưởng (khá hấp dẫn) sẽ được trao cho các đội xuất sắc nhất.

Mặt tốt của hackathon rất nhiều và dễ thấy. Các sự kiện này thường rất vui và có ích cho lập trình viên: Họ được vui chơi, được code, được tặng quà, xây dựng quan hệ, kết nối với cộng đồng. Tuy nhiên, mặt xấu thì ít người để ý. Hackathon được tổ chức với lý do là "phát triển cộng đồng, giới thiệu công nghệ". Thật ra, đây là một cách rẻ mạt để chôm thời gian và công sức của lập trình viên vào 2 ngày cuối tuần. Luật bản quyền cũng ít khi được coi trọng (ứng dụng tham gia hackathon, mặc định bản quyền thuộc về hackathon) nên bạn có thể dễ dàng bị chôm hoặc đạo ý tưởng mà không làm được gì.

Stackoverflow là một mạng hỏi-đáp dành cho lập trình viên. Tham gia stackoverflow sẽ giúp bạn nâng cao trình độ và kiến thức. Tuy nhiên, tham gia stackoverflow, bạn đã vô tình tham gia vào đội ngũ "chuyên gia" miễn phí, cho stackoverflow thu tiền quảng cáo. Nó còn áp dụng Gamification (Điểm, huy hiệu) để dụ dỗ bạn đóng góp thời gian, công sức. Một câu trả lời trên stackoverflow tiết kiệm cho bạn 1 tiếng fix lỗi, bạn bỏ ra 15 phút ngồi đóng góp lại cũng ko sao, nhưng đừng bỏ 3, 4 tiếng vào đó nhé.

Dù vậy, mình vẫn khuyên các bạn tham dự các hackathon và stackoverflow, chúng rất hữu ích với mọi lập trình viên. Nhưng nhớ biết nhận thức và dè chừng những mặt xấu của nó nhé.

Vấn đề outsource và bóc lột

Ở Việt Nam có khá nhiều công ty outsource (FPT, Harvey Nash, …). Các công ty này cung cấp phần lớn công ăn việc làm, nâng cao trình độ lập trình viên, đóng góp vào GDP nước nhà. Outsource cũng không phải hoàn toàn xấu, nhưng quá lệ thuộc vào outsource, chỉ cắm đầu vào gia công phần mềm thì ngành IT nước nhà sẽ rất khó phát triển.

Vì sao các công ty nước ngoài lại outsource, chung quy cũng (lại) là vì tiền. Giá nhân sự của Việt Nam, Ấn Độ rẻ hơn lập trình viên nước ngoài nhiều. Các công ty outsource Việt Nam phải trả phí máy móc, lấy hợp đồng, trả một phần nhỏ làm lương cho lập trình viên.

Nhắc tới chuyện bóc lột, hình thức bóc lột "dã man" nhất là làm thêm giờ, còn gọi là OT (Overtime). Nếu bạn xui xẻo rơi vào dự án "cháy", OT là chuyện như cơm bữa. Bạn sẽ phải đi làm tới 8-9 giờ tối hoặc cả thứ 7 chủ Nhật. Điều này ảnh hưởng không nhỏ đến sức khỏe cũng như quan hệ xã hội và gia đình. Mình từng nghe "truyền

thuyết" về một team ở công ty cũ, do OT ăn mì tôm riết mấy tháng mà cả nhóm bị mỡ trong máu.

Về chuyện tiền nong, có nhiều công ty trả tiền OT rất sòng phẳng, cũng có nhiều nơi quy ra ngày phép, hoặc xù luôn. Làm với Nhật thì không cái khái niệm OT, đi làm 9-10h tối mới về là "văn hóa làm việc" bên đó.

Giỏi technical không chưa đủ

Xin nhắc lại một câu nói từ bài viết cũ:

> Một điều khá may mắn trong ngành IT là: Vì đây là một ngành nặng về kĩ thuật. do đó bạn chỉ cần giỏi tập trung trau dồi technical cho giỏi. Chỉ cần technical giỏi, bạn sẽ được đồng nghiệp coi trọng, cấp trên tin tưởng giao phó trách nhiệm. Chỉ cần technical giỏi, con đường sự nghiệp của bạn sẽ rộng thênh thang, bạn sẽ nhanh chóng leo lên vị trí senior, team leader, technical lead, … Chỉ cần technical giỏi, lương bạn sẽ tăng vù vù, từ 500$, 1000$, 2000$, các quảng cáo tuyển dụng toàn cần người giỏi technical còn gì?

Nhiều bạn ngáo ngơ mới ra trường vẫn tưởng điều này là đúng, thấy mình hơi giỏi là tỏ thái độ khi phỏng vấn. Môi trường làm việc đòi hỏi nhiều thứ hơn cả khả năng lập trình, đó là khả năng giao tiếp, khả năng trình bày và thuyết trình, kĩ năng thương thảo, khả năng làm việc nhóm, … Tới những cấp bậc cao hơn như team leader, quản lý, bạn còn phải biết cách quản lý cấp dưới, phân chia công việc, cách thuyết phục và nói cho người khác nghe. Khả năng code là bắt buộc, nhưng chỉ code không là chưa đủ nếu bạn muốn sống sót và thăng tiến trong ngành này.

Cấp trên thường "đéo biết gì cả"

Đôi khi nói cấp trên "đéo biết gì" cũng không sai. Một sự thật buồn cười của ngành mình là: Nếu giỏi technical, developer sẽ được… đẩy lên làm quản lý. Ở cấp quản lý, họ ít có thời gian code, mà phải lo phân chia công việc, quản lý nhân sự, quản lý dự án,… toàn những việc trước giờ họ chẳng bao giờ làm. Ở cương vị mới, họ cũng phải học dần dần về quản lý, giống như chúng ta mới bắt đầu học code vậy thôi.

Các bạn lập trình viên thường có suy nghĩ rằng "Ông team leader/PM có code bao giờ đâu, công nghệ mới cũng *éo biết". Điều này đúng, vì họ còn phải lo nhiều việc khác: nhân sự, tài nguyên, phỏng vấn ứng viên, hoặc đấu đá nội bộ công ty ,… không có đủ thời gian để trau dồi kĩ thuật. Đôi khi tranh cãi về một vấn đề, bạn chỉ thấy vấn đề technical, họ còn phải chú ý đến nhiều khía cạnh khác như: sắp xếp team, tiến độ dự án, trình độ thành viên, …

Do vậy, đừng giữ thái độ thù hằn, cay cú hay xem thường cấp trên. Hãy cứ cố gắng thấu hiểu, tôn trọng và học hỏi họ đi, biết đâu sẽ có lúc bạn "bị" đẩy lên làm quản lý

như họ thôi. Gợi ý nhỏ: Nếu cấp trên thật sự "ngu" và bạn không chịu nổi, cứ thoải mái chuyển công ty hoặc chuyển phòng ban khác thôi, ngành mình cũng dễ xin việc lắm.

Lập trình viên dần đang mất giá

Hôm trước, mình nhận được email giới thiệu việc làm thêm của vlance – một trang web cho tìm việc làm freelance Việt Nam. Nhìn giá cả các dịch vụ IT ở Việt Nam mà thấy thương: Một ứng dụng chỉ có giá 2 triệu, một website chỉ khoảng 1 triệu. Một phần do các công ty IT ở Việt Nam phá giá, làm mọi thứ từ web tới app với giá rẻ mạt. Một phần do các bạn developer Việt Nam đang bán rẻ chất xám, bán rẻ sức lao động của chính mình.

Ảnh từ vlance.vn, một trang tìm việc freelance

Ở đầu bài, mình từng nói rằng ngành lập trình viên có mức lương kha khá. Với các vị trí như cao như senior, team leader, PM thì lương 1000-3000$/tháng là chuyện bình

thường[30]. Tuy nhiên, mức lương ở các vị trí thấp hơn lại … khá là thấp. Một phần là do nhiều bạn trẻ đổ xô đi học công nghệ thông tin, học lập trình, khi ra trường lại không đủ trình độ, đành ngậm ngùi làm những công việc nhàm chán, mức lương thấp.

Công nghệ hiện đại làm mọi người dễ tiếp cận việc lập trình hơn, nên người dùng có thể làm web, làm ứng dụng dễ dàng. Lập trình viên thì ngày càng "lười" hơn, "nhàn" hơn nhờ các framework[31]. Trong tương lai, có thể ngành lập trình sẽ không còn hot như bây giờ, lập trình viên sẽ rớt giá "thê thảm".

Thế giờ em phải sống làm sao?

Nghe mình "hù" nãy giờ, chắc bạn cũng hơi nghi ngại rồi phải không? Đừng lo, hãy áp dụng một số lời khuyên dưới đây của mình:

- Nắm vững kiến thức nền tảng, học nhiều ngôn ngữ chứ đừng ôm khư khư 1 ngôn ngữ. C++, C#, Java, … có hay đến đâu cũng chỉ là công cụ. Quan trọng là bạn làm được gì với ngôn ngữ mình biết.
- Ngành mình có rất nhiều developer, nhưng có rất ít developer giỏi. Hãy tự trao dồi kiến thức và kĩ năng để tự nâng giá bản thân nhé, đã giỏi thì không lo thất nghiệp hay lương thấp.
- Xác định con đường mình muốn đi: Manager hay Technical, hoặc mở công ty riêng. Học thêm mấy thứ ngoài luồng như quản trị, kinh tế, khởi nghiệp,… sau này sẽ có ích lắm đấy.
- Sống khôn hơn và đừng ngáo ngơ nữa. Chịu khó đi xã giao, tạo dựng danh tiếng, quan hệ. Hãy đọc blog của một số developer nổi tiếng để học cách suy nghĩ, cách làm việc của họ.

Tóm tắt

- Trong ngành IT, công nghệ liên tục thay đổi nên kiến thức của bạn rất dễ lỗi thời
- Hackathon và stackoverflow là nơi rất tốt để bạn thể hiện, tạo quan hệ, nhưng cần để ý đến mặt trái của nó
- Ngành IT có cường độ làm việc cao, đôi khi OT nhiều, ảnh hưởng đến sức khỏe

[30] Xem lại bài viết "Con đường nghề nghiệp của lập trình viên"

[31] Xem thêm trong bài "Điều gì ngăn cản bạn đạt đến cảnh giới tối cao trong code học"

- Đừng nghĩ rằng cấp trên "ngu" hay "không biết gì", họ cũng từng là developer như bạn đấy
- Lập trình viên ở những vị trí thấp đang dần "mất giá"

TOP 18 SAI LẦM MÀ CÁC LẬP TRÌNH VIÊN "NON TRẺ" HAY MẮC PHẢI

Bài viết này nói về những sai lầm ta hay phạm phải khi bước những bước đầu tiên trên con đường lập trình. Để trưởng thành hơn trên con đường phát triển bản thân, phát triển sự nghiệp, ta cần phải học hỏi không ngừng. Có một câu danh ngôn khá hay là: "Hãy học từ sai lầm của người khác. Bạn sẽ không bao giờ sống đủ lâu để phạm phải tất cả sai lầm."

Những sai lầm này được chia sẻ bởi các lập trình viên trên Quora, nơi tụ họp của rất nhiều lập trinh viên giỏi và nổi tiếng trên toàn thế giới, cùng với một phần trải nghiệm cá nhân mình. Hi vọng chúng sẽ là hành trang có ích cho những bạn lập trình viên còn "non trẻ".

Sai lầm trong phát triển bản thân

Có thể tóm gọn các sai lầm này trong "5 chữ Không" và "4 chữ Luôn":

1. Không bao giờ đọc sách kĩ thuật hoặc tự tìm hiểu ngôn ngữ hay công nghệ mới.
2. Không bao giờ đọc các blog lập trình để học hỏi, tìm hiểu các xu thế mới.
3. Không quan tâm đến sức khỏe bản thân: thức khuya cày game, rượu chè nhậu nhẹt, code thâu đêm suốt sáng.
4. Không bao giờ đào sâu nghiên cứu mà chỉ học hời hợt những thứ bề mặt để hoàn thành công việc.
5. Không quan tâm tới việc xây dựng quan hệ, tạo dựng hình ảnh hay phát triển sự nghiệp.
6. Luôn chỉ biết tập trung vào dự án của mình làm mà không quan tâm đến tình hình công ty hay các team khác.
7. Luôn trông chờ vào việc team leader hay quản lý sẽ định hướng nghề nghiệp, cất nhắc mình: Quản lý có nhiều việc phải làm và nhiều nhân viên cần quản lý, rất khó để họ có thời gian chăm chút định hướng cho một cá nhân.
8. Luôn thích làm việc với những đứa "dốt" hơn mình để dễ cầm đầu. Muốn giỏi hơn, cần phải làm việc với những người giỏi hơn mình, được dạy bảo hàng ngày mới khá lên được.
9. Luôn sống trong "comfort zone", chỉ làm những việc mình đã giỏi hay đã biết, không muốn mở rộng những kĩ năng đã có. (Xưa mình từng nghe có

một chị senior C# 6 năm bảo trì dự án WinForm, phỏng vấn vào ASWIG và rớt vì… không biết MVC).

Sai lầm trong thái độ làm việc

10. Giữ thái độ cao ngạo và biết tuốt: Có thể gặp ở các bạn sinh viên trường danh giá, tốt nghiệp loại giỏi (Mình chỉ nghe một số bác kể lại thôi, bạn nào sinh viên trường nổi tiếng, tốt nghiệp loại giỏi mà không có thái độ này thì cứ nghĩ "chắc thằng code dạo nó chừa mình ra").
11. Tỏ thái độ khi có người góp ý cho design hoặc code của mình: Tương tự cái ở trên. Kĩ năng code chỉ là một phần, quan trọng phải là "thái độ".
12. Giữ bo bo kiến thức cho mình, không chia sẻ vì sợ bọn đồng nghiệp sẽ giỏi hơn: Hồi xưa đi học mình cũng hay thế, giờ đã đi làm nên đỡ rồi, nhờ vậy các bạn mới có quyển sách này mà đọc đấy.
13. Luôn trông chờ được giao việc tận tay, chỉ làm những việc được giao: Để thăng tiến, bạn cần năng nổ, chủ động hơn hơn. Ví dụ nếu xong task sớm có thể chủ động bảo anh leader "để em phụ anh senior refactor code và đưa code lên CI[32] sau này dễ build"….
14. Luôn tuân thủ tuyệt đối mệnh lệnh của quản lý, team leader, senior mà không bao giờ hỏi nguyên do: Cấp trên, senior chưa chắc lúc nào cũng đúng. Tranh luận, phản bác đúng cách sẽ làm họ coi trọng mình hơn (Nhưng nhớ để ý thái độ, đừng tỏ ra hổ báo hay thể hiện).
15. Khi đi hỏi người khác, không xác định được vấn đề cần hỏi mà chỉ đặt câu hỏi mơ hồ chung chung, gây khó khăn cho cả người hỏi lẫn người trả lời.
16. Chỉ chăm chăm nhờ người khác làm giúp hoặc trả lời giúp: Gặp chút khó khăn là lên facebook để nhờ người khác Team Viewer giải giùm.
17. Quá bi quan/lạc quan/chủ quan khi đánh giá thời hạn hoàn thành công việc: Câu nói hay gặp "Task này em làm 1 ngày là xong". Một tuần sau: "Anh ơi do nó có bug ẩn nên em làm mới được có 30% hà :'(".
18. Tự hào vì mình làm việc chăm chỉ: Nếu đồng nghiệp chỉ làm 8 tiếng/ngày mà bạn phải làm việc tới 10 tiếng/ngày, đừng tự hào là mình chăm chỉ, mà hãy tự hỏi: Do mình dốt làm chậm hay do ông leader bóc lột giao việc cho mình nhiều quá???

Nếu đọc hết danh sách mà các bạn thấy bản thân mình dính hơi bị nhiều "sai lầm" thì cũng đừng lo nhé, cứ từ từ mà sửa thôi! Kể cả những lập trình viên giỏi nhất cũng từng mắc phải những sai lầm như vậy, huống chi là chúng ta. Điểm khác biệt là ở chỗ:

[32] Tìm hiểu trong bài "Giải thích đơn giản về CI"

những người giỏi nhận thấy được mình đang mắc sai lầm, từ đó phấn đấu và cải thiện bản thân hơn. Chúc các bạn may mắn trên con đường lập trình mình đã chọn.

Tóm tắt

- Các sinh viên mới ra trường hoặc vừa đi làm rất dễ phạm phải một số sai lầm trong thái độ làm việc và phát triển bản thân
- Nên nhận ra những sai lầm mình mắc phải, hậu quả của chúng để dần dần tìm hiểu và hoàn thiện mình

ĐỪNG COI NGÔN NGỮ LẬP TRÌNH NHƯ TÔN GIÁO

Chúng ta đang xem ngôn ngữ lập trình như một thứ tôn giáo

Giữa lập trình viên với nhau luôn có những cuộc tranh cãi liên tục bất tận về ngôn ngữ và công nghệ: Ngôn ngữ nào mạnh nhất, công nghệ nào tốt nhất. Ngôn ngữ, thứ vốn chỉ là công cụ, nay lại được nâng lên tầm tôn giáo.

Lập trình viên chia thành đạo Java, đạo PHP, đạo C#, đạo này công kích chửi bới đạo kia. Mức độ cuồng tín đôi khi chắc cũng không thua fan bóng đá, fan cuồng K-pop hay ISIS. Những cuộc cãi vã chê bai đầy rẫy trên mạng, các bạn có thể thử Google: Why C# sucks, Why Java sucks, Why PHP sucks, … để tìm hiểu.

Khi làm việc nhiều với một ngôn ngữ, một developer sẽ quen dần với ngôn ngữ đó, tìm ra được nhiều điều hay ho ẩn trong ngôn ngữ. Nhiều người sẽ nghĩ rằng ngôn ngữ của mình là nhất, có thể giải quyết được mọi vấn đề (Giống như ISIS nghĩ rằng đạo Hồi là nhất, mọi lời nói của đấng tối cao đều đúng đắn).

Khi ngôn ngữ mình thích bị chê bai, bị xúc phạm, họ cảm thấy như chính tôn giáo của mình bị xúc phạm. Họ xù lông nhím lên, kêu gọi bạn bè và đồng đội cùng đạo nhảy vào ném đá cho chết "cái thằng bố láo, dám chê Java, PHP, C++, … của bố". Ngày xưa mình là lập trình viên C#, mình cũng hay nhảy vào ném gạch khi nghe có đứa mở mồm chê C# và .NET.

Về bản chất, ngôn ngữ chỉ là công cụ

Ngôn ngữ lập trình chỉ là thứ chúng ta sử dụng chứ nó không định hình nên con người chúng ta. Để mở rộng tầm nhìn, bạn hãy thử tìm hiểu nhiều ngôn ngữ xem. Bạn sẽ ngạc nhiên khi thấy giữa chúng đều dựa trên một vài khái niệm và khuôn mẫu

chung. (Mình từng dùng MVC.NET, Struts2, Django, 3 framework của 3 ngôn ngữ khác nhau nhưng đều dựa trên khái niệm MVC cả).

Nói một các công bằng, ngôn ngữ nào cũng có cái hay của nó:

- C, C++ làm web khá cực và mất thời gian, nhưng để lập trình nhúng, lập trình game hay cần tốc độ thì khó ai bằng nó.
- JavaScript ban đầu là một ngôn ngữ kì dị điên khùng và bị ghét cay ghét đắng. Tuy nhiên do có vô số framework đi kèm nên hiện tại và nó đang trở thành một trong các ngôn ngữ hot nhất[33].
- PHP được thiết kế dở tệ (Vốn nó được tạo ra chỉ để viết mấy trang web nho nhỏ), nhưng có vô số framework, cộng đồng lập trình viên đông. Nó là lựa chọn số 1 nếu muốn tạo web nhanh, nhiều tính năng, ít lỗi.
- C# .NET, muốn dùng phải cài rất nhiều thứ nặng nề và tốn tiền. Nhưng nó lại được rất nhiều công ty sử dụng vì nhiều tính năng, bảo mật tốt, phát triển nhanh, v…v

Dừng tranh cãi lại, bớt gạch đá đi!

Xét cho cùng, thứ quan trọng không phải là ngôn ngữ mà mà là khả năng tư duy logic, kĩ năng giải quyết vấn đề, tầm nhìn hệ thống. Khách hàng sẽ đánh giá chúng ta qua sản phẩm – thứ họ thấy, chứ không ai quan tâm đến code bạn viết đâu.

Bạn có ngừng dùng Facebook vì nó viết bằng PHP – thứ ngôn ngữ cùi bắp không? KHÔNG. Bạn có bỏ stackoverflow khi biết nó được xây dựng dựa trên MVC.NET của C#, ngôn ngữ vừa chậm vừa mắc tiền không? Dĩ nhiên là KHÔNG. Vậy thì hãy đánh giá một lập trình viên qua thứ họ làm ra, chứ đừng thông qua ngôn ngữ họ sử dụng.

Thay vì chê bai, tranh cãi khi có người chê ngôn ngữ mình thích, hãy bỏ thời gian ra tìm hiểu và chia sẻ kiến thức. Giữ một cái nhìn khách quan về ngôn ngữ lập trình, bạn sẽ dễ dàng phát triển hơn. Như bản thân mình, ngày xưa mình cũng ghét JavaScript và PHP lắm, sau khi tự học nó lại thấy nó chúng có kha khá thứ thú vị đấy chứ.

Tóm tắt

- Lập trình viên chúng ta đang xem ngôn ngữ lập trình như một thứ tôn giáo, sẵn sàng xù lông nhím lên tranh cãi khi có người xúc phạm "tôn giáo" của mình

[33] Xem lời khuyên của mình trong bài "Học ngôn ngữ lập trình gì bây giờ?"

- Bản thân ngôn ngữ lập trình chỉ là công cụ, mỗi ngôn ngữ có những điểm mạnh điểm yếu riêng
- Thứ người dùng quan tâm là sản phẩm chứ không phải là ngôn ngữ hay công nghệ. Thay vì tranh cãi về ngôn ngữ, hãy tìm cách áp dụng nó để tạo ra sản phẩm tốt nhất

LẬP TRÌNH VIÊN NÊN ĐỌC NHỮNG SÁCH GÌ?

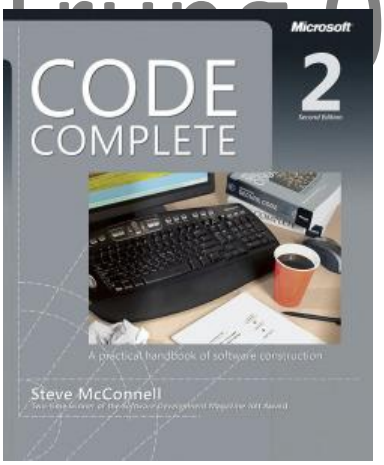
Nhiều bạn thường nghĩ rằng: developer thì cần gì phải đọc sách, code nhiều là giỏi thôi. Vâng, các cậu có cu, nhầm, các cụ đã có câu là "practice make perfect", cứ làm hoài là giỏi. Tuy nhiên, phải làm đúng cách thì mới giỏi được, code dở mà không chịu tìm cách cải thiện kĩ năng code, cứ code hoài mỗi kiểu cũ thì chẳng bao giờ giỏi được.

Mình đọc cũng kha khá nhiều sách kĩ thuật và sách về ngành lập trình, hay có dở có. Mỗi cuốn sách dù hay hay dở đều làm mình ngộ ra được vài điều. Khảo sát trong cuốn Code Complete cho thấy trung bình một developer đọc ít hơn một cuốn sách mỗi năm, ở Việt Nam chắc còn thấp hơn nữa.

Chỉ cần các bạn làm theo mình, mỗi năm đọc ít nhất một cuốn sách, các bạn sẽ giỏi hơn khoảng 90% developer còn lại rồi nhé. Bài viết này sẽ giới thiệu một số sách về kĩ thuật, về nghề nghiệp mà mỗi lập trình viên nên đọc. Các bạn có thể tìm bản ebook trên mạng, đặt mua sách từ amazon hoặc một số tiệm sách tiếng Anh nhé.

Sách về kĩ thuật lập trình

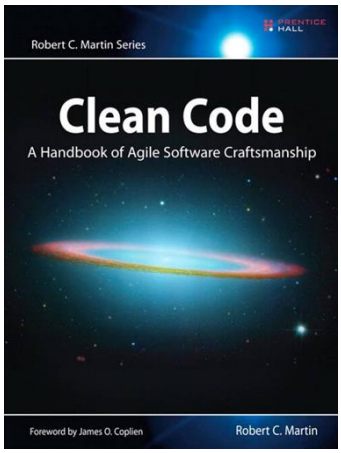
1. Code Complete



Nếu bạn muốn theo đuổi công việc lập trình một cách nghiêm túc, bạn nên đọc cuốn sách này. Mình được một ông anh giới thiệu. Có thể nói là đọc tới đâu ngộ ra tới đấy. Trong quá trình code, có lúc bạn sẽ gặp các trường hợp như: tách method thế nào, chia class ra sao, đặt tên biến thế nào, …

Cuốn sách này sẽ là người thầy, người anh của bạn, với vô số hướng dẫn từ tổng quan như: xây dựng kiến trúc, liên hệ giữa các component, … cho tới chi tiết như: cách tổ chức function, cách đọc tên biến.

2. Clean Code: A Handbook of Agile Software Craftsmanship

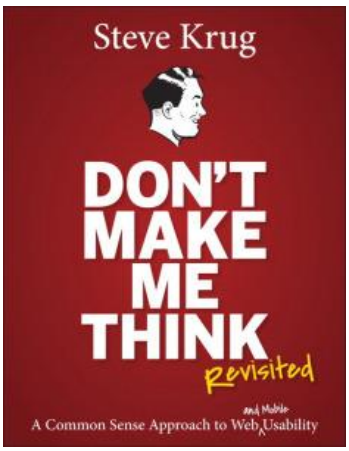


Mình đọc Clean Code trong thời gian còn làm việc ở FPT Software. Cuốn sách này xứng đáng là sách gối đầu giường của mọi developer. Mình khuyên các bạn nên mua bản gốc, vừa để đọc, vừa để phòng khi gặp đồng nghiệp code ngu, có thể cầm cuốn này đập vào đầu nó và bắt nó đọc.

Như cái tên "Clean Code", đây là cuốn sách hướng dẫn các bạn developer viết ra "code sạch". Thế nào là code sạch? Theo định nghĩa của sách, đó là code dễ đọc, dễ hiểu, dễ sửa chữa và bảo trì. Có bạn sẽ xì mũi bảo: Giời, có gì đâu, cái đấy ai code chả được. Mời bạn thử đọc lại code của một dự án mình đã làm cách đây 3-6 tháng, xem có hiểu được mình viết gì ko? Nếu không tức là code của bạn chưa được sạch lắm đâu. Cuốn sách sẽ thay đổi cách nghĩ cũng như cách viết code của bạn.

Sách về thiết kế và UI/UX

3. Don't make me think

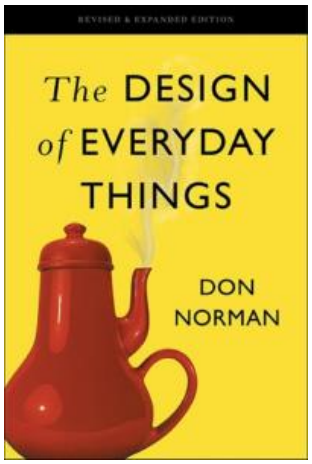


Một cuốn sách rất hay về thiết kế giao diện. Nó đưa ra một qui tắc rất đơn giản và hữu dụng trong thiết kế UI: Người dùng rất lười, hãy thiết kế sao cho người dùng ít suy nghĩ nhất. Cuốn sách không hướng dẫn cách thiết kế đẹp mà hướng dẫn cách thiết kế đơn giản nhất và dễ sử dụng đỡ tốn công sức người dùng nhất.

Sách còn hướng dẫn một số control nên dùng khi thiết kế web: form, checkbox, radio, dropdown, … và cách sử dụng những control này hợp lý. Ngoài ra còn có một câu chuyện về "Một button đáng giá 300 triệu đô" trong sách, về sự đắt giá của việc thiết kế giao diện[34].

[34] Đọc thêm về câu chuyện này trong bài viết "Một button trị giá 300 triệu đô"

4. The Design of Everyday Things

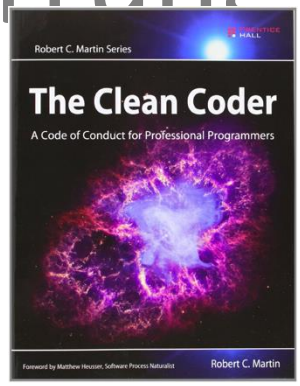


Cuốn sách này được viết bởi Don Norman. Ông là một bậc thầy về tâm lý học và design. Ông đưa ra khái niệm "user-centered design" (Design tập trung vào người dùng), chú trọng vào tính hữu dụng và dễ dùng của thiết kế, yếu tố thẩm mĩ chỉ là phụ.

Ý kiến cá nhân mình thì cuốn này khá hay, ảnh minh họa cũng rất nhiều. Sách sẽ làm thay đổi tư duy thiết kế của bạn, do đó dân design hay lập trình cũng đều nên đọc. Như tựa đề, cuốn sách phân tích thiết kế của các vật dụng thường ngày như: cánh cửa, tủ lạnh, máy lạnh, … cho tới điện thoại, hệ thống máy tính, … đồng thời phân tích tại sao design này thành công, design kia thất bại. Đảm bảo các bạn sẽ nhiều lần gật gù "à ra thế" khi đọc cuốn này.

Sách về nghề nghiệp và ngành lập trình

5. The Clean Coder

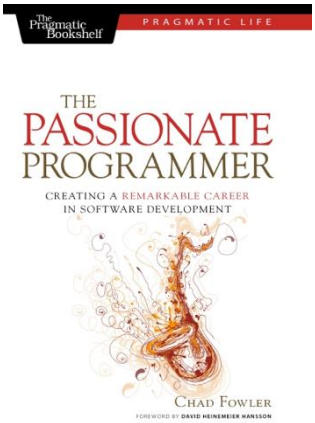


Cuốn sách mang tên The Clean Coder, đọc cũng giông giống Clean Code. Tuy nhiên, có một điều thú vị là nội dung hai cuốn sách lại… hoàn toàn trái ngược nhau!

Trong khi Clean Code tập trung vào khía cạnh kĩ thuật: hướng dẫn lập trình viên cách tổ chức code và viết code sạch; The Clean Coder lại tập trung vào khía cạnh nghề nghiệp: thái độ với công việc, cách làm việc nhóm, quản lý thời gian.

Trong The Clean Coder, Uncle Bob nói về nhiều khía cạnh: đạo đức nghề lập trình, ứng phó với áp lực công việc, rèn luyện làm mới kĩ năng, làm việc nhóm… Mỗi khía cạnh đều đi kèm với những trải nghiệm quí giá của chính tác giả qua hơn 42 năm tuổi nghề với đủ mọi vị trí từ dev, manager, PM cho tới CEO.

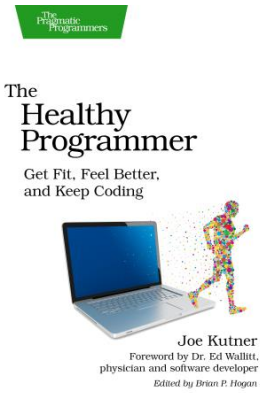
6. The Passionate Programmer



Đây là một cuốn sách viết về con đường phát triển sự nghiệp cho developer nói chung, cũng như dân IT nói riêng. Tác giả xem sự nghiệp IT như một sản phẩm hàng hóa. Để có một sản phẩm thành công, các doanh nghiệp thường thực hiện 4 bước: Nghiên cứu thị trường, đầu tư sản phẩm, phát triển sản phẩm, marketing.

Một cuốn sách khá hay với cách viết thú vị và dễ đọc. Sách vô cùng hữu dụng cho những bạn lập trình viên đang cảm thấy phân vân, cần những lời khuyên trong sự nghiệp.

7. The Healthy Programmer



Bạn có biết lập trình là một trong những ngành độc hại: Ngồi nhiều sẽ dẫn tới vô số những bệnh trầm trọng và mãn tính; lập trình viên rất dễ mắc các bệnh tim mạch, béo phì, đau khớp, ...?

Hãy đọc The Healthy Programmer – một cuốn sách viết về sức khỏe dành riêng cho developer. Cuốn này mình đọc từ hồi đầu năm khi tình cờ thấy nó trên một trang ebook. Như tựa đề, đây là một cuốn sách dành cho dân lập trình, nhưng nội dung không nói gì về lập trình mà lại nói đến một vấn đề mà lập trình viên nào cũng quan tâm: sức khỏe.

SỰ THẬT ĐẮNG LÒNG: ĐÔI KHI CẮM ĐẦU NGỒI CODE LÀ CÁCH … NGU NHẤT ĐỂ GIẢI QUYẾT VẤN ĐỀ

Xin bắt đầu bài viết bằng cách kể một câu chuyện nho nhỏ về một chàng coder nghèo tên K (Có thể tạm gọi là Khoa Khoe Khoang hay Khải gì đó tuỳ bạn).

Thời niên thiếu của K

Tiếp xúc với máy tính từ năm 10 tuổi, K vô cùng ngạc nhiên trước sức mạnh của cỗ máy vô tri vô giác ấy và nuôi mơ ước trở thành một lập trình viên. Lên cấp 3, nhờ giỏi Toán, K được vào lớp chuyên Toán của trường. Với niềm đam mê lập trình, K nhanh chóng tiếp cận và thành thạo Pascal và C, giật được vài giải Olympic tin học.

Nhờ điểm cao, K đậu vào một trường đại học công lập khá danh tiếng. Vào trường, được học thêm về các ngôn ngữ lập trình, về cấu trúc dữ liệu và thuật toán, K càng ngày càng thích code hơn. K code ngày code đêm, cắm đầu vào luyện thuật toán cho thành trùm, lúc rảnh rỗi K lại kiếm sách bài tập để làm… cho đã thèm. K luôn nộp bài sớm hơn các bạn để thể hiện sự hơn người của mình. Do suốt ngày chỉ biết cắm mặt vào máy tính, K trải qua 4 năm đại học mà vẫn FA…

Ra trường và đi làm

Tốt nghiệp ra trường, những tưởng một người lập trình tốt, giỏi kĩ thuật (Như K tự nhận xét mình) sẽ kiếm được việc ngon, lương cao, dễ dàng thăng tiến. K apply vào một công ty nọ. Vòng phỏng vấn diễn ra khá suôn sẻ, K trả lời trơn tru những câu hỏi về kĩ thuật, nhưng lại ngập ngừng ở một số câu hỏi kiểu "Khi có xung đột với thành viên trong nhóm, em xử lý thế nào?" hoặc "Em đã có sản phẩm cá nhân nào nổi bật chưa?".

Đồng nghiệp của K có một thằng tên là H. K rất khinh thằng H này vì nó chỉ tốt nghiệp đại học dân lập (FPT), đã thế trong giờ làm còn ít ngồi code mà toàn vào stackoverflow, pluralsight, webtretho, lâu lâu còn mở wordpress gõ gõ gì đó, hình như là blog. Thế rồi, vì vài chuyện nhỏ nhặt, K dần dần chuyển từ khinh sang ghét và căm thù thằng H.

Chuyện con gà tức nhau tiếng gáy

Chuyện thứ nhất: Công ty có một chương trình khá cũ chạy trên một con server cùi bắp. Gần đây, vì lưu lượng sử dụng nhiều, lâu lâu chương trình lại bị OutOfMemoriesException (tràn bộ nhớ). Tự tin với khả năng code của mình, K xin anh trưởng nhóm source code của chương trình để ngồi optimize lại. Hì hục mất 2 tuần, K chỉ mới optimize được 30% chương trình cho tiết kiệm RAM. Thằng H nghe chuyện liền nhanh nhảu đoảng liên hệ với anh team leader và bên phòng IT, lắp thêm

2 thanh RAM 4GB vào con server cũ. Chỉ mất 5 phút, từ đấy chương trình chạy vo vo không còn OutOfMemories nữa.

Chuyện thứ hai: Team cần làm một phần mềm về phim ảnh, trích xuất dữ liệu từ Galaxy Cinema, Lotte, CGV. Với tinh thần hăng hái ham học hỏi, K tự nghiên cứu về web parser để trích xuất thông tin từ các trang này. Do các trang này có sử dụng Ajax, Paging phức tạp, K mất gần 2 tuần để viết code và test việc parse dữ liệu từ Galaxy và Lotte. Công sức của K tiếp tục đổ sông đổ biển khi ngay hôm sau, thằng H nhanh nhảu nói với anh team leader: "Anh ơi, bên 123phim nó có API sẵn rồi, có cả Cinebox với 4,5 hãng nữa cơ. Phí chỉ có 50k/tháng thôi, mình liên hệ mua rồi tích hợp vào nha anh" và anh team leader gật đầu đồng ý.

Chuyện thứ ba: K thương thầm một em tester da trắng dáng cao ngực khủng tên M cùng công ty. Ngoài làm tester, em còn chuyên đăng hình khoe thân kiêm bán kem trộn trắng da trên facebook. Nghe tiếng K code giỏi, em thủ thỉ "Anh K làm cho em cái trang web bán hàng nha, làm đẹp đẹp em thưởng cho". Vốn dại gái, lại tự tin rằng mình giỏi Java thần thánh, K bắt đầu vẽ database, dùng Spring Boot, Heroku để code một trang web bán hàng hoành tráng.

Hôm sau lên công ty, đang định show giao diện cho em M xem thì K thấy em đang đứng bên cạnh thằng H, mắt long lanh đầy ngưỡng mộ "Anh H giỏi quá, làm web vừa đẹp lại vừa nhanh, em nhờ hôm qua mà giờ đã xong rồi". Hóa ra thằng ku dùng PHP – thứ ngôn ngữ rẻ tiền và chôm đại cái theme có sẵn đâu đó để làm web cho ẻm một cách nhanh chóng. Hết giờ làm, bé M thưởng thằng H bằng cách đi ăn uống, xem phim rùi đi chỗ-mà-ai-cũng-biết-là-chỗ-nào-đấy. Sau hôm đó, K nhìn thằng H bằng ánh mắt mang hình viên đạn.

Chuyện cuối cùng: Cay cú với thằng H, K quyết định viết một blog cá nhân để cạnh tranh với nó. Thấy thằng ku dùng wordpress cùi bắp, K quyết tự khẳng định bản thân bằng cách code luôn một blog engine cho nó biết mặt. Code gần 3 tháng mà vẫn chưa xong, blog thì vẫn chưa viết được dòng nào, trong khi blog kia của thằng H đã được hơn 1 triệu lượt xem.

Những câu chuyện trên 100% là hư cấu. Hãy khoan chửi em M là con bitch ngực to, hay ông team leader dốt không thấy được khả năng của K. Rõ ràng, cùng làm một việc, nhưng H lại hoàn thành một cách nhanh chóng và nhẹ nhàng hơn K. Lý do là vì K tiếp cận vấn đề theo cách của một coder, H lại tiếp cận vấn đề theo cách của một developer.

Coder và Developer

Với coder, code là thứ tối thượng có thể giải quyết được mọi vấn đề. Họ thích tự học những ngôn ngữ/công nghệ mới. Họ là những người tập trung công sức để viết những dòng code thật tốt, chạy thật nhanh, optimize thật kĩ. Họ không thích tái sử

dụng code của người khác, mà thích tự code để "học hỏi và phát triển". Thế nhưng, một trong những sự thật đau lòng của ngành lập trình là đôi khi… khách hàng chẳng ai thèm quan tâm đến code cả.

Khác với coder, developer thường tiếp cận vấn đề theo cách khác. Họ cũng có thể code và code giỏi, nhưng họ tập trung trước nhất vào vấn đề và cách giải quyết. Khi có yêu cầu từ khách hàng, họ sẽ phân tích và tìm cách giải quyết phù hợp, ít tốn thời gian và công sức nhất trước khi bắt tay vào code.

Giỏi kĩ thuật sẽ giúp bạn làm một coder giỏi, nhưng chỉ vậy là không đủ. Bạn cần phải rèn luyện tư duy lập trình, tư duy sản phẩm, tư duy phân tích thiết kế và vô số thứ khác để trở thành một developer đúng nghĩa.

Tóm tắt

- Coder thích tập trung vào code, còn developer tập trung vấn đề và cách giải quyết chúng.
- Hãy luyện cách nhìn nhận vấn đề và suy nghĩ như một Developer.

PHẦN 2 – KĨ NĂNG CƯNG CỨNG

Trong phần 1, mình tập trung nhiều vào kĩ năng mềm. Điều đó không có nghĩa là kĩ năng cứng không quan trọng. Kĩ năng mềm có tốt đến mấy mà kĩ năng cứng không ổn thì bạn cũng khó mà sống lâu trong nghề và đạt được sự nể trọng của anh em đồng nghiệp. Vì lẽ đó, phần này của quyển sách sẽ đưa bạn từ mềm đến cứng, trở nên hoàn thiện.

Vài khái niệm cơ bản trong ngành phần mềm

Phần này giải thích về một số khái niệm quan trọng trong ngành phần mềm như: Scrum/Agile, UI/UX, CI, Technical Debt, comment, design pattern,… Một số khái niệm này đã được dạy ở trường. Tuy nhiên, sách giải thích thêm bằng giọng văn hài hước, dễ hiểu, đồng thời bổ sung thêm cách áp dụng chúng vào thực tế qua các ví dụ cụ thể.

Trung 0335740004

XÓA MÙ VỀ AGILE VÀ SCRUM

Phần mềm không tự nó sinh ra cũng không tự nó nâng cấp, mà phải được phát triển và bảo trì bởi các lập trình viên. Nhiều qui trình được thành lập để giúp viện phát triển phần mềm trở nên dễ dàng và bài bản hơn. Mỗi lập trình viên điều phải hiểu về các qui trình này để có thể làm việc một cách hiệu quả.

Tuy nhiên, mình thấy đa số bạn sinh viên chỉ hiểu mang máng về các qui trình này. Một số bạn đã đi làm nhưng cũng chỉ mù quáng tuân theo qui trình mà không hiểu rõ mục đích cũng như ý nghĩa của nó. Bài viết này sẽ cho bạn cái nhìn rõ ràng hơn về các qui trình phát triển phần mềm, cũng như cách các công ty áp dụng chúng trong thực tế.

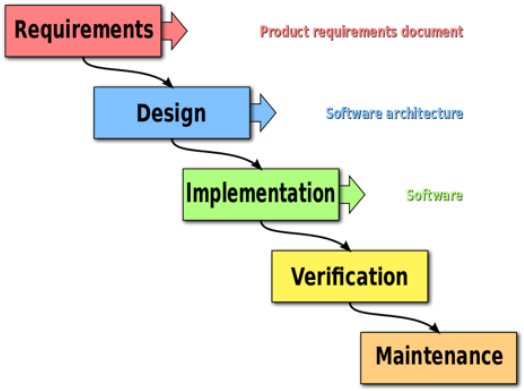
Lưu ý: Bản thân Agile và Scrum là những chủ đề rất rộng, cần đến vài quyển sách và khóa học để nói về chúng. Trong phạm vi bài viết, mình chỉ đưa ra những điểm chính yếu để bạn đọc có nền tảng kiến thức và có thể tự tìm hiểu sâu hơn.

Phần 1 – Xóa mù về Agile

Qui trình phát triển phần mềm

Để hiểu hơn về Agile, trước hết ta hãy xem lại qui trình thường dùng để phát triển phần mềm. Qui trình này thường bao gồm 5 quá trình:

1. Lấy yêu cầu (requirement) từ khách hàng
2. Thiết kế hệ thống
3. Developer phát triển hệ thống
4. Tester kiểm thử hệ thống và để khách hàng xác nhận
5. Bảo trì, sửa chữa hoặc thêm chức năng cho hệ thống



Với mô hình phát triển truyền thống (còn gọi là waterfall), các quá trình này được thực hiện tuần tự từ đầu đến cuối. Tuy nhiên, rất khó để lấy chính xác yêu cầu (requirement) của khách hàng ở giai đoạn đầu, vì họ thường không biết mình cần gì cho đến khi nhìn thấy sản phẩm (phần mềm).

Khi requirement thay đổi, ta phải làm lại các bước thiết kế và phát triển, kiểm thử, viết lại tài liệu… Kết quả là sản phẩm làm ra không đúng yêu cầu của khách hàng, hoặc bị trễ thời gian, quá ngân sách.

Sự ra đời của Agile

Vì lẽ đó, vào tháng 2 năm 2001, 17 nhân vật có tiếng tăm trong ngành phần mềm đã có một cuộc họp để tìm ra phương pháp xây dựng phần mềm tinh gọn, hiệu quả. Thành quả của cuộc họp là một tấm hình tự sướng của vài người đàn ông chổng mông vào màn ảnh và một bản "Tuyên ngôn Agile".

Ảnh gốc của Tuyên Ngôn Agile

Nội dung tuyên ngôn:

Cá nhân và tương tác hơn là **quy trình và công cụ**
Phần mềm chạy tốt hơn là **tài liệu đầy đủ**
Cộng tác với khách hàng hơn là **đàm phán hợp đồng**
Phản hồi với thay đổi hơn là **bám sát kế hoạch**

Mặc dù các điều bên phải vẫn còn giá trị, nhưng chúng tôi đánh giá cao hơn các mục ở bên trái.[35]

Những điểm hay của Agile

[35] Bản dịch từ hocvienagile.com. Bản gốc từ: http://agilemanifesto.org/

Sau khi Agile ra đời, nó nhanh chóng nhận được sự ủng hộ và đón nhận của cộng đồng. Tuân theo những nguyên lý Agile góp phần làm tăng tỷ lệ thành công của dự án. Bản thân Agile cũng mang lại những góc nhìn rất mới mẻ trong việc phát triển phần mềm:

- Xây dựng phần mềm theo kiểu "cuốn chiếu": Với Agile, ta phát triển phần mềm theo các iteration khoảng 2-4 tuần. Trong mỗi iteration, ta thực hiện đầy đủ các qui trình: lấy yêu cầu, thiết kế, code, test,... Sản phẩm được xây dựng dần dần nên khách hàng có thể thấy sản phẩm để đưa ra góp ý hoặc requirement mới.
- Sự tham gia của khách hàng: Trong qui trình waterfall, ta chỉ lấy yêu cầu của khách hàng lúc ban đầu và đưa sản phẩm cho họ sau khi hoàn thành. Với Agile, ta trực tiếp đưa khách hàng tham gia vào việc phát triển phần mềm ở mỗi iteration. Khách hàng trực tiếp tham gia những buổi họp với cả nhóm.
- Tập trung vào sản phẩm, phần mềm chứ không phải tài liệu: Một trong những yếu điểm của waterfall chính là phải viết quá nhiều tài liệu dư thừa trước khi viết code. Mỗi thay đổi từ requirement sẽ dẫn đến việc thay đổi cả tài liệu lẫn code, làm qui trình trở nên chậm chạp và nặng nề.
- Coi trọng sự giao tiếp giữa các thành viên và tầm quan trọng của team: Thay cho các tài liệu khô cứng, việc nói chuyện trực tiếp, mặt đối mặt sẽ giúp trao đổi thông tin hiệu quả hơn
- Bản chất của phần mềm là thay đổi. Tất cả những điều trên giúp góp phần giảm "Cost of Change" (Chi phí khi thay đổi). Với qui trình waterfall, rất khó để khách hàng thay đổi requirement sau khi hệ thống đã được design. Với Agile, ta phát triển dự án theo iteration ngắn nên khách hàng dễ dàng thấy được thành phẩm, đưa ra yêu cầu mới ở mỗi iteration.

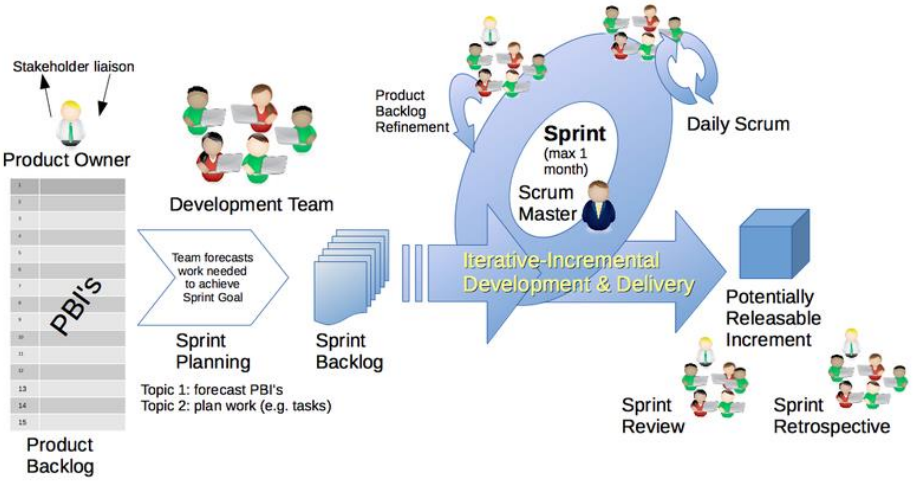
Kết luận

Nhiều bạn thường lầm tưởng Agile và Scrum là một, nhưng thực tế không phải vậy. Agile không phải là một qui trình hoàn thiện mà chỉ là một tập hợp các nguyên lý chung chung mà chúng ta nên tuân theo để phát triển phần mềm một cách uyển chuyển tinh gọn.

Dựa theo những nguyên lý của Agile, người ta đề ra những bộ khung (framework) hoặc qui trình phát triển phần mềm như: Lean, Kanban, XP, Crystal, Scrum. Hiện tại, Scrum là qui trình phổ biến nhất và được áp dụng ở nhiều công ty phần mềm trong và ngoài nước. Cùng tìm hiểu về Scrum trong phần 2 nhé.

Phần 2 – Tìm hiểu và áp dụng Scrum

Trái với Agile, Scrum không phải là những nguyên lý chung chung mà là một bộ khung (framework), với các công cụ (artifact), vai trò (role) và qui trình rõ ràng dựa trên các nguyên lý của Agile.



Hầu hết các bài viết về Scrum hiện nay đều tập trung vào việc giải thích các khái niệm của Scrum. Để bạn đọc dễ hiểu hơn, mình sẽ lấy ví dụ cụ thể về một dự án phần mềm: Công ty A. nọ muốn tạo ra một hệ thống bán sách mang tên Taka.vn, cạnh tranh trực tiếp với Tiki.vn.

Một số khái niệm quan trọng của Scrum

- User Story: Đây là tóm tắt một chức năng của sản phẩm đứng từ góc nhìn của người dùng. User story thường được viết trong một hai câu ngắn gọn, đơn giản, ví dụ như:
    - Là khách hàng mua sách, tôi có thể xem toàn bộ sách có trong cửa hàng
    - Là quản lý, tôi có thể thay đổi số lượng sách trong kho
- Story Point: Là một đơn vị được dùng để đánh giá công sức phải bỏ ra để thực hiện một user story. Story point càng nhiều thì user story đó càng lớn và tốn nhiều thời gian thực hiện.
- Product Backlog: Đây là danh sách những việc mà team cần phải làm để có thể hoàn thành sản phẩm. Mỗi việc này có thể là việc technical (tạo database, tích hợp API) hoặc là user stories.
- Sprint (Iteration): Có thể hiểu là một chu kì làm việc kéo dài từ 2 đến 4 tuần. Trong một sprint, team thực hiện các quá trình như thiết kế, code, test để tạo ra sản phẩm vào cuối sprint.

- Sprint Backlog: Ở đầu mỗi sprint, cả team sẽ có một cuộc họp để lựa chọn những user story quan trọng mà team phải hoàn thành trong sprint này. Những user story này sẽ được bỏ vào Sprint Backlog. Ví dụ ở sprint đầu, team cần một bản demo để người mua sách sử dụng, họ sẽ chọn những user story liên quan đến người mua để thêm vào backlog.
- Definition of Done (Định nghĩa hoàn thành): Trước khi bắt đầu, cả nhóm phải thống nhất thế nào là "hoàn thành" một công việc. Hoàn thành có thể là user stories đó đã được code và test xong, hoặc module đó đã demo cho khách hàng và được khách hàng duyệt. Định nghĩa này tùy thuộc vào mỗi team.

Một số vị trí (role) trong Scrum

- Scrum Master: Đây là người đảm bảo mọi người hiểu và thực hiện đúng qui trình Scrum. Họ cũng có trách nhiệm huấn luyện các thành viên trong nhóm về qui trình, tổ chức các cuộc họp.
- Product Owner: Là người quản lý Product Backlog. Với Taka.vn, product owner là người quyết định những chức năng nào nên được thêm vào, những chức năng nào nên bị bỏ đi, giải thích từng chức năng cho các thành viên trong nhóm làm việc.
- Development Team: Dựa theo các user story trong product backlog, team tạo ra sản phẩm. Trong team không chia ra các vị trí như leader, tester,... mà tất cả đều là developer ngang hàng nhau. Cả team sẽ tự phân chia công việc, tự dự đoán thời gian hoàn thành (trong thực tế làm việc, mình thấy các nhóm Scrum vẫn có vị trí leader, hỗ trợ các thành viên chọn công việc và ước đoán).

Chuyện họp hành

Trong qui trình Scrum, thành viên thường phải tham gia các cuộc họp sau:

- Sprint Planning Meeting: Cuộc họp này được bắt đầu mỗi Sprint, kéo dài khoảng 4 tiếng. Cuộc họp giúp xác định mục tiêu cần đạt được trong Sprint và những công việc (user story) cần làm. Trong cuộc họp này, cả nhóm cũng ước đoán story point cho mỗi user story.
- Daily Meeting: Xuyên suốt Sprint, mỗi ngày cả team sẽ bỏ ra khoảng 15 phút để tổ chức một cuộc họp ngắn. Việc họp này giúp các thành viên nắm được tình hình dự án. Trong cuộc họp, mỗi thành viên trả lời 3 câu hỏi:
    - Hôm qua tôi đã làm được gì?
    - Hôm nay tôi sẽ làm gì?
    - Tôi đang gặp trở ngại, vướng mắc gì?
- Sprint Review Meeting: Cuộc họp này diễn ra sau khi kết thúc Sprint. Cả nhóm sẽ xem xét lại những việc đã hoàn thành, chưa hoàn thành, sau đó

demo những phần đã hoàn thành cho Product Owner. Product Backlog sẽ được cập nhật lại.

- Retrospective Meeting: Trong buổi họp này, cả nhóm cùng ngồi nhìn lại và đánh giá những điểm được và chưa được trong Sprint vừa qua, đồng thời đề xuất các biện pháp cải tiến.

Một Sprint làm việc của team Scrum

Giả sử mình là một thành viên trong Development Team của Taka.vn, một Sprint làm việc của mình sẽ trông như sau:

- Đầu tháng 2, bắt đầu Sprint. Mình tham dự Sprint Planning Meeting. Trong cuộc họp này, anh Product Owner xác nhận mục tiêu là "cho phép người mua xem và bình chọn sách trong hệ thống". Những user story được chọn cho Sprint Backlog là:
    - Là một người mua, tôi muốn xem toàn bộ sách trong hệ thống (3 points)
    - Là một người mua, tôi muốn tìm sách theo tên tác giả (5 points)
    - Là một người mua, tôi muốn bình chọn cho sách (8 points)
    - Là một người mua, tôi muốn xem những sách được bình chọn cao nhất (8 points)
    - ...
- Sau buổi họp, mình và các bạn trong nhóm chọn những user stories sẽ làm. Mỗi sáng, mình tham dự họp Daily Meeting từ 10h tới 10h15. Mình trả lời các câu hỏi (không dài dòng, nếu cần bàn luận chuyên môn hãy tổ chức cuộc họp khác):
    - Hôm qua mình đã dựng xong giao diện cho chức năng bình chọn
    - Hôm nay mình sẽ thiết kế database và bắt đầu viết code cho chức năng này
    - Khó khăn: Database trên máy mình không truy cập được, có thể tốn thời gian sửa
- Giữa tháng 2, kết thúc Sprint, mình tham dự Sprint Review Meeting. Cả team demo chức năng đã làm xong (chọn sách, bình luận). Sau khi xem demo, Product Owner đưa ra nhận xét, cập nhận lại requirement, thêm yêu cầu mới.
- Sau đó, cả nhóm tham gia Retrospective Meeting. Team đã làm tốt việc giao tiếp với nhau, làm đúng yêu cầu khách hàng, đúng hẹn. Tuy vậy, do chưa có chuẩn chung nên mọi người code và thiết kế một kiểu. Cả nhóm đề xuất biện pháp "tạo ra coding và design convention để mọi người tuân theo" nhằm cải tiến qui trình.

Lưu ý: Hầu hết các công ty không tuân theo qui trình Scrum "chuẩn" hoàn toàn mà sẽ biến đổi nó để phù hợp với văn hóa và qui trình của công ty. Thứ quan trọng nhất là hiệu quả công việc, sản phẩm làm ra chứ không phải là "theo đúng qui trình". Ngoài ra, đừng quá lo lắng khi bạn chưa thể hình dung được cách qui trình Scrum hoạt động. Chỉ cần đi làm khoảng 1,2 tuần và tham gia vào một dự án, bạn sẽ hiểu ngay thôi.

Tóm tắt

- Agile và Scrum là hai khái niệm khác nhau.
- Agile không phải là một qui trình hoàn thiện mà chỉ là một tập hợp các nguyên lý chung để phát triển phần mềm một cách uyển chuyển tinh gọn.
- Trái với Agile, Scrum là một bộ khung (framework) với các công cụ (artifact), vai trò (role) và qui trình rõ ràng dựa trên các nguyên lý của Agile
- Đa phần các công ty không tuân theo hoàn toàn qui trình Scrum mà sẽ tùy biến cho phù hợp

TỔNG QUAN VỀ UI/UX TRONG NGÀNH LẬP TRÌNH

Gần đây, thuật ngữ UX đang nhận được khá nhiều sự quan tâm. Bài viết sẽ giải thích tổng quát về UI và UX cùng vai trò của chúng trong ngành lập trình.

Hai khái niệm này thường hay dễ bị nhầm lẫn với nhau. UI không phải là UX mặc dù chúng có quan hệ rất gần gũi. Một phần cũng do bản thân trong ngành IT, nhiều công ty cũng "gom nhóm" hai khái niệm này lại (Gọi chung là UI/UX Designer) nên gây ra sự lẫn lộn trên.

Lưu ý: UI và UX đã là đối tượng nghiên cứu từ cách đây vài chục năm. Để nói về UI hay UX cũng cần đến vài quyển sách giáo khoa. Bài viết chỉ giải thích các khái niệm chính yếu và tầm quan trọng của chúng cho các bạn đọc có một nền tảng cơ bản để tìm hiểu thêm.

Định nghĩa về UI và UX thông qua... phụ nữ

Thay vì đưa khái niệm "chuẩn" về UI và UX trong sách giáo khoa, mình sẽ giải thích ngắn gọn hai khái niệm này. Đây là cách giải thích được phần lớn cộng đồng chấp nhận.

- UI (User Interface) – Giao diện người dùng: Đây là những gì người dùng nhìn thấy và giúp họ tương tác với hệ thống (giao diện Web, button, ...).
- UX (User Experience) – Trải nghiệm người dùng: Đây là những gì người dùng trải nghiệm khi sử dụng sản phẩm và dịch vụ (Bao gồm cảm xúc, suy nghĩ trong quá trình sử dụng). UI chỉ là một phần của UX.

Để dễ hiểu, hãy tưởng tượng trong công ty bạn có một em gái tên M[36]. Em M da trắng trẻo, gầm cao máy thoáng, ba vòng đầy đặn, mặt mũi dễ thương. Nhan sắc và ngoại hình của em M chính là UI. Có thể thấy em M là một sản phẩm có UI rất tốt.

Hãy tiếp tục tưởng tượng, sau một thời gian tán tỉnh, bạn đã cưa được em M. Sau khi quen nhau, bạn mới biết em rất hay mè nheo, nhõng nhẽo, nửa đêm đòi bạn mua trà sữa qua cho uống. Em cũng rất ít khi quan tâm bạn mà chỉ biết đòi quà, lại hay đi rắc thính lung tung. Bạn cảm thấy bực mình, buồn, khó chịu. Những trải nghiệm khó chịu này chính là UX. Có thể thấy UX của bé M không tốt chút nào.

Quan hệ giữa UI và UX

Chỉ có UI mà ko có UX sẽ làm cho sản phẩm trông rất đẹp và bắt mắt nhưng không dùng được (Như bé M ở trên). Có UX mà ko có UI sẽ tạo ra một ứng dụng ổn, được

[36] Đã xuất hiện trong bài "Đôi khi code là cách ngu nhất..."

việc nhưng… trông thấy gớm (Giống như mấy bạn nữ tốt, ngoan hiền nhưng "mặt tiền" xấu thường khó kiếm người yêu).

UI và UX không bao giờ đứng riêng rẽ mà luôn song hành với nhau để tạo nên sản phẩm hoàn chỉnh. Một trang web với font chữ cẩu thả, màu sắc lòe loẹt, button nhỏ xíu (UI tồi) thì khó mà mang lại UX tốt cho người dùng được.

Bản thân UX không chỉ gói gọn trong sản phẩm (phần mềm) mà còn liên quan tới nhiều khía cạnh như: dịch vụ khách hàng, giá trị thương hiệu. Ví dụ như UX của sản phẩm Apple rất tốt vì ngoài sản phẩm tốt, họ còn chăm sóc khách hàng tận tình, thương hiệu "sang chảnh". Trong phạm vi bài viết chúng ta thảo luận về UX trong ngành lập trình, tức là UX của sản phẩm.

Tầm quan trọng của UI và UX

Ngày xưa, hầu như các chương trình máy tính đều chạy trên màn hình Console, người dùng phải gõ lệnh vào để sử dụng. Ngày nay thì khác, các ứng dụng có mặt ở khắp nơi, từ web, mobile cho tới các thiết bị điện tử.

Các ứng dụng cạnh tranh với nhau rất gay gắt (có đến vài chục ứng dụng to-do-list, vài chục ứng dụng nghe nhạc, vài chục ứng dụng quản lý tiền bạc….). Ứng dụng có UI đẹp và chuyên nghiệp thì sẽ thu hút được người dùng. Ứng dụng có UX tốt thì sẽ chiếm được tình cảm của người dùng, được họ sử dụng thường xuyên và giới thiệu cho bạn bè.

Người dùng ngày một trở nên "lười" và khó tính. Dù sản phẩm bạn làm ra có tốt đến mấy, nếu trông nó "xấu xấu", người dùng sẽ chẳng thèm sử dụng. Nếu UX thiết kế tồi, quá rườm rà khó sử dụng, người dùng sẽ bỏ cuộc ngay sau 5 phút dùng thử. Có thể nói chính UI và UX góp phần quyết định sự thành bại của một sản phẩm.

Developer thì cần gì phải biết UI và UX?

Tất nhiên, ở các công ty lớn thì họ có sự tách bạch rất rõ ràng cho các vị trí UX Designer, UI Designer và Developer. Chúng ta chỉ việc nhận thiết kế từ tay designer rồi code. Mặc dù vậy, trong các công ty startup hoặc các công ty vừa và nhỏ, đôi khi developer chỉ nhận được yêu cầu từ khách hàng và phải tự tay thực hiện toàn bộ các khâu từ thiết kế UX, vẽ UI cho tới code.

Kiến thức về UI/UX đều hữu dụng trong cả hai trường hợp trên. Ở các công ty lớn, biết về UI/UX sẽ giúp bạn dễ trao đổi với designer và hiểu điều họ muốn hoàn thành. Ở công ty vừa và nhỏ, biết về UI và UX sẽ giúp bạn tạo ra chức năng tốt hơn, phù hợp với người dùng hơn.

Tóm tắt

- UI và UX là hai khái niệm hoàn toàn khác nhau
- Tuy vậy, UI và UX không bao giờ đứng riêng rẽ mà luôn song hành với nhau để tạo nên sản phẩm hoàn chỉnh
- Người dùng ngày một trở nên "lười" và khó tính, sản phẩm vượt trội phải có UI đẹp và UX tốt
- Kiến thức về UI và UX sẽ giúp lập trình viên làm việc tốt hơn

MỘT BUTTON TRỊ GIÁ 300 TRIỆU ĐÔ – CÁI NHÌN KHÁC VỀ GIAO DIỆN VÀ CHỨC NĂNG

Ngày xửa ngày xưa, có một trang web bán hàng...

Đây là một chuyện nho nhỏ, về một button nho nhỏ và một số tiền... không nhỏ chút nào. Mình đọc được chuyện này được trong cuốn Don't make me think – một cuốn sách khá hay về UI/UX[37].

Ngày xửa ngày xưa, ở một đất nước nọ, có một trang web bán hàng... Chức năng cơ bản của một trang web bán hàng thì ai cũng biết: hiển thị hàng, cho hàng vào giỏ và và thanh toán. Câu chuyện của chúng ta bắt đầu ở chức năng "Thanh toán", khi người dùng đã cho hết hàng vào giỏ, một form nho nhỏ xinh xinh sẽ hiện ra với 2 trường Tên Đăng Nhập và Mật Khẩu, 2 button Đăng Nhập và Đăng Ký. Thế nhưng, chính cái form be bé xinh xinh này đã gây thiệt hại đến 300.000.000$/năm cho trang web bán hàng.

Vấn đề ở chỗ, người dùng phải đăng nhập hoặc đăng ký trước khi muốn thanh toán. Đội lập trình nghĩ rằng "Chỉ cần đăng ký tài khoản lần đầu, thông tin người dùng có thể được lưu lại, ở những lần sau họ không cần nhập địa chỉ và thông tin thanh toán nữa. Người dùng tiết kiệm được thời gian, cũng khuyến khích được người dùng quay lại mua hàng, hai bên cùng có lợi còn gì?".

Lập trình viên đôi khi nghĩ rằng mình hiểu người dùng

Đội ngũ designer đã làm một cuộc thử nghiệm, đưa tiền cho người dùng để họ thực hiện quá trình mua hàng và thanh toán. Đối với những khách hàng mới của trang web, họ phát hiện ra một điều: người dùng rất ghét việc đăng ký, với suy nghĩ "Mình muốn mua hàng, chứ không phải muốn đăng ký đăng kiếc gì cả". Chưa kể, người dùng còn sợ bị mất thông tin cá nhân hoặc bị gửi mail spam vào hộp thư.

Với những người dùng quay lại lần thứ hai, thứ ba – đối tượng mà các developer nhắm tới, tình cảnh cũng chẳng khá khẩm hơn. Họ không nhớ được mật khẩu của mình. Mặc dù chức năng "Quên mật khẩu" vẫn hoạt động, đến tận 160.000 người dùng chức năng này mỗi ngày, 75% trong số đó không tiếp tục quá trình thanh toán sau khi đã yêu cầu mật khẩu.

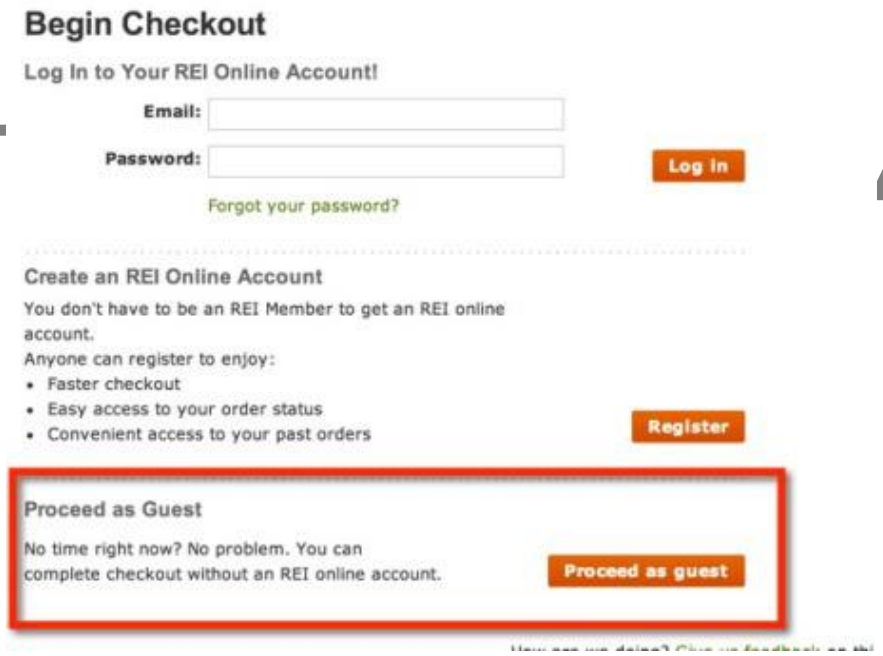
[37] Đã giới thiệu trong bài "Lập trình viên thì nên đọc sách gì?"

Chiếc form nho nhỏ xinh xinh kia, hóa ra lại là thứ ngăn cản người dùng mua hàng, rất nhiều người dùng. Thế mới biết, developer nhiều khi nghĩ mình hiểu được người dùng, nhưng thật ra không phải vậy.

Button trị giá 300 triệu đô la

Đội ngũ designer đã giải quyết vấn đề này một cách vô cùng đơn giản. Họ bỏ đi nút Đăng Ký, thay vào đó bằng nút Tiếp Tục và dòng chữ "Bạn không cần đăng ký, hãy bấm nút Tiếp Tục để thanh toán. Để thanh toán nhanh hơn ở những lần sau, bạn có thể đăng ký một tài khoản vào lúc thanh toán".

Kết quả: Lượng thanh toán của khách hàng tăng lên đến 45%. Sau tháng đầu tiên, doanh số tăng 15.000.000$. Sau năm đầu tiên, thu nhập của web tăng đến tận 300 triệu đô la. Tất cả những việc mà họ đã làm là gì? Chỉ là thêm một button mà thôi.



Ảnh minh họa

Một cái nhìn khác về UI/UX và chức năng

Xin nhắc lại một chút về UI và UX[38]:

[38] Xem lại bài viết "Tổng quát về UI và UX"

- UI (User Interface) – Giao diện người dùng: Đây là những gì người dùng nhìn thấy và giúp họ tương tác với hệ thống (Form, button, website).
- UX (User Experience) – Trải nghiệm người dùng: Đây là những gì người dùng trải nghiệm, bao gồm cảm xúc, suy nghĩ, quá trình. UI chỉ là một phần của UX. Ví dụ như khi bạn mua hàng trên tiki. Các trang web thanh toán, web mua hàng chính là UI. Quá trình mua hàng và nhận hàng, sự thoải mái khi thanh toán, sự hỗ trợ của Tiki dành cho khách hàng chính là UX.

Trong câu chuyện trên, bằng cách thay đổi UI (Thêm một button) và không bắt đăng kí, họ đã trực tiếp cải thiện UX (Thay đổi trải nghiệm mua hàng, mua nhanh hơn mà không cần đăng ký), từ đó làm tăng doanh số bán hàng.

Hẳn nhiều bạn lập trình viên vẫn còn suy nghĩ: Chức năng mới là quan trọng, còn mấy thứ ruồi bu như giao diện thì làm thế nào cũng được (Một phần chắc là do lúc mới học lập trình toàn phải viết chương trình Console). Đây là một suy nghĩ hoàn toàn sai lầm.

Nếu chỉ viết vài tool đơn giản cho chính mình hoặc bạn bè sử dụng thì đúng là giao diện không quan trọng. Nhưng nếu đã tạo ra sản phẩm hướng tới người dùng, thì UI/UX là những thứ quan trọng nhất nhì, có khi còn hơn cả chức năng. Hãy nhớ một điều mình muốn nói qua câu chuyện này: Đôi khi chỉ một button nho nhỏ lại có giá trị đến 300 triệu đô đấy.

Tóm tắt

- Lập trình viên thường nghĩ rằng mình hiểu người dùng, nhưng đôi khi không phải vậy
- UI/UX có tầm quan trọng không kém gì chức năng của một chương trình.Hãy nhớ, đôi khi một button nho nhỏ cũng có thể có giá đến 300 triệu đô

LUẬN VỀ TECHNICAL DEBT – NỢ KIẾP NÀY, DUYÊN KIẾP TRƯỚC

Technical Debt (Nợ kĩ thuật) là một món nợ mà hầu như lập trình viên nào cũng phải gánh trong quá trình làm việc. Hẳn bạn sẽ thắc mắc: Hầu hết lập trình viên chúng ta đều là những con người siêng năng chăm chỉ, không cờ bạc gái gú, hết giờ làm là đi nhậu rồi mát xa … à không, về nhà với vợ con. Chúng ta không vay mượn ai bao giờ thì làm sao có nợ???

Muốn biết câu trả lời, hãy đọc kĩ bài viết này để tìm hiểu thêm về Technical Debt nhé! Đây là một khái niệm khá quan trọng đấy.

Technical Debt là gì?

Khái niệm này được đưa ra bởi Ward Cunningham (Cha đẻ của wiki đầu tiên). Trong cuộc sống, đôi khi bạn sẽ phải mượn tiền để xài trước, sau đó cày cuốc trả. Số tiền này được gọi là nợ. Trong lập trình cùng thế, đôi khi ta chọn cách giải quyết "mì ăn liền", giải quyết được vấn đề ngay, nhưng sẽ gây khó khăn cho quá trình phát triển và bảo trì về sau. Mỗi lần như vậy, ta tạo thêm một khoản "nợ kĩ thuật" cho dự án.

Technical Debt ban đầu rất ít, nhưng theo quá trình code thì càng ngày nó càng nhiều lên, trở thành nợ nần chồng chất. Một số ví dụ:

- Để tái sử dụng code đã viết, ta copy và paste code và sửa đôi chút (thay vì phải tách thành module riêng). Cách làm này nhanh, nhưng khi có bug thì sửa… hộc máu vì code được copy ở nhiều chỗ.
- Khi có requirement mới, thay với thiết kế code cho dễ mở rộng, ta viết thêm hàm if. Cách này nhanh, nhưng nếu mở rộng nhiều thì code sẽ có một đống if.
- Có bug khủng liên quan tới kiến trúc hệ thống, thay vì fix bug và refactor thì ta try/catch nuốt lỗi và fix tạm ở phần ngọn, gọi là hotfix.

Technical Debt là điều tất yếu trong quá trình code. Mỗi quyết định ta đưa ra trong lúc code đều làm tăng số nợ này lên. Điều quan trọng là mượn xong thì phải trả, nếu để lâu, techinical debt tích luỹ sẽ gây ra nhiều hậu quả nguy hiểm khôn lường.

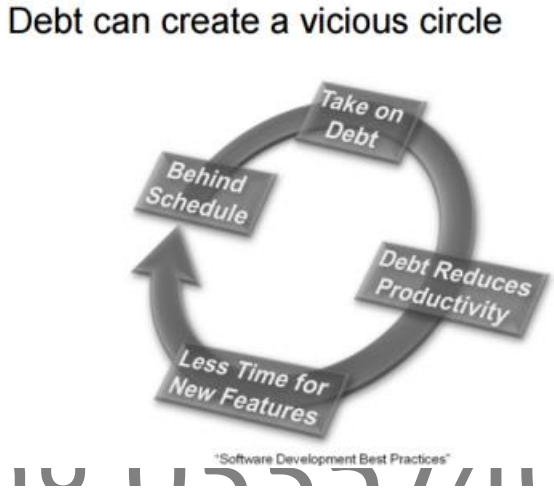
Tác hại "khủng khiếp" của nợ

Nếu không trả nợ, cả vốn lẫn lãi sẽ dần chồng chất trong quá trình phát triển. Quá nhiều technical debt làm chậm tốc độ của team, đồng thời ảnh hưởng đến tinh thần làm việc của các thành viên trong nhóm.

Trong nhiều dự án, vì ban đầu bị trễ deadline nên team phải code ẩu, sinh ra technical debt. Nợ làm cho tốc độ phát triển chậm dần lại, dẫn tới trễ dealine -> code

ẩu -> thêm nợ, tạo thành một vòng lẩn quẩn. Một tính năng bình thường có thể chỉ mất 1 ngày để hoàn thành, nhưng nếu technical debt quá nhiều sẽ mất tới 1 tuần.

Tới một mức nào đó, khi không trả được lãi nữa, ta sẽ bị "phá sản". Lúc này, code hiện tại đã nát tới mức cực kì khó mở rộng hay bảo trì, phải đập đi viết lại. Đây cũng là nguyên nhân gây trễ deadline và dẫn đến thất bại cho nhiều dự án.



Vòng tròn lẩn quẩn: trễ deadline -> nợ -> code chậm -> trễ deadline

Nợ ơi em từ đâu tới?

Nếu như nợ công của Việt Nam là do các bác "ở trển" dùng vốn không đúng cách thì nợ kĩ thuật (technical debt) lại do chính bản thân các developer gây ra.

Có rất nhiều lý do gây ra technical debt:

- Do khách hàng thay đổi requirement liên tục, kiến trúc dự án không kịp thay đổi cho phù hợp
- Do bị deadline dí hoặc manager gây áp lực nên developer code ẩu để hoàn thành công việc
- Do bản thân developer làm biếng nên code không có comment và không viết tài liệu
- Do team không có technical lead giỏi, hoặc các thành viên không có nền tảng kĩ thuật tốt

Đôi khi technical debt là do cố ý: Chấp nhận làm nhanh để có sản phẩm giao khách hàng, giành dự án, vấn đề technical tính sau. Hoặc trong các công ty startup, người ta xây dựng sản phẩm khả thi tối thiểu (MVP) nhanh nhất có thể để khảo sát nhu cầu

người dùng. Lúc này, chức năng và tốc độ phát triển mới là quan trọng nhất, code ẩu hay kiến trúc tệ cũng không quan trọng.

Làm sao trả nợ?

Như mình đã nói, code nào cũng sẽ có bug, dự án nào cũng sẽ có technical debt. Cách đối phó với technical debt là tạm ngưng việc phát triển và tập trung vào trả nợ. Ta có thể trả nợ bằng cách phân tích và tái cấu trúc hệ thống hoặc viết thêm tài liệu, viết thêm test case, refactor code để code rõ ràng hơn, dễ cải tiến hơn.

Đôi lúc ta cũng có thể bỏ qua technical debt, ví dụ như khi làm prototype để demo cho khách hàng. Vì prototype xong rồi vứt luôn nên ta có thể xù nợ. Tuy nhiên nên cẩn thận, có rất nhiều trường hợp khách hàng đòi mở rộng hoặc nâng cấp prototype thành sản phẩm để… tiết kiệm thời gian. Lúc này ta phải vắt chân lên cổ mà trả nợ luôn!

Prototype bằng giấy cho đỡ tốn công code

Hãy nhớ một điều: Mỗi lần bạn code ẩu, code ẩu, bạn đang thêm nợ cho dự án. Nợ đời có vay có trả, bạn không trả thì người khác trong team sẽ trả. Technical debt phải trả bằng thời gian, công sức và mồ hôi nước mắt của lập trình viên đấy nhé.

À, nếu sắp nghỉ việc, chuyển công ty thì các bạn cứ code ẩu thoải mái, không sao đâu! Một lập trình viên "xấu số" nào khác sẽ trả nợ giùm bạn.

Tóm tắt

- Technical Debt là những món nợ về kĩ thuật, ta "mượn nợ" để phát triển nhanh hơn, nhưng sẽ ảnh hưởng đến tốc độ phát triển về sau
- Nếu không trả nợ kịp thời, nợ ngày càng chồng chất sẽ làm chậm tiến độ dự án hoặc "phá sản"
- Có nhiều nguyên nhân gây ra nợ: do khách hàng, do developer code ẩu, do qui trình không tốt

- Để trả nợ, có thể viết thêm tài liệu, unit test, refactor hoặc tái cấu trúc hệ thống

GIẢI THÍCH ĐƠN GIẢN VỀ CI – CONTINUOUS INTEGRATION (TÍCH HỢP LIÊN TỤC)

Với các bạn sinh viên, khái niệm Continuous Integration (Tích hợp liên tục) là một cái gì đó nghe rất cao siêu và hoành tráng. Mình sẽ nêu khái niệm trước, sau đó đưa ra một câu chuyện đơn giản để giải thích cho khái niệm này.

> Tích hợp liên tục (CI) là phương pháp phát triển phần mềm đòi hỏi các thành viên trong nhóm tích hợp công việc thường xuyên. Mỗi ngày, các thành viên đều phải theo dõi và phát triển công việc của họ ít nhất một lần. Việc này sẽ được một nhóm khác kiểm tra tự động, nhóm này sẽ tiến hành kiểm thử truy hồi (regression test) để phát hiện lỗi nhanh nhất có thể. Cả nhóm thấy rằng phương pháp tiếp cận này giúp giảm bớt vấn đề về tích hợp hơn và cho phép phát triển phần mềm gắn kết nhanh hơn.

Tr )4

Để dễ hiểu, các bạn hãy đọc câu chuyện nho nhỏ phía dưới

Chuyện ngày xưa của Nam

Ngày xửa ngày xưa, Nam còn là sinh viên ngành IT của trường B. Mỗi lần code và làm bài tập nhóm đối với Nam là một cực hình. Cả team ngồi phác thảo ra từng module nho nhỏ, sau đó chia ra, mạnh ai về nhà code phần của người đó. Cuối tuần, cả nhóm hẹn nhau ra quán cafe để "ráp code", copy các phần đã làm qua USB, bỏ vào một project chung. Mỗi lần "ráp code", chương trình thường không build được hoặc một

núi lỗi xuất hiện, cả nhóm phải hì hục mất nguyên buổi chiều để sửa lỗi. "Ráp code" trở thành một cơn ác mộng đối với Nam và các bạn trong nhóm.

Ra trường rồi đi làm, Nam đi phỏng vấn xin việc, được nhận vào công ty phần mềm C. Ở đây, mọi người đã biết dùng SVN nên không còn cảnh phải copy code qua lại nữa. Từ "ráp code" cũng biến mất mà thay vào đó là từ "tích hợp" (integration). Tuy nhiên, team của Nam vẫn còn làm việc khá ẩu tả. Mỗi sáng, các thành viên trong team lấy code từ SVN/Git về, code say sưa, sau đó commit code lên trước khi về nhà.

Đôi khi code không build được, cả team lại nháo nhào "truy tìm thủ phạm": anh Phạm Văn A sửa code mà quên commit file mới lên, chị Lê Thị B sửa connection string, … Đời nhiều khi còn trái ngang hơn, anh A sửa code, chị B sửa code làm phần của Nam chạy bị lỗi, thế là Nam lãnh đủ. Mỗi cuối tuần, cả team lại phải làm thêm giờ để tích hợp và sửa lỗi.

Phần mềm không bán được vì quá nhiều lỗi, phát hành chậm, công ty cũ phá sản, Nam phải đi tìm việc mới.

Ích lợi của CI

Với khả năng của mình, Nam được nhận vào công ty D. Công ty này khá chuyên nghiệp, có áp dụng CI – tích hợp liên tục. Nhờ có CI, team của Nam hiện tại không còn gặp những rắc rối khi tích hợp:

- Mỗi khi có người commit code, hệ thống CI sẽ tự lấy code từ SVN, thực hiện việc build. Hệ thống sẽ gửi mail thông báo cho toàn bộ thành viên nếu như build bị lỗi. Cả nhóm chỉ việc đọc mail, xem ai là người commit revision đó, và "nắm đầu" thủ phạm ra, bắt hắn sửa lỗi.
- Project của Nam được viết Unit Test rất cẩn thận, đầy đủ. Khi anh A, chị B sửa code, trước khi commit lên, hệ thống sẽ build và chạy lại toàn bộ các Unit Test. Nếu có case nào bị fail, anh A, chị B phải phải sửa code để các case này pass mới được commit.
- Việc tích hợp và build code được diễn ra hàng ngày, nhiều lần trong ngày. Mỗi khi ai đó commit code làm hư build hoặc gây lỗi, cả team có thể giải quyết vấn đề ngay lập tức. "Tích hợp" không còn là nỗi ác mộng mà trở thành chuyện thường ngày đối với Nam.

Kết

Hiện tại, hầu như các công ty phần mềm đều áp dụng CI thông qua một số công nghệ như: TFS, TeamCity, Hudson, Jenkin, Travis, … Hãy tự trang bị cho mình những kiến thức về CI để có giá hơn trong mắt nhà tuyển dụng nhé.

Tóm tắt

- Continous Integration – tích hợp liên tục là phương pháp mà các team Agile sử dụng để đảm bảo code của toàn dự án luôn build được, luôn chạy đúng.
- Nhờ có CI, phần mềm có thể được tích hợp và phát triển nhanh hơn.
- Một số công nghệ hỗ trợ CI: TFS, TeamCity, Hudson, Jenkin,…

SỰ KHÁC BIỆT GIỮA WEB SITE VÀ WEB APPLICATION

Hiện nay một số bạn học ngành IT vẫn còn lẫn lộn về sự khác nhau giữa website và web app. Đây là một câu hỏi "tưởng dễ mà không phải dễ", bởi vì ranh giới giữa website và webapp khá mong manh. Mình phải tổng hợp khá nhiều câu trả lời từ Stackoverflow và Stack Exchange mới đưa ra được một câu trả lời "gần đúng" nhất.

Khái niệm Web Site (Trang Web)

Ngày xưa ngày xưa, khi Internet còn thô sơ, web được viết bằng HTML đơn lẻ. Mỗi trang web đơn lẻ viết bằng HTML được gọi là Web Page. Tập hợp nhiều trang web đơn lẻ, thành một trang web lớn, có chung tên miền, được gọi là Web Site. Ví dụ đơn giản: Mỗi bài viết trên blog của mình chính là một Web Page, tập hợp toàn bộ các bài viết lại chính là một website, tên là toidicodedao.com.

Khái niệm Web Application (Ứng dụng Web)

Đầu tiên, ta hãy xem lại khái niệm application - ứng dụng (trên wiki).

> Ứng dụng là một loại chương trình có khả năng làm cho máy tính thực hiện trực tiếp một công việc nào đó mà người dùng muốn thực hiện.

Ban đầu, các website chỉ bao gồm chữ, hình ảnh và video, liên kết với nhau thông qua các đường link. Tác dụng của website là lưu trữ và hiển thị thông tin. Người dùng chỉ có thể đọc, xem, click các link để di chuyển giữa các page.

Về sau, với sự ra đời của các ngôn ngữ server side: CGI, Perl, PHP, … các website đã trở nên "linh động" hơn, có thể tương tác với người dùng. Từ đây, người dùng có thể dùng web để "thực hiện một công việc nào đó bằng máy tính", do đó web app ra đời.

Nói dễ hiểu, web app là những ứng dụng chạy trên nền web. Thông qua web app, người dùng có thể thực hiện một số công việc: tính toán, chia sẻ hình ảnh, mua sắm … Tính tương tác của web app cao hơn website rất nhiều.

Với một số người không rành về IT, tất cả những thứ online, truy cập được bằng trình duyệt đều là website cả. Do đó họ thường yêu cầu bạn làm một website quản lý siêu thị, website bán hàng, … thực chất chúng đều là web app hết.

So sánh Web Site và Web App

Trên thực tế, ranh giới giữa web app và website khá mong manh. Một trang báo mạng – vnexpress chẳng hạn, trong mắt người đọc nó là website. Nhưng trong mắt biên tập viên hoặc admin, nó lại là web app. Một số trang web cho phép người dùng search, comment nhưng nó vẫn chỉ là website, chưa phải là web app.

Tóm tắt

Web Site	**Web Application**
Tính tương tác thấp, ít chức năng (Xem, đọc, click qua lại giữa các link…)	Tính tương tác cao, nhiều chức năng (Đăng thông tin, upload file, xuất báo cáo…)
Được tạo thành từ các trang html tĩnh và một số tài nguyên (hình ảnh, âm thanh, video)	Được tạo bởi HTML/CSS ở front-end và code ở back-end (PHP, C#, Java, …)[39]
Được dùng để lưu trữ, hiển thị thông tin	Được dùng để "thực hiện một công việc", thực hiện các chức năng của một ứng dụng

[39] Xem lại bài "Kĩ năng cần có của Web Developer"

LUẬN VỀ COMMENT CODE (PHONG CÁCH KIẾM HIỆP)

Comment code luôn là vấn đề gây tranh cãi sứt đầu mẻ trán trong giới võ lâm. Xưa kia, thuở còn mài đít trên ghê nhà trường, ta thường được các thầy dặn rằng: Code nhớ phải comment. Thuở mới đi làm, sơ nhập giang hồ, mỗi khi đọc code không hiểu, ta cũng hay đập bàn mà chửi: "Thằng này code ngu quá đ*o comment gì cả".

Thế nhưng, lưu lạc giang hồ bao năm, đọc code cũng nhiều; từ code không comment cho tới code comment đầy đủ và sạch sẽ; ta vẫn băn khoăn không biết thật sự phải comment thế nào mới đúng. Thế rồi, sau khi đọc Clean Code, ta như lượm được bí tịch võ công trong truyền thuyết.

Ta ngộ ra được cái gọi là "đạo code", "đạo comment", đặt một bước chân vào con đường võ đạo đỉnh cao (Đạo ở đây là đạo lý, ko phải đạo nhạc như Sơn Tùng, các đạo hữu đừng hiểu lầm).

Lấy chuyện kiếm hiệp nói chuyện code

Thần Điêu Đại Hiệp của Kim Dung có đoạn: Dương Quá bị lạc vào kiếm mộ của Độc Cô Cầu Bại. Thuở còn sống, Độc Cô cầu bại là người kiếm thuật vô song, danh chấn thiên hạ. Trước khi chết, ông chôn 4 thanh kiếm ở nơi gọi là kiếm mộ.

Tại hạ muốn quý vị chú ý chi tiết này:

> Một hồi sau, chàng mới đặt thanh kiếm nặng đó xuống, nhấc thanh kiếm thứ ba lên, lần này chàng lại bị lầm. Chàng cứ tưởng thanh kiếm này phải nặng hơn thanh kiếm vừa rồi, nên vận lực ra cánh tay. Nào ngờ nó nhẹ tênh như không, chàng ngưng thần xem kỹ, hóa ra đó là một thanh kiếm gỗ, chôn dưới đá lâu năm, thân và cán kiếm đều đã bị mục, đọc dưới mặt đá có khắc dòng chữ:
>
> Sau bốn mươi tuổi, không mang binh khí.
>
> Thảo mộc trúc thạch đều có thể dùng làm kiếm.
>
> Cứ thế tinh tu, đạt tới cảnh giới vô kiếm thắng hữu kiếm.

Tổng kết lại, cuộc đời tu kiếm của Độc Cô Cầu Bại bao gồm ba giai đoạn:

- Lúc trai trẻ, lòng đầy nhiệt huyết mà thiếu sự chín chắn thì sử dụng tử vi kiếm là thanh bảo kiếm sắc, nhẹ và linh hoạt.
- Khi đạt độ chín của suy nghĩ và sức lực, sử dụng thanh kiếm sắt nặng nề mà không sắc bén.

- Khi bắt đầu về già, suy nghĩ và kiếm thuật đạt trình độ cao, vũ khí chỉ còn là thanh kiếm gỗ và đạt mức thượng thừa thì không kiếm mà thắng đối thủ, bất cứ thứ gì cũng là kiếm.

Ta tự thấy hầu như lập trình viên nào cũng sẽ phải trải qua ba giai đoạn này trên con đường "cầu đạo". Kẻ nào ngộ tính cao thì chỉ cần làm việc một hai năm đã đạt tới cảnh giới cuối. Kẻ nào đầu óc u mê lạc lối thì cả đời vẫn cứ ở giai đoạn hai mà thôi.

Giai đoạn trẻ trâu đầy nhiệt huyết (Code không comment hoặc comment lung tung)

Kẻ nào sơ nhập giang hồ đều trải qua giai đoạn này. Đây là khi ta còn ngồi trên ghế nhà trường hoặc vừa mới đi làm. Code chạy được là mừng, đôi khi code cho xong là nộp, chả cần comment.

Nhiều khi ta cũng nghe lời các sư phụ, thêm comment vào code. Đống comment này nhiều nhưng khá vô dụng, đọc rất buồn cười, như thể mấy kẻ mới học võ công mà bày trò múa máy bắt chước các chiêu thức cao siêu trong võ học.

```
//Chia đôi toàn bộ các phần tử của mảng đưa vào
public int[] half(int[] y){
  //Tạo một array mới chứa kết quả
  int[] x;
  //Duyệt các phần tử của mảng y
  for (int i = 0; i < y.length ; i++)
  {
     x[i] = y[i]/2; //Gán trá trị vào array chứa kết quả
  };
  return x; //Trả kết quả ra
}
```

Ta thấy đoạn code trên tuy comment khá đầy đủ nhưng chẳng có ý nghĩa mấy, thời gian viết comment còn nhiều hơn viết code. Một số coder thường dùng trò: Thêm comment để biết ngày giờ sửa code, comment lại một số code không dùng nữa. Một số lỗi khác mà các đạo hữu này hay mắc phải như:

- Comment lại code cũ để "đề phòng". Code không dùng nữa thì xóa đi, đừng comment. Có SVN rồi thì khi cần code cũ chỉ việc restore lại là xong.
- Comment để làm log. Đừng comment theo kiểu: //02/03/1992 Sửa tên class. Sử dụng SVN có thể tra ra log rõ ràng và tiện dụng hơn nhiều.

Giai đoạn tạm chín chắn trong suy nghĩ (Biết cách comment)

Code một thời gian, hầu như mọi lập trình viên sẽ đạt đến giai đoạn này. Ở giai đoạn này, ta đã cảm thụ được một chân lý võ học : Comment nên là what, không phải how. Comment để nói code làm gì, không phải để giải thích việc code làm.

Lúc này, lập trình viên đã hiểu rõ hơn về cách comment. Với những đoạn code phức tạp, dài dòng, 1 dòng comment ngắn gọn sẽ giúp người sau dễ dàng hiểu đoạn code làm gì:

```
public object DoThing() {
  //Lấy 1 phần tử random từ database
  //Một đống code phức tạp
  return obj;
}
```

Tuy vậy, các bậc cao nhân xưa đã rút ra được một điều: Comment là thứ dối trá. Khi sửa code, comment thường không được sửa, sẽ dẫn tới tình trạng code một đằng, comment một nẻo. Ta phải đọc cả code lẫn comment để biết code làm gì, phí gấp đôi thời gian. Cách comment tốt nhất chính là dùng code, chính code sẽ nói code làm gì. Ngộ ra được điều này, các đạo hữu sẽ đạt tới cảnh giới cuối cùng trong "code học".

Cảnh giới vô kiếm thắng hữu kiếm (Vô comment thắng hữu comment)

Đây là cảnh giới mà ta hy vọng các đạo hữu có thể đạt được. Ở cảnh giới này, ta code không cần comment nhiều. Giống như Độc Cô Cầu Bại có thể lấy kiếm gỗ làm kiếm, lấy lá cỏ làm kiếm, lấy tay làm kiếm; ta thấy bất cứ thứ gì cũng có thể làm comment: tên biến cũng là comment, tên hàm cũng là comment, tên database cũng là comment. Code cũng như comment, comment cũng như code, code và comment tuy hai mà một.

Tuy nhiên, đôi khi bắt buộc phải dùng comment: Khi viết JavaDoc, API v...v, ta bắt buộc phải comment và document, vì người dùng API chỉ được đọc document chứ không đọc code. Một số trường hợp khác: comment phải là why chứ không phải what. Ta viết comment để người khác (hoặc chính ta sau này) biết vì sao ta lại viết code như vậy. Comment có tác dụng nói những điều bản thân code không nói được. Còn việc code làm gì, chạy như thế nào, chỉ cần đọc code là hiểu.

```
// Chỉ cần nhìn tên hàm là biết hàm làm gì
public object GetRandomObjectFromDatabase() {
  return randomObj;
}

public int[] HalfAllNumbersFromInput(int[] input) {
   int[] output; //Nhìn tên biến là biết biến làm gì
   for (int i = 0; i < input.length ; i++)
   {
      output[i] = input[i]/2;
      // Viết để dubug, sau này phải remove
      // Đây là comment bắt buộc phải viết,
      // để giải thích lý do ta viết code
      Console.Write("Call this");
   };
   return output;
}
```

Ở đây, ta không phủ nhận độ hữu dụng của comment. Nhưng vấn đề là: Nhiều gã coder lợi dụng comment để viết code không rõ ràng, khó hiểu, sau đó sử dụng comment để lấp liếm. Điều ta muốn các bằng hữu hiểu là: Hãy có gắng viết code một có rõ ràng nhất có thể, bằng cách đặt tên biến, tên hàm, tách code,… Đó mới là cảnh giới tối cao của "code đạo". Đừng học đòi theo lũ trẻ trâu mà viết code kiểu "càng ngắn càng tốt", càng khó hiểu càng tốt, đó chỉ là cái dũng của kẻ thất phu mà thôi. Chúc các bằng hữu sớm thành cao thủ trên con đường cầu đạo, nhầm, cầu code của mình.

Tóm tắt

- Đừng nghĩ rằng comment càng nhiều càng tốt, mà hãy viết code sao cho dễ đọc, dễ hiểu, không cần comment.
- Comment nên là what (nói code làm gì) mà nên là why (Giải thích tại sao lại phải thiết kế, viết code như vậy).

NHẬP MÔN DESIGN PATTERN (PHONG CÁCH KIẾM HIỆP)

Nhập đề

Kinh thư ghi lại rằng, con đường tu chân có 3 cảnh giới: Luyện khí, Trúc cơ và Kết đan. Luyện khí là quá trình rèn thân luyện thể, cho phàm thân kiên cường dẻo dai. Trúc cơ là quá trình du nhập thiên địa linh khí vào thể nội, giúp khai thông kinh mạch. Khi thiên địa linh khí trong đan điền đạt tới một nồng độ nhất định, sẽ kết thành Kim Đan, đặt bước chân đầu tiên con đường tu chân đại đạo.

Con đường khởi đầu của code học cũng có 3 cảnh giới: Học đồ (Junior Developer), Học sĩ (Developer), Đại sư (Senior Developer). Để đạt đến cảnh giới Đại sư (senior), bất kì Học Sĩ (dev) nào cũng cần phải tường tận vài Design Pattern cơ bản để phòng thân. Bài viết này do tại hạ viết ra trong một phút cao hứng nhất thời, nhằm chia sẻ với các nhân sĩ võ lâm trên con đường truy cầu đại đạo.

Nhiều kẻ khi đạt đến cảnh giới Đại sư (Senior) cứ ngỡ rằng mình đã đạt đến cảnh giới tối cao của võ học mà không biết rằng "Thiên ngoại hữu thiên, nhân ngoại hữu nhân". Phía trên cảnh giới Đại sư còn có vô số cao thủ đạt tới những cảnh giới khác như Chưởng Môn (Project Manager) hoặc Tông sư (Software Architect). Những kẻ này hiếm thấy như phượng mao lân giác (lông phượng sừng lân), mang một thân võ công cao ngất ngưởng và lương cũng cao ngất ngưỡng, lướt gió mà đi, đạp mây mà về. Do cảnh giới bản thân còn thấp, bần đạo tạm thời không bàn tới[40].

Hỏi thế gian DS là chi, mà bọn Dev thề nguyền sống chết

Nói một cách đơn giản, design pattern là các mẫu thiết kế có sẵn, dung để giải quyết một vấn đề. Áp dụng mẫu thiết kế này sẽ làm code dễ bảo trì, mở rộng hơn nhưng có thể sẽ khó hiểu hơn một chút.

Nói văn hoa, design pattern là tinh hoa trong võ học, đã được các bậc tông sư đúc kết, truyền lưu từ đời này qua đời khác. Design pattern là thiết kế dựa trên code, nó nằm ở một cảnh giới cao hơn code, do đó đệ tử của bất kì môn phái nào (C#, Java, Python) cũng có thể áp dụng vào được.

Trước khi dạy võ, các bậc đạo sư luôn dặn học trò rằng học võ là để tu thân hành hiệp giúp đời, không phải để ý vào một thân võ học mà đi bắt nạt kẻ yếu. Nay ta cũng có một lời khuyên tương tự: Học design pattern là để nâng cao trình độ, để giải quyết vấn đề, không phải đế lấy ra lòe thiên hạ.

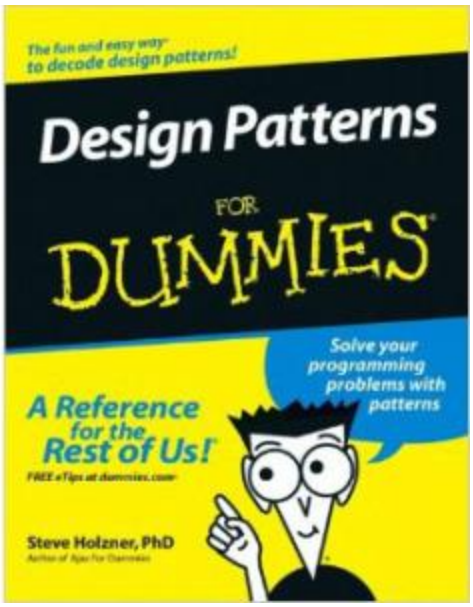
---

[40] Bằng hữu nào hứng thú xin đọc lại bài "Con đường phát triển nghề nghiệp của developer"

Nhiều kẻ học nghệ chưa tinh, ngựa non háu đá, nhét design pattern vào dự án một cách vô tội vạ, nhẹ thì tẩu hỏa nhập ma, võ công sụt giảm, nặng thì hồn phi phách tán, vĩnh kiếp không được siêu sinh. Các đạo hữu hãy nhìn kẻ than tàn ma dại phía dưới mà làm gương.

Design Pattern Kiếm Phổ

Bí kíp võ công đầu tiên về design pattern được giang hồ biết đến là là Design Patterns: Elements of Reusable Object-Oriented Software. Tuy nhiên, khẩu quyết trong quyển này khá khô cứng, khó truyền dạy, do đó các bậc cao nhân đã chỉnh sửa, xuất bản 2 cuốn bí kíp dễ hiểu hơn cho hậu thế là Head First Design Patterns và Design Patterns For Dummies. Thuở xưa khi đặt bước chân đầu tiên trên con đường cầu đạo-cạo đầu, bần đạo cũng tự tu luyện từ hai cuốn bí kiếp này. Các đạo hữu có thể lên mạng tải về ngâm cứu.



Khẩu quyết nhập môn Design Pattern

Có khá nhiều chiêu thức design pattern lưu lạc trong chốn giang hồ, song ta có thể tạm phân loại làm Tam Thức:

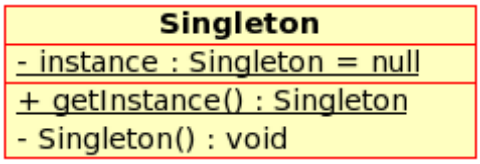
- **Khởi Thức (Creational Design Pattern):** Liên quan đến việc khởi tạo object. VD: Factory, Object Pool, Abstract Factory, Builder.
- **Cấu Thức (Structure Design Pattern):** Liên quan đến kết cấu, liên hệ giữa các object. VD: Adapter, Bridge, Decorator, Proxy, Composite, Facede.
- **Vi Thức (Behavioral Design Pattern):** Liên quan tới hành vi của các object. VD: Iterator, Mementor, Strategy, Template Method, Visitor.

Khẩu quyết một chiêu thức Design Pattern thường có 3 phần. Khi muốn học một design pattern mới, hãy tập trung chú ý vào 3 phần này:

- **Thức Đề:** Vấn đề mà design pattern đó giải quyết
- **Thức Đồ:** Sơ đồ UML mô tả design pattern
- **Thức Phổ:** Code minh họa

Dưới đây là một Design pattern đơn giản nhất mà hầu như học sĩ nào cũng biết: Đơn Thân Độc Mã, thuộc Khởi Thức, hay còn gọi là Singeton, thuộc loại Creational Design Pattern.

- **Thức Đề:** Design Pattern này được dùng khi ta muốn đảm bảo chỉ có duy nhất một object được sinh ra trong toàn bộ hệ thống
- **Thức Đồ:**



- **Thức Phổ:**

```
public class Singleton {
    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```


Thay lời kết

Xin nhắc lại một lần nữa: Design pattern được tạo ra để giải quyết vấn đề, chứ không phải để phức tạp hóa nó. Các bậc cao nhân có câu: nước có thể dâng thuyền, cũng có thể lật thuyền. Design pattern có thể giải quyết vấn đề, cũng có thể làm nó rắc rối phức tạp hơn.

Kẻ sĩ dùng design pattern cũng chia làm ba cảnh giới. Kẻ sơ nhập thì nhìn đâu cũng thấy pattern, chỉ lo áp dụng, nhét rất nhiều pattern vào mà không quan tâm đến thiết kế. Lăn lộn giang hồ một thời gian, đến cảnh giới cao thủ, sẽ học được rằng khi nào cần dùng pattern, khi nào không. Đến cấp bậc đại sư, chỉ dùng pattern khi đã rõ lợi hại của nó, biết lấy sự đơn giản hài hòa của design tổng thể làm trọng. Có thể tổng kết quá trình này bằng một câu:

> Khi chưa học đạo, ta thấy núi là núi, sông là sông. Khi mới học đạo, ta thấy núi không phải là núi, sông không phải là sông. Sau khi học đạo, ta lại thấy núi chỉ là núi, sông chỉ là sông.

Tóm tắt

- Design pattern là những mẫu thiết kế có sẵn, dùng để giải quyết một vấn đề nào đó.

- Áp dụng design pattern vào code sẽ giúp nó dễ bảo trì và mở rộng hơn, nhưng cũng có thể làm tăng tính phức tạp của thiết kế.
- Đừng cố gắng áp dụng design pattern một cách vô tội vạ, càng nhiều càng tốt. Hãy giữ cho thiết kế đơn giản hết mức có thể mà vẫn giải quyết được vấn đề

Các khái niệm và kĩ thuật nâng cao

Phần này đề cập đến những khái niệm phức tạp hoặc ít được nói đến trong sách vở như: Nguyên lý SOLID, Dependency Injection, leaked abstraction … Bên cạnh đó là những mánh khóe, kĩ năng code, debug và những lời khuyên để nâng cao khả năng kĩ thuật. Do sử dụng nhiều thuật ngữ, khái niệm chuyên ngành nên phần này sẽ hơi khó hiểu với các bạn không thuộc ngành IT.

SOLID LÀ GÌ – ÁP DỤNG CÁC NGUYÊN LÝ SOLID ĐỂ TRỞ THÀNH LẬP TRÌNH VIÊN CODE "CỨNG"

Trong quá trình học, hầu như các bạn sinh viên đều được học một số khái niệm OOP cơ bản như sau:

- Abstraction (Tính trừu tượng)
- Encapsulation (Tính bao đóng)
- Inheritance (Tính kế thừa)
- Polymophirsm (Tính đa hình)

Những khái niệm này đã được dạy khá rõ ràng và hầu như những buổi phỏng vấn nào cũng có những câu hỏi liên quan đến chúng. Vì 4 khái niệm này khá cơ bản, bạn nào chưa vũng có thể Google để ôn lại thêm.

Giới thiệu chung

Trái ngược với 4 khái niệm nói trên, những nguyên lý được giới thiệu trong bài viết này là những nguyên lý thiết kế trong OOP. Đây là những nguyên lý được đúc kết bởi máu xương vô số developer, rút ra từ hàng ngàn dự án thành công và thất bại. Một project áp dụng những nguyên lý này sẽ có code dễ đọc, dễ test, rõ ràng hơn.

Chúng còn giúp việc bảo trì code và sửa chữa dễ hơn rất nhiều (Ai có kinh nghiệm trong ngành IT đều biết thời gian code chỉ chiếm 20-40%, còn lại là thời gian để bảo trì: thêm bớt chức năng và sửa lỗi). Nắm vững những nguyên lý này, đồng thời áp dụng chúng trong việc thiết kế và viết code sẽ giúp bạn tiến thêm một bước trên con đường thành senior (Một anh senior bên FPT Software từng dạy mình thế).

Ok, 10 phút quảng cáo đã qua, bây giờ đến phần giới thiệu. SOLID tức là "cứng", áp dụng nhiều thì bạn sẽ code "cứng". Đùa thôi, nó là tập hợp 5 nguyên tắc sau đây:

- **S**ingle responsibility principle
- **O**pen/closed principle
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

Trong phạm vi bài viết, mình chỉ giới thiệu tổng quát để các bạn có cái nhìn tổng quan về các nguyên lý này. Mỗi nguyên lý sẽ được giới thiệu kĩ hơn ở các bài viết sau.

1. Single Responsibility Principle

Nguyên lý đầu tiên, tương ứng với chữ **S** trong SOLID. Nội dung nguyên lý:

> Một class chỉ nên giữ một trách nhiệm duy nhất (Chỉ có thể thay đổi class vì một lý do duy nhất)

Để hiểu nguyên lý này, ta hãy lấy ví dụ với 1 class vi phạm nguyên lý. Ta có 1 class *ReportManager* như sau:

```
public class ReportManager()
{
   public void ReadDataFromDB();
   public void ProcessData();
   public void PrintReport();
}
```

Class này giữ tới 3 trách nhiệm: Đọc dữ liệu từ database, xử lý dữ liệu, in kết quả. Do đó, chỉ cần ta thay đổi database hoặc thay đổi cách xuất kết quả, ... ta sẽ phải sửa đổi class này. Càng về sau class sẽ càng phình to ra.

Theo đúng nguyên lý, ta phải tách class này ra làm 3 class riêng. Tuy số lượng class nhiều hơn những việc sửa chữa sẽ đơn giản hơn, class ngắn hơn nên cũng ít bug hơn.

2. Open/Closed Principle

Nguyên lý thứ hai, tương ứng với chữ **O** trong SOLID. Nội dung nguyên lý:

> Có thể thoải mái mở rộng 1 class, nhưng không được sửa đổi bên trong class đó (open for extension but closed for modification).

Theo nguyên lý này, mỗi khi ta muốn thêm chức năng,.. cho chương trình, chúng ta nên viết class mới mở rộng class cũ (bằng cách kế thừa hoặc sở hữu class cũ) không nên sửa đổi class cũ.

3. Liskov Substitution Principle

Nguyên lý thứ ba, tương ứng với chữ **L** trong SOLID. Nội dung nguyên lý:

> Trong một chương trình, các object của class con có thể thay thế class cha mà không làm thay đổi tính đúng đắn của chương trình

Hơi khó hiểu nhỉ? Không sao, lúc mới đọc mình cũng vậy. Hãy tưởng tượng bạn có 1 class cha tên *Vịt*. Các class *VịtBầu, VịtXiêm* có thể kế thừa class này, chương trình chạy bình thường. Tuy nhiên nếu ta viết class *VịtChạyPin*, cần pin mới chạy được. Khi class này kế thừa class Vịt, vì không có pin không chạy được, sẽ gây lỗi. Đó là 1 trường hợp vi phạm nguyên lý này.

4. Interface Segregation Principle

Nguyên lý thứ tư, tương ứng với chữ **I** trong SOLID. Nội dung nguyên lý:

> Thay vì dùng một interface lớn, ta nên tách thành nhiều interface nhỏ, với nhiều mục đích cụ thể

Nguyên lý này khá dễ hiểu. Hãy tưởng tượng chúng ta có 1 interface lớn với khoảng 100 methods. Việc implement sẽ khá cực khổ, ngoài ra còn có thể dư thừa vì 1 class không cần dùng hết 100 method. Khi tách interface ra thành nhiều interface nhỏ, gồm các method liên quan tới nhau, việc implement và quản lý sẽ dễ hơn.

5. Dependency Inversion Principle

Nguyên lý cuối cùng, tương ứng với chữ **D** trong SOLID. Nội dung nguyên lý:

> 1. Các module cấp cao không nên phụ thuộc vào các module cấp thấp. Cả 2 nên phụ thuộc vào abstraction.
>
> 2. Interface (abstraction) không nên phụ thuộc vào chi tiết, mà ngược lại. (Các class giao tiếp với nhau thông qua interface, không phải thông qua implementation.)

Nguyên lý này khá lắt léo, mình sẽ lấy ví dụ thực tế. Chúng ta đều biết 2 loại đèn: đèn tròn và đèn huỳnh quang. Chúng cùng có đuôi tròn, do đó ta có thể thay thế đèn tròn bằng đèn huỳnh quang cho nhau một cách dễ dàng.

Ở đây, interface chính là đuôi tròn, implementation là bóng đèn tròn và bóng đèn huỳnh quang. Ta có thể chuyển đổi dễ dàng giữa 2 loại bóng đèn vì ổ điện chỉ quan tâm tới interface (đuôi tròn), không quan tâm tới implementation.

Trong code cũng vậy, khi áp dụng Dependency Inversion, ta chỉ cần quan tâm tới interface. Để kết nối tới database, ta chỉ cần gọi hàm Get, Save … của Interface *IDataAccess*. Khi thay đổi database, ta chỉ cần thay implementation của interface này.

Mình nói kĩ về nguyên lí này vì nó khá quan trọng. Nó là tiền đề để các bạn tìm hiểu Dependency Injection và Inversion of Control, hai khái niệm khó hiểu nhưng được dùng rất phổ biến trong các framework hiện đại.

SINGLE RESPONSIBILITY PRINCIPLE – NGUYÊN LÝ ĐƠN TRÁCH NHIỆM

Nội dung nguyên lý

> Một class chỉ nên giữ một trách nhiệm duy nhất (Chỉ có thể thay đổi class vì một lý do duy nhất)

Giải thích nguyên lý

Ta có thể tạm hiểu "trách nhiệm" ở đây tương đương với "chức năng". Tại sao một class chỉ nên giữ một chức năng duy nhất?? Để hiểu điều này, hãy nhìn vào hình dưới.

Hãy xem con dao trong hình như một class với rất nhiều chức năng. Con dao này "có vẻ" khá là tiện dụng, nhưng lại cồng kềnh và nặng nề. Đặc biệt, khi có một bộ phận bị hư hỏng, ta phải tháo nguyên con dao ra để sửa. Việc sửa chữa và cải tiến rất phức tạp và có thể ảnh hưởng tới nhiều bộ phận khác nhau.

Một class có quá nhiều chức năng cũng sẽ trở nên cồng kềnh và phức tạp. Trong ngành IT, requirement rất hay thay đổi, dẫn tới sự thay đổi code. Nếu một class có quá nhiều chức năng, quá cồng kềnh, việc thay đổi code sẽ rất khó khăn, mất nhiều thời gian, còn dễ gây ảnh hưởng tới các module đang hoạt động khác.

Áp dụng SRP vào con dao phía trên, ta có thể tách nó ra làm kéo, dao, mở nút chai,... riêng biệt là xong, cái gì hư chỉ cần sửa cái đấy. Với code cũng vậy, ta chỉ cần thiết kế

các module sao cho đơn giản, mỗi module chỉ có một chức năng duy nhất là xong (Nói vậy chứ việc xác định, gom nhóm chức năng không hề dễ đâu nhé).

Ví dụ minh họa

Đoạn code dưới đây là ví dụ cho việc vi phạm SRP. Lỗi này hồi mới học code mình cũng hay mắc phải. Class *Student* có quá nhiều chức năng: chứa thông tin học sinh, format hiển thị thông tin, lưu trữ thông tin.

```
public class Student {
  public string Name { get; set;}
  public int Age { get; set;}

  // Format class này dưới dạng text, html, json để in ra
  public string GetStudentInfoText() {
    return "Name: " + Name + ". Age: " + Age;
  }
  public string GetStudentInfoHTML() {
    return "<span>" + Name + " " + Age + "</span>";
  }
  public string GetStudentInfoJson() {
    return Json.Serialize(this);
  }
  // Lưu trữ xuống database, xuống file
  public void SaveToDatabase() {
    dbContext.Save(this);
  }
  public void SaveToFile() {
    Files.Save(this, "fileName.txt");
  }
}
```

Code như thế thì có làm sao không? Hiện tại thì không sao cả, nhưng khi code lớn dần, thêm chức năng nhiều hơn, class *Student* sẽ bị phình to ra. Chưa kể, nếu như có thêm các class khác như *Person*, *Teacher* v...v, đoạn code hiển thị và lưu trữ thông tin sẽ nằm rải rác ở nhiều class, rất khó sửa chữa và nâng cấp.

Để giải quyết, ta chỉ cần tách ra làm nhiều class, mỗi class có một chức năng riêng là xong. Khi cần nâng cấp, sửa chữa, sẽ diễn ra ở từng class, không ảnh hưởng tới các class còn lại.

```
// Student bây giờ chỉ chứa thông tin
public class Student {
  public string Name { get; set;}
  public int Age { get; set;}
}

// Class này chỉ format thông tin hiển thị student
public class Formatter {
  public string FormatStudentText(Student std) {
    return "Name: " + std.Name + ". Age: " + std.Age;
  }
  public string FormatStudentHtml(Student std) {
    return "<span>" + std.Name + " " + std.Age + "</span>";
  }
  public string FormatStudentJson(Student std) {
    return Json.Serialize(std);
  }
}

// Class này chỉ lo việc lưu trữ
public class Store {
  public void SaveToDatabase(Student std) {
    dbContext.Save(std);
  }
  public void SaveToFile(Student std) {
    Files.Save(std, "fileName.txt");
  }
}
```

Không phải lúc nào cũng nên áp dụng nguyên lý này vào code. Một ngoại lệ hay gặp là các class dạng *Helper* hay *Utilities* – các class này vi phạm SRP một cách trắng trợn. Nếu số lượng hàm ít, ta vẫn có thể cho tất cả các hàm này vào 1 class, xét cho cùng, toàn bộ các hàm trong *Helper* đều có chức năng xử lý các tác vụ nho nhỏ.

```
// Class Helper vi phạm SRP
// Nhưng vì nhỏ, ta có thể bỏ qua
public class Helper {
  public string GetUser();
  public DateTime GetTime();
  public string GetCurrentLocation();
  public DbConnection GetDatabaseConnection();
}
```

Tuy nhiên, khi Helper có thêm nhiều chức năng, nó trở nên phức tạp và cồng kềnh hơn (Bạn mình từng gặp trường hợp một class có tới gần 10000 dòng code). Lúc này, ta cần áp dụng SRP để chia nó thành các module nho nhỏ để dễ quản lý.

```
// Helper đã phức tạp, ta cần tách ra
public class Helper {
  public string GetUser();
  //.....
  public DateTime GetTime();
  //.....
  public string GetCurrentLocation();
  //.....
  public DbConnection GetDatabaseConnection();
}

// Tách helper thành các helper nhỏ hơn
public class UserHelper {
}
public class TimeLocationHelper {
}
public class DatabaseHelper {
}
```


Lưu ý và kết luận

Về bản chất, nguyên lý chỉ là nguyên lý, nó chỉ là hướng dẫn chứ không phải là quy tắc tuyệt đối bất di bất dịch. Nếu tìm hiểu kĩ, các bạn sẽ thấy vẫn có vài lập trình viên mổ xẻ, phản đối, chỉ ra những chỗ chưa ổn của các nguyên lý này. Tuy vậy, việc hiểu rõ chúng vẫn giúp code ta viết ra dễ đọc, dễ hiểu, dễ quản lý hơn.

OPEN/CLOSED PRINCIPLE – NGUYÊN LÝ ĐÓNG/MỞ

Nội dung nguyên lý

> Chiều dài váy con gái nên đủ ngắn để khơi **MỞ** tính tò mò của con trai, nhưng nên đủ dài để **ĐÓNG** lại những suy nghĩ đen tối của bọn nó.

À xin lỗi, mình đùa đấy. Nội dung nguyên lý đây:

> Có thể thoải mái mở rộng 1 module, nhưng hạn chế sửa đổi bên trong module đó (open for extension but closed for modification).

Giải thích nguyên lý

Theo nguyên lý này, một module cần đáp ứng 2 điều kiện sau:

> **Dễ mở rộng**: Có thể dễ dàng nâng cấp, mở rộng, thêm tính năng mới cho một module khi có yêu cầu.
>
> **Khó sửa đổi**: Hạn chế hoặc cấm việc sửa đổi source code của module sẵn có.

Hai điều kiện này thoạt nghe có vẻ mâu thuẫn quá nhỉ? Theo lẽ thường, khi muốn thêm chức năng thì ta phải viết thêm code hoặc sửa code đã có. Đằng này, nguyên lý lại hạn chế việc sửa đổi source code!! Vậy làm thế nào để ta có thể thiết kế một module dễ mở rộng, nhưng lại khó sửa đổi?

Trước khi nói về lập trình, hãy cùng phân tích một vật dụng được thiết kế chuẩn theo Nguyên lý Đóng Mở: khẩu súng. Để bắn được xa hơn, ta có thể gắn thêm ống ngắm; để không gây tiếng động, ta có thể gắn nòng giảm thanh; để tăng số lượng đạn, ta có thể gắn thêm băng đạn phụ; khi cần cận chiến ta có thể gắn lưỡi lê vào luôn.

Dễ thấy, khẩu súng được thiết kế để ta dễ dàng mở rộng tính năng mà không cần phải mổ xẻ sửa chữa các bộ phận bên trong (source code) của nó. Một module phù hợp OCP cũng nên được thiết kế như vậy.

Ví dụ minh họa

Ta hãy cùng đọc code trong ví dụ dưới đây. Ta có 3 class là *Square*, *Circle*, *Rectangle*. Class *AreaDisplay* tính diện tích các hình này và in ra. Theo code cũ, để tính diện tích, ta cần dùng hàm if để check và ép kiểu object đưa vào, sau đó bắt đầu tính.

```
public class Shape
{
}
public class Square : Shape
{
  public double Height { get; set; }
}
public class Circle : Shape
{
  public double Radius { get; set; }
}
public class Triangle : Shape
{
  public double FirstSide { get; set; }
  public double SecondSide { get; set; }
  public double ThirdSide { get; set; }
}
// Module in ra diện tích các hình
public class AreaDisplay
{
  public double ShowArea(List<Shape> shapes)
  {
    foreach (var shape in shapes) {
      // Nếu yêu cầu thay đổi, thêm shape khác, ta phải sửa module Area
Calculator
      if (shape is Square) {
        Square square = (Square)shape;
        var area += Math.Sqrt(square.Height);
        Console.WriteLine(area);
      }
      if (shape is Triangle) {
        Triangle triangle = (Triangle)shape;
        double TotalHalf = (triangle.FirstSide + triangle.SecondSide +
triangle.ThirdSide) / 2;
        var area += Math.Sqrt(TotalHalf * (TotalHalf - triangle.FirstSide) *
        (TotalHalf - triangle.SecondSide) * (TotalHalf -
triangle.ThirdSide));
        Console.WriteLine(area);
      }
      if (shape is Circle) {
        Circle circle = (Circle)shape;
        var area += circle.Radius * circle.Radius * Math.PI;
        Console.WriteLine(area);
      }
    }
  }
}
```

Dễ thấy nếu trong tương lai ta thêm nhiều class nữa, ta phải sửa class *AreaDisplay*, viết thêm chừng đó hàm if nữa. Sau khi chỉnh sửa, ta phải compile và deploy lại class *AreaDisplay*, dài dòng và dễ lỗi phải không nào??

Áp dụng nguyên lý Đóng/Mở, ta sẽ cải tiến lại như sau:

```
// Ta có 3 class: vuông, tròn, tam giác, kế thừa class Shape
// Chuyển logic tính diện tích vào mỗi class
public abstract class Shape
{
  public double Area();
}
public class Square : Shape
{
  public double Height { get; set; }
  public double Area() {
    return Math.Sqrt(this.Height);
  }
}
public class Circle : Shape
{
  public double Radius { get; set; }
  public double Area() {
    return this.Radius * this.Radius * Math.PI;
  }
}
public class Triangle : Shape
{
  public double FirstSide { get; set; }
  public double SecondSide { get; set; }
  public double ThirdSide { get; set; }
  public double Area() {
    double TotalHalf = (this.FirstSide + this.SecondSide + this.ThirdSide) / 2;
    var area += Math.Sqrt(TotalHalf * (TotalHalf - this.FirstSide) *
        (TotalHalf - this.SecondSide) * (TotalHalf - this.ThirdSide));
    return area;
  }
}

// Module in ra diện tích các hình
public class AreaDisplay
{
  public double ShowArea(List<Shape> shapes)
  {
    foreach (var shape in shapes) {
      Console.WriteLine(shape.Area());
    }
  }
}
```

Ta chuyển module tính diện tích vào mỗi class. Class *AreaDisplay* chỉ việc in ra. Trong tương lai, khi thêm class mới, ta chỉ việc cho class này kế thừa class *Shape* ban đầu. Class *AreaDisplay* có thể in ra diện tích của các class thêm vào mà không cần sửa gì tới source code của nó cả.

Lưu ý và kết luận

Nguyên lý đóng/mở xuất hiện khắp mọi ngóc ngách của ngành lập trình. Một ứng dụng của nguyên lý này là hệ thống plug-in (cho Eclipse, Visual Studio, add-on Chrome). Để thêm các tính năng mới cho phần mềm, ta chỉ việc cài đặt các plug-in này, không cần can thiệp gì đến source code sẵn có.

Nguyên lý này cũng được áp dụng chặt chẽ khi viết các thư viện/framework. Ví dụ, khi sử dụng MVC, ta không được xem hay chỉnh sửa code của các class Router, Controller có sẵn, nhưng có thể viết các Controller mới, Router mới để thêm tính năng. Hoặc khi sử dụng IonicFramework, ta có thể tải thêm plug-in để sử dụng chức năng chụp hình, quét barcode mà không cần động vào source code của Ionic.

Khi áp dụng nguyên lý này trong thiết kế, ta cần phải xác định được những thứ cần thay đổi, để thiết kế phù hợp với thay đổi đó. Đây là một việc rất rất khó, kể cả với các developer lâu năm. Nó đòi hỏi nhiều kinh nghiệm, tầm nhìn và một ít khả năng dự đoán. Còn nếu như lỡ đoán sai những điều cần thay đổi thì sao? Cứ viết code chạy được trước đã rồi refactor dần dần thôi.

LISKOV SUBSTITUTION PRINCIPLE – NGUYÊN LÝ THAY THẾ LISKOV

Nội dung nguyên lý

Trong một chương trình, các object của class con có thể thay thế class cha mà không làm thay đổi tính đúng đắn của chương trình

Giải thích nguyên lý

Xin cảnh báo trước một chút là nguyên lý này hơi trừu tượng và khó hiểu (các bác developer nước ngoài cũng tranh cãi khá nhiều về nó), do đó mình sẽ cố gắng giải thích một cách đơn giản nhất có thể. Nếu đọc lần đầu không hiểu, các bạn cố gắng đọc kĩ lại vài lần nhé, nếu vẫn không hiểu thì… chịu khó Google tìm hiểu thêm vậy.

Để giữ tính đúng đắn của chương trình, class con phải thay thế được class cha. Nói dễ hiểu là thế này: Ngày xửa ngày xưa, hẳn bạn nào cũng có 1 cái *Individual Portable Brick Game*, tên tiếng Việt là máy chơi game xếp hình "thần thánh". Thuở ấy, mỗi lần máy hết pin, mình lại xin mấy nghìn ra ngoài hàng mua pin con ó gắn vào chơi tiếp.

Một lần nọ, hết pin mà không có tiền, mình đi lượm mấy cục pin con heo gắn vào chơi tạm. Không ngờ gắn pin vào xong, vừa hí hửng bật máy lên thì máy bị cháy vì điện thế của pin hơn bình thường. Thế là đi tong luôn cái máy chỉ vì… tiếc tiền mua pin.

Theo lý thuyết, class *PinConHeo* là con của class Pin, khi ta dùng *PinConHeo* gắn vào làm Pin thì máy phải chạy bình thường. Tuy nhiên trong trường hợp của mình, class PinConHeo đã vi phạm LSP vì đã gây lỗi khi dùng thay cho class Pin.

Code minh họa

Mình sẽ đưa ra 2 ví dụ thường gặp về việc vi phạm LSP:

1. Class con throw exception khi gọi hàm

Giả sử, ta muốn viết một chương trình để mô tả các loài chim bay. Đại bàng, chim sẻ, vịt bay được, nhưng chim cánh cụt không bay được. Do chim cánh cụt cũng là chim, ta cho nó kế thừa class *Bird*. Tuy nhiên, vì cánh cụt không biết bay, khi gọi hàm bay của chim cánh cụt, ta sẽ throw *NoFlyException*.

```
public class Bird {
  public virtual void Fly() { Console.Write("Fly"); }
}
public class Eagle : Bird {
  public override void Fly() { Console.Write("Eagle Fly"); }
}
public class Duck : Bird {
  public override void Fly() { Console.Write("Duck Fly"); }
}
public class Penguin : Bird {
  public override void Fly() { throw new NoFlyException(); }
}

var birds = new List { new Bird(), new Eagle(),
                       new Duck(), new Penguin() };
foreach(var bird in birds) bird.Fly();
// Tới pengiun thì lỗi vì cánh cụt quăng Exception
```

Ta tạo 1 mảng chứa các loài chim rồi duyệt các phần tử. Khi gọi hàm *Fly* của class *Penguin*, hàm này sẽ quăng lỗi. Class *Penguin* gây lỗi khi chạy, không thay thế được class cha của nó là *Bird*, do đó nó đã vi phạm LSP.

2. Class con thay đổi hành vi class cha

Đây là ví dụ kinh điển về hình vuông và hình chữ nhật mà mọi người thường dùng để giải thích LSP, mình chỉ viết và giải thích lại đôi chút.

Đầu tiên, hãy cùng đọc đoạn code dưới đây. Ta có 2 class cho hình vuông và hình chữ nhật. Ai cũng biết hình vuông là hình chữ nhật có 2 cạnh bằng nhau, do đó ta có thể cho class *Square* kế thừa class *Rectangle* để tái sử dụng code.

```
public class Rectangle
{
    public int Height { get; set; }
    public int Width { get; set; }

    public virtual void SetHeight(int height)
    {
        this.Height = height;
    }
    public virtual void SetWidth(int width)
    {
        this.Width = width;
    }
    public virtual int CalculateArea()
    {
        return this.Height * this.Width;
    }
}

public class Square : Rectangle
{
    public override void SetHeight(int height)
    {
        this.Height = height;
        this.Width = height;
    }

    public override void SetWidth(int width)
    {
        this.Height = width;
        this.Width = width;
    }
}
```

Do hình vuông có 2 cạnh bằng nhau, mỗi khi set độ dài 1 cạnh thì ta set luôn độ dài của cạnh còn lại. Tuy nhiên, khi chạy thử, hành động này đã thay đổi hành vi của của class *Rectangle*, dẫn đến vi phạm LSP.

```
Rectangle rect = new Rectangle();
rect.SetHeight(10);
rect.SetWidth(5);
System.Console.WriteLine(rect.CalculateArea()); // Kết quả là 5 * 10

Rectangle square = new Square();
square.SetHeight(10);
square.SetWidth(5);
System.Console.WriteLine(square.CalculateArea()); // Kết quả là 5 x 5.

// Nếu đúng phải là 10x5, vì diện tích 1 hình chữ nhật là dài x rộng
// Class Square sửa hành vi của class cha Rectangle
```

Trong trường hợp này, để code không vi phạm LSP, ta phải tạo 1 class cha là class *Shape*, sau đó cho *Square* và *Rectangle* kế thừa class *Shape* này.

Lưu ý và kết luận

Đây là nguyên lý... dễ bị vi phạm nhất, nguyên nhân chủ yếu là do sự thiếu kinh nghiệm khi thiết kế class. Thông thường, ta design các class dựa theo đời thật: hình vuông là hình chữ nhật, chim cánh cụt là chim. Tuy nhiên, không thể bê nguyên văn mối quan hệ này vào code.

Hãy nhớ một điều:

> Trong đời sống, A là B (hình vuông là hình chữ nhật, chim cánh cụt là chim) không có nghĩa là class A nên kế thừa class B. Chỉ cho class A kế thừa class B khi class A thay thế được cho class B.
> Pin con heo là pin nhưng không thay thế được cho pin, chim cánh cụt là chim nhưng không thay thế được cho chim, do đó 2 ví dụ này vi phạm LSP.

Nguyên lý này ẩn giấu trong hầu hết mọi đoạn code, giúp cho code linh hoạt và ổn định mà ta không hề hay biết. Ví dụ như trong C#, ta có thể chạy hàm foreach với *List, ArrayList, LinkedList* bởi vì chúng cùng kế thừa interface *IEnumerable*. Các class *List, ArrayList*, .. đã được thiết kế đúng LSP, chúng có thể thay thế cho *IEnumerable* mà không làm hỏng tính đúng đắn của chương trình.

INTERFACE SEGREGATION PRINCIPLE – NGUYÊN LÝ PHÂN TÁCH INTERFACE

Nội dung nguyên lý

Thay vì dùng 1 interface lớn, ta nên tách thành nhiều interface nhỏ, với nhiều mục đích cụ thể

Giải thích nguyên lý

Trước khi giải thích nguyên lý, mình nhắc lại khái niệm interface một chút cho các bạn quên mất nhé.

> Interface là một lớp rỗng chỉ chứa khai báo về tên phương thức không có khai báo về thuộc tính hay thứ gì khác và các phương thức này cũng là rỗng. Bởi vậy bất kỳ lớp nào sử dụng lớp interface đều phải định nghĩa các phương thức đã khai báo ở lớp interface[41].

Để thiết kế một hệ thống linh hoạt, dễ thay đổi, các module của hệ thống nên giao tiếp với nhau thông qua interface. Mỗi module sẽ gọi chức năng của module khác thông qua interface mà không cần quan tâm tới implementation bên dưới. Như đã

[41] Xem thêm tại: http://viecbonus.com/blog/interface-va-asbtract-class-trong-php-la-gi/

nói ở trên, do interface chỉ chứa khai báo rỗng về method, khi một class implement một interface, class đó phải implement toàn bộ các method được khai báo trong interface đó.

Điều này tương đương với việc nếu ta tạo ra một interface lớn (hơn 100 method chẳng hạn), mỗi class sẽ phải implement toàn bộ 100 method đó, kể những method không bao giờ sử dụng đến. Nếu áp dụng ISP, ta sẽ chia interface này ra thành nhiều interface nhỏ, các class chỉ cần implement những interface có chức năng mà chúng cần, không cần phải implement những chức năng thừa nữa.

Ví dụ minh họa

Đây là nguyên lý dễ hiểu nhất trong SOLID, các bạn chỉ đọc code một chút là hiểu ngay! Giả sử ta muốn viết một chương trình giới thiệu thuộc tính của các loài động vật. Động vật nào cũng có thể ăn, uống, ngủ, ta thiết kế interface IAnimal như sau.

```
public interface IAnimal {
  void Eat();
  void Drink();
  void Sleep();
}
public class Dog : IAnimal {
  public void Eat () {}
  public void Drink () {}
  public void Sleep () {}
}
public class Cat : IAnimal {
  public void Eat () {}
  public void Drink () {}
  public void Sleep () {}
}
```

Khi ta muốn thêm 1 số loài động vật mới và tính năng vào, ta phải thêm có thêm method vào trong interface như: bơi lội, bay, săn mồi,... Điều này làm interface phình to ra. Khi một loài động vật kế thừa interface, nó phải implement luôn cả những hàm không dùng đến.

```
public interface IAnimal {
  void Eat();
  ...
  void Swim();
  void Fly();
}
public class Fish : IAnimal {
  public void Eat () {}
  ...
  public void Swim() {}
  public void Fly() { throw new Exception("Cá ko biết bay"); }
}
public class Bird : IAnimal {
  public void Eat () {}
  ...
  public void Swim() { throw new Exception("Chim ko biết bơi"); }
  public void Fly()
}
```

Trong thực tế cũng vậy, khi cần thêm chức năng mới, ta thường thêm method vào interface, làm cho interface ngày càng phình to ra. Khi thêm method vào interface *IAnimal*, những class cũ như *Dog*, *Cat* đều phải implement những method mới nên mất thời gian. Giải pháp trong tình huống này là tách interface *IAnimal* ra thành các interface nhỏ như sau:

```
public interface IAnimal {
  void Eat();
  void Drink();
  void Sleep();
}
public interface IBird {
  void Fly();
}
public interface IFish {
  void Swim();
}

// Các class chỉ cần kế thừa những interface
// có chúc năng chúng cần
public class Dog : IAnimal {
  public void Eat() {}
  public void Drink() {}
  public void Sleep() {}
}
public class FlappyBird: IAnimal, IBird {
  public void Eat() {}
  public void Drink() {}
  public void Sleep() {}
  public void Fly() {}
}
public class FormosaFish: IAnimal, IBird {
  public void Eat() {}
  public void Drink() {}
  public void Sleep() {}
  public void Swim() {}
}
```

Để có thể phân tách một interface lớn thành các interface nhỏ một cách hợp lý, các bạn nên xem lại bài viết về Single Responsibility Principle. Tuy nhiên, đôi khi việc tách ra nhiều interface có thể làm tăng số lượng interface, tăng số lượng class, ta cần cân nhắc lợi hại trước khi áp dụng nhé.

Lưu ý và kết luận

Áp dụng nguyên lý ISP sẽ làm hệ thống linh hoạt hơn, đồng thời giảm thiểu code thừa (do phải implement những tính năng không cần thiết). Tuy nhiên, trong thực tế vẫn có nhiều trường hợp bất khả kháng mà chúng ta phải tạo 1 interface lớn.

Ví dụ: Trong một ứng dụng ASP.NET, nếu muốn implement chức năng đăng nhập/phân quyền cho user, ta phải implement

iinterface *MembershipProvider*. Interface này khá là bự với hơn 27 method (Giờ có khi còn nhiều hơn).

```
public bool ChangePassword(string username,
                           string oldPassword, string newPassword)
public bool DeleteUser(string username,
                           bool deleteAllRelatedData)
public bool EnablePasswordRetrieval()
public int MinRequiredPasswordLength()

// Thực sự ta chỉ cần method này
public bool ValidateUser(string username, string password)
```

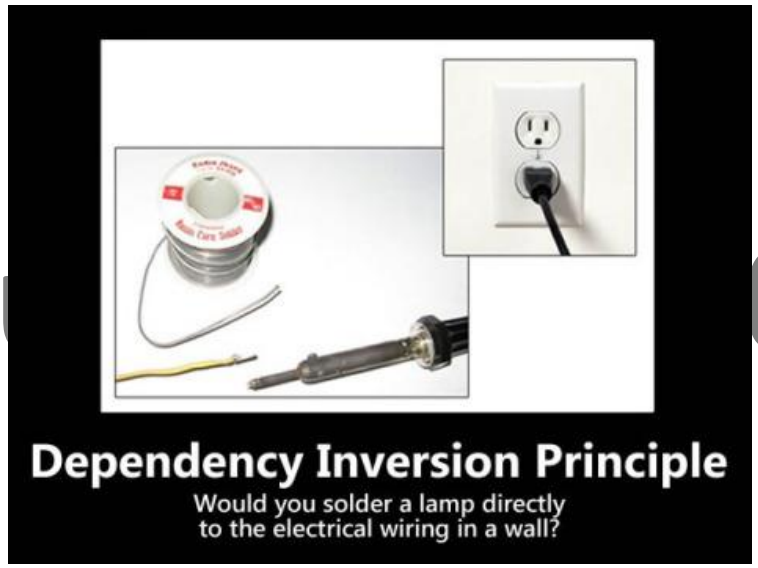
Có lẽ mục đích của Microsoft khi design interface này là để bắt ta phải code toàn bộ các chức năng cần có của việc đăng nhập và phân quyền. Tuy nhiên, do interface này lớn, nếu ta chỉ muốn làm phân quyền đơn giản thì việc implement 27 method này là hoàn toàn mất thời gian và không cần thiết.

DEPENDENCY INVERSION PRINCIPLE – NGUYÊN LÝ ĐẢO NGƯỢC DEPENDENCY

Nội dung nguyên lý

1. Các module cấp cao không nên phụ thuộc vào các module cấp thấp. Cả 2 nên phụ thuộc vào abstraction.

2. Interface (abstraction) không nên phụ thuộc vào chi tiết, mà ngược lại. (Các class giao tiếp với nhau thông qua interface, không phải thông qua implementation.)



Giải thích nguyên lý

Trong bài, mình hay dùng từ module. Trong thực tế, module này có thể là 1 project, 1 file dll, hoặc một service. Để dễ hiểu, chỉ trong bài viết này, các bạn hãy xem mỗi module là một class nhé.

Với cách code thông thường, các module cấp cao sẽ gọi các module cấp thấp. Module cấp cao sẽ phụ thuộc và module cấp thấp, điều đó tạo ra các dependency. Khi module cấp thấp thay đổi, module cấp cao phải thay đổi theo. Một thay đổi sẽ kéo theo hàng loạt thay đổi, giảm khả năng bảo trì của code.

Nếu tuân theo DIP, các module cấp thấp lẫn cấp cao đều phụ thuộc vào 1 interface không đổi. Ta có thể dễ dàng thay thế, sửa đổi module cấp thấp mà không ảnh hưởng gì tới module cấp cao.

Để dễ hiểu, bạn hãy nhìn vào cái mấy cái đèn điện trong nhà mình. Ở đây, module cấp cao chính là ổ điện, interface chính là đuôi đèn tròn, 2 module cấp thấp là bóng đèn tròn và bóng đèn huỳnh quang.

Hai module này đều kế thừa interface đuổi tròn, Ta có thể dễ dàng thay đổi 2 loại bóng vì module cấp cao (ổ điện) chỉ quan tâm tới interface (đuôi tròn), không quan tâm tới implementation (bóng đèn tròn hay huỳnh quang).

Trong code cũng vậy, khi áp dụng DIP, các module liên kết với nhau thông qua interface. Để kết nối tới database, ta chỉ cần gọi hàm Get, Save … của interface *IDataAccess*. Khi thay database, ta chỉ cần thay implementation của interface này.

Ví dụ minh họa

Code khi chưa áp dụng DIP

```
// Cart là module cấp cao
public class Cart
{
    public void Checkout(int orderId, int userId)
    {
        // Database, Logger, EmailSender là module cấp thấy
        Database db = new Database();
        db.Save(orderId);

        Logger log = new Logger();
        log.LogInfo("Order has been checkout");

        EmailSender es = new EmailSender();
        es.SendEmail(userId);
    }
}
```

Code sau khi đã thiết kế lại, áp dụng DIP:

```
// Interface
public interface IDatabase
{
    void Save(int orderId);
}
public interface ILogger
{
    void LogInfo(string info);
}
public interface IEmailSender
{
    void SendEmail(int userId);
}

// Các Module implement các Interface
public class Logger : ILogger
{
    public void LogInfo(string info) {}
}
public class Database : IDatabase
{
    public void Save(int orderId) {}
}
public class EmailSender : IEmailSender
{
    public void SendEmail(int userId) {}
}

// Hàm checkout mới sẽ như sau
public void Checkout(int orderId, int userId)
{
    // Nếu muốn thay đổi database, logger
    // ta chỉ cần thay dòng code dưới
    // Các Module XMLDatabase, SQLDatabase
    //phải implement IDatabase

    // IDatabase db = new XMLDatabase();
    // IDatebase db = new SQLDatabase();
    IDatabase db = new Database();
    db.Save(orderId);

    ILogger log = new Logger();
    log.LogInfo("Order has been checkout");

    IEmailSender es = new EmailSender();
    es.SendEmail(userId);
}
```

Trong thực tế, người ta thường áp dụng pattern Dependency Injection để đảm bảo nguyên lý DI trong code. Do 2 khái niệm này liên quan đến nhau nên mình sẽ nó kĩ hơn ở bài viết kế tiếp về Dependency Injection

Tóm tắt

- Ngoài 4 khái niệm cơ bản trong OOP, ta còn phải biết 5 nguyên lý SOLID để thiết kế và code tốt hơn.
    - **Single Responsibility**: Một class chỉ nên giữ một trách nhiệm duy nhất (Chỉ có thể sửa đổi class với một lý do duy nhất)
    - **Open/Closed**: Có thể thoải mái mở rộng một class, nhưng không được sửa đổi bên trong class đó (open for extension but closed for modification)
    - **Liskvol Substitution**: Trong một chương trình, các object của class con có thể thay thế class cha mà không làm thay đổi tính đúng đắn của chương trình
    - **Interface Segregation**: Thay vì dùng một interface lớn, ta nên tách thành nhiều interface nhỏ, với nhiều mục đích cụ thể
    - **Dependency Inversion**:. Các module cấp cao không nên phụ thuộc vào các modules cấp thấp. Cả hai nên phụ thuộc vào abstraction. Interface (abstraction) không nên phụ thuộc vào chi tiết, mà ngược lại. (Các class giao tiếp với nhau thông qua interface, không phải thông qua implementation.)
- Các nguyên lý này chỉ là hướng dẫn giúp cho code của bạn tốt hơn, sạch hơn, dễ bảo trì hơn. Áp dụng các nguyên lý này vào code có thể giúp bạn cải thiện được chất lượng code, nhưng cũng có thể làm nó rườm rà, dài hơn và phức tạp hơn.
- Không phải cứ cứng nhắc áp dụng nhiều nguyên lý và design pattern thì code sẽ tốt hơn. Một developer giỏi sẽ hiểu rõ những ưu nhược điểm của chúng và chỉ áp dụng một cách hợp lý để giải quyết vấn đề.

DEPENDENCY INJECTION VÀ INVERSION OF CONTROL

Như đã nói ở bài trước, Dependency Injection (DI) và Inversion of Control (IoC) là hai khái niệm khó hiểu nhưng rất phổ biến trong các framework hiện đại. Mình cũng thấy vài bạn đã đi làm 1, 2 năm mà vẫn còn "ngáo ngơ" về DI, IoC, chỉ biết sử dụng nhưng không hiểu rõ bản chất của nó.

Do đó, mình viết bài này nhằm giải thích một cách đơn giản dễ hiểu về Dependency Injection. Các bạn junior nên đọc thử vì DI được áp dụng rất nhiều trong các ứng dụng doanh nghiệp và rất hay gặp khi đi làm hoặc phỏng vấn.

Phần 1: Định nghĩa

Trong bài, mình hay dùng từ module. Trong thực tế, module này có thể là 1 project, 1 file dll, hoặc một service. Để dễ hiểu, chỉ trong bài viết này, các bạn hãy xem mỗi module là một class nhé.

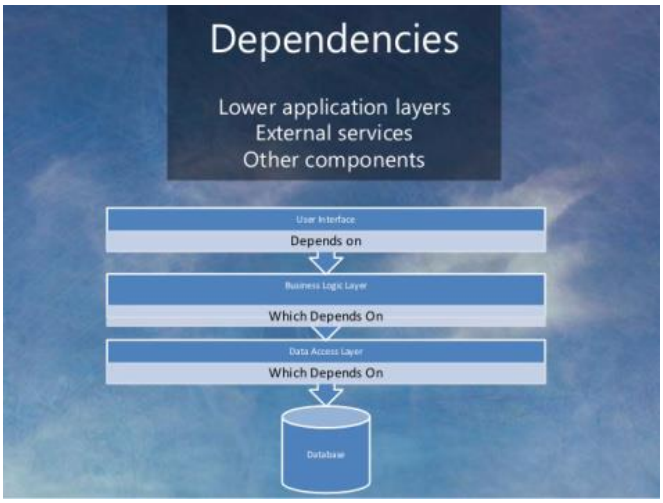
Nhắc lại kiến thức

Trước khi bắt đầu với Dependency Injection, các bạn hãy đọc lại bài viết về trước về những nguyên lý SOLID thiết kế và viết code. Nguyên lý cuối cùng trong SOLID chính là Dependency Inversion:

1. Các module cấp cao không nên phụ thuộc vào các modules cấp thấp. Cả 2 nên phụ thuộc vào abstraction.

2. Interface (abstraction) không nên phụ thuộc vào chi tiết, mà ngược lại. (Các class giao tiếp với nhau thông qua interface, không phải thông qua implementation.)

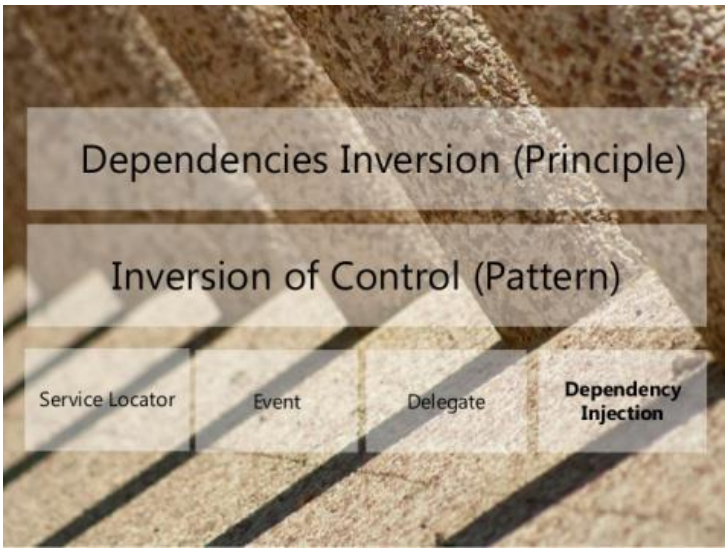
Với cách code thông thường, các module cấp cao sẽ gọi các module cấp thấp. Module cấp cao sẽ phụ thuộc và module cấp thấp, điều đó tạo ra các dependency. Khi module cấp thấp thay đổi, module cấp cao phải thay đổi theo. Một thay đổi sẽ kéo theo hàng loạt thay đổi, giảm khả năng bảo trì của code.



Nếu tuân theo nguyên lý Dependendy Inversion, các module cùng phụ thuộc vào 1 interface không đổi. Ta có thể dễ dàng thay thế, sửa đổi module cấp thấp mà không ảnh hưởng gì tới module cấp cao.

Định nghĩa và khái niệm DI

Hiện nay, các lập trình viên hay lẫn lộn giữa các khái niệm Dependency Inversion, Inversion of Control (IoC), Dependency Injection (DI). Ba khái niệm này tương tự nhau nhưng không hoàn toàn giống nhau.



Sự khác biệt giữa 3 khái niệm trên:

- **Dependency Inversion**: Đây là một nguyên lý để thiết kế và viết code.
- **Inversion of Control**: Đây là một design pattern được tạo ra để code có thể tuân thủ nguyên lý Dependency Inversion. Có nhiều cách hiện thực pattern này: ServiceLocator, Event, Delegate, … Dependency Injection là một trong các cách đó.
- **Dependency Injection**: Đây là một cách để hiện thực Inversion of Control Pattern (Có thể coi nó là một design pattern riêng cũng được). Các module phụ thuộc (dependency) sẽ được inject vào module cấp cao.

Khi nói tới DI, đa phần là nói tới Depedency Injection. Hiện nay, một số DI Container như Unity, StructureMap,… hỗ trợ chúng ta trong việc cài đặt và áp dụng Dependency Injection vào code, tuy nhiên vẫn có thể gọi chúng là IoC Container, ý nghĩa tương tự nhau.

Có thể hiểu Dependency Injection một cách đơn giản như sau:

1. Các module không giao tiếp trực tiếp với nhau, mà thông qua interface. Module cấp thấp sẽ implement interface, module cấp cao sẽ gọi module cấp thấp. Ví dụ: Để giao tiếp với database, ta có interface *IDatabase*, các module cấp thấp là *XMLDatabase, SQLDatabase*. Module cấp cao là *CustomerBusiness* sẽ sử dụng interface *IDatabase*.
2. Việc khởi tạo các module cấp thấp sẽ do DI Container thực hiện. Ví dụ: Trong module *CustomerBusiness*, ta sẽ không khởi tạo *IDatabase db = new XMLDatabase()*, việc này sẽ do DI Container thực hiện. Module *CustomerBusiness* sẽ không biết gì về module *XMLDatabase* hay *SQLDatabase*.
3. Việc Module nào gắn với interface nào sẽ được thiết lập trong code hoặc trong file XML.
4. DI được dùng để làm giảm sự phụ thuộc giữa các module, dễ dàng hơn trong việc thay đổi module, bảo trì code và testing.

Các dạng DI

Có 3 dạng Dependency Injection:

- **Constructor Injection**: Các dependency sẽ được container truyền vào (inject vào) 1 class thông qua constructor của class đó. Đây là cách thông dụng nhất.
- **Setter Injection**: Các dependency sẽ được truyền vào 1 class thông qua các hàm Setter.
- **Interface Injection**: Class cần inject sẽ implement 1 interface. Interface này chứa 1 hàm tên *Inject*. Container sẽ injection dependency vào 1 class thông

qua việc gọi hàm *Inject* của interface đó. Đây là cách rườm rà và ít được sử dụng nhất.

Ưu điểm và khuyết điểm của DI

Dĩ nhiên, DI không phải vạn năng, nó cũng có những ưu điểm và khuyết điểm, do đó không phải project nào cũng nên áp dụng DI. Với những dự án lớn, code nhiều, cần sử dụng DI để đảm bảo code dễ bảo trì, dễ thay đổi.

Vì vậy, bản thân các framework nổi tiếng như Spring, Struts2, ASP.NET MVC, … đều hỗ trợ hoặc tích hợp sẵn DI. ASP.NET MVC từ bản 5 trở xuống cho phép ta sử dụng DI container từ thư viện, từ bản 6 thì tích hợp sẵn DI luôn, không cần phải thêm thư viện gì.

Ưu điểm	**Khuyết điểm**
Giảm sự kết dính giữa các module Code dễ bảo trì, dễ thay thế module Rất dễ test và viết Unit Test	Khái niệm DI khá "khó tiêu", các developer mới sẽ gặp khó khăn khi học

Dễ dàng thấy quan hệ giữa các module (Vì các dependency đều được inject vào constructor)	Sử dụng interface nên đôi khi sẽ khó debug, do không biết chính xác module nào được gọi Các object được khởi tạo toàn bộ ngay từ đầu, có thể làm giảm performance Làm tăng độ phức tạp của code

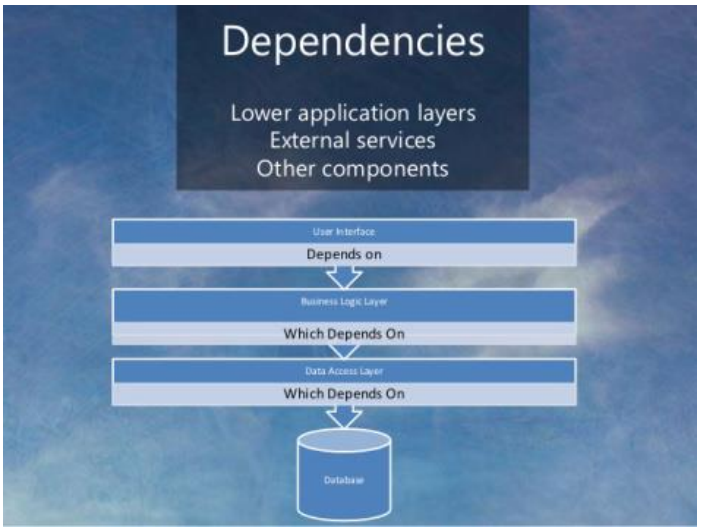
Bài viết khá nặng về lý thuyết nên nếu bạn vẫn chưa mường tượng được sẽ áp dụng DI như thế nào vào code cũng đừng lo. Ở phần 2 mình sẽ bổ sung code minh họa, các bạn đọc xong quay lại đọc bài này sẽ dễ "thông" hơn.

Phần 2: Áp dụng DI vào code

Phần 2 này sẽ cung cấp những đoạn code mẫu, giải thích rõ hơn về những điều mình đã nói ở phần 1. Sau khi đọc xong phần này, các bạn quay lại phần 1 thì sẽ thấy "thông" ra được nhiều thứ nhé.

Dependency là gì?

Dependency là những module cấp thấp, hoặc các service gọi từ bên ngoài. Với cách code thông thường, các module cấp cao sẽ gọi các module cấp thấp. Module cấp cao sẽ phụ thuộc và module cấp thấp, điều đó tạo ra các dependency.



Để dễ hiểu, hãy xem hàm *Checkout* của class *Cart* dưới đây. Hàm này sẽ lưu order xuống database và gửi email cho người dùng. Class *Cart* sẽ khởi tạo và gọi module *Database*, module *EmailSender*, module *Logger*, các module này chính là các dependency.

```
public class Cart
{
    public void Checkout(int orderId, int userId)
    {
        Database db = new Database();
        db.Save(orderId);

        Logger log = new Logger();
        log.LogInfo("Order has been checkout");

        EmailSender es = new EmailSender();
        es.SendEmail(userId);
    }
}
```

Cách làm này có gì sai không? Có vẻ là không, viết code cũng nhanh nữa. Nhưng cách viết này "có thể" sẽ dẫn tới một số vấn đề trong tương lai:

Rất khó test hàm *Checkout* này, vì nó dính dáng tới cả hai module *Database* và *EmailSender*.

Trong trường hợp ta muốn thay đổi module *Database, EmailSender*,... ta phải sửa toàn bộ các chỗ khởi tạo và gọi các module này. Việc làm này rất mất thời gian, dễ gây lỗi.

Về lâu dài, code sẽ trở nên "kết dính", các module có tính kết dính cao, một module thay đổi sẽ kéo theo hàng loạt thay đổi. Đây là nỗi ác mộng khi phải bảo trì code. Inversion of Control và Dependency Injection đã ra đời để giải quyết những vấn đề này.

Làm sao để hạn chế coupling giữa các class? Đã có Inversion of Control

Để các module không "kết dính" với nhau, chúng không được kết nối trực tiếp, mà phải thông qua interface. Đó cũng là nguyên lý cuối cùng trong SOLID.

1. Các module cấp cao không nên phụ thuộc vào các modules cấp thấp. Cả 2 nên phụ thuộc vào abstraction.

2. Interface (abstraction) không nên phụ thuộc vào chi tiết, mà ngược lại. (Các class giao tiếp với nhau thông qua interface, không phải thông qua implementation.)

Ta lần lượt tạo các interface *IDatabase, IEmailSender, ILogger*, các class kia ban đầu sẽ lần lượt kế thừa những interface này. Để dễ hiểu, giờ mình sẽ tạm gọi *IDatabase, IEmailSender, ILogger* là **Interface**, các class như *Database, EmailSender, Logger* là **Module**.

```
// Interface
public interface IDatabase
{
    void Save(int orderId);
}
public interface ILogger
{
    void LogInfo(string info);
}
public interface IEmailSender
{
    void SendEmail(int userId);
}

// Các Module implement các Interface
public class Logger : ILogger
{
    public void LogInfo(string info)
    {
        //...
    }
}
public class Database : IDatabase
{
    public void Save(int orderId)
    {
        //...
    }
}
public class EmailSender : IEmailSender
{
    public void SendEmail(int userId)
    {
        //...
    }
}
```

Hàm checkout mới sẽ trông như sau:

```
public void Checkout(int orderId, int userId)
{
    // Nếu muốn thay đổi database, ta chỉ cần thay dòng code dưới
    // Các Module XMLDatabase, SQLDatabase phải implement IDatabase
    //IDatabase db = new XMLDatabase();
    //IDatebase db = new SQLDatabase();
    IDatabase db = new Database();
    db.Save(orderId);

    ILogger log = new Logger();
    log.LogInfo("Order has been checkout");

    IEmailSender es = new EmailSender();
    es.SendEmail(userId);
}
```

Với interface, ta có thể dễ dàng chuyển đổi các module cấp thấp mà không ảnh hưởng tới module *Cart*. Đây là bước đầu của IoC.

Để dễ quản lý, ta có thể bỏ tất cả những hàm khởi tạo module vào constructor của class *Cart*.

```
public class Cart
{
    private readonly IDatabase _db;
    private readonly ILogger _log;
    private readonly IEmailSender _es;

    public Cart()
    {
        _db = new Database();
        _log = new Logger();
        _es = new EmailSender();
    }

    public void Checkout(int orderId, int userId)
    {
        _db.Save(orderId);
        _log.LogInfo("Order has been checkout");
        _es.SendEmail(userId);
    }
}
```

Cách này thoạt nhìn khá khá ổn. Tuy nhiên, nếu có nhiều module khác cần dùng tới *Logger*, *Database*, ta lại phải khởi tạo các Module con ở constructor của module đó. Có vẻ không ổn phải không nào?

Từ Service Locator tới Dependency Injection

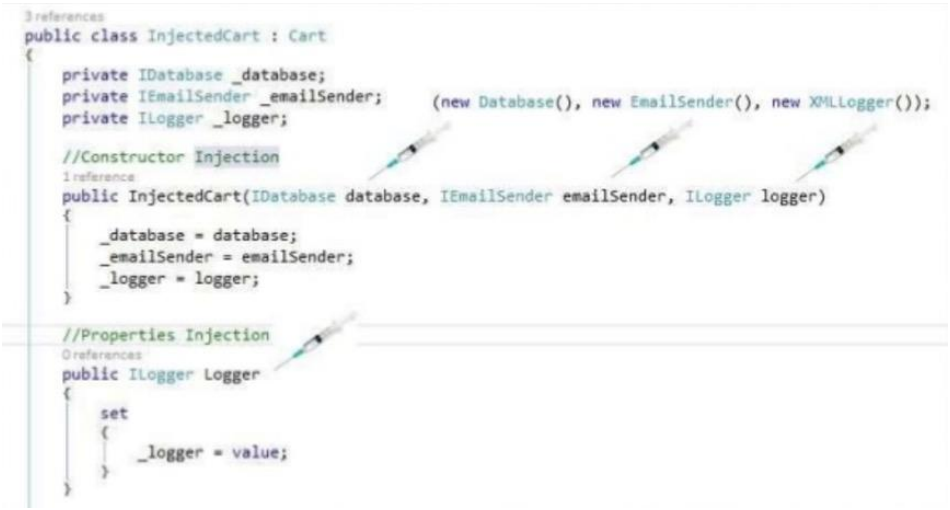
Ban đầu, người ta dùng ServiceLocator để giải quyết vấn đề này. Với mỗi Interface, ta set một Module tương ứng. Khi cần dùng, ta sẽ lấy Module đó từ *ServiceLocator*. Đây cũng là một cách để hiện thực IoC.

```
public static class ServiceLocator
{
    public static T GetModule()
    {
        //....
    }
}

//Ta chỉ việc gọi hàm GetModule
public class Cart
{
    public Cart()
    {
        _db = ServiceLocator.GetModule();
        _log = ServiceLocator.GetModule();
        _es = ServiceLocator.GetModule();
    }
}
```

Cách này vẫn còn khuyết điểm: toàn bộ các class đều phụ thuộc vào ServiceLocator.

Dependency Injection giải quyết được vấn đề này. Các Module cấp thấp sẽ được inject (truyền vào) vào Module cấp cao thông qua Constructor hoặc thông qua Properties. Nói một cách đơn giản dễ hiểu về DI:

> Ta không gọi toán tử new để khởi tạo instance, mà instance đó sẽ được truyền từ ngoài vào (Truyền manual, hoặc nhờ DI Container).



Sau khi áp dụng Dependency Injection, ta sẽ sử dụng class Cart như sau:

```
public Cart(IDatabase db, ILogger log, IEmailSender es)
{
        _db = db;
        _log = log;
        _es = es;
}

// Dependency Injection một cách đơn giản nhất
Cart myCart = new Cart(new Database(),
                  new Logger(), new EmailSender());
// Khi cần thay đổi database, logger
myCart = new Cart(new XMLDatabase(),
              new FakeLogger(), new FakeEmailSender());
```

Chắc bạn nghĩ: Sau khi dùng Dependency Injection thì cũng phải khởi tạo Module à, thế thì còn dở hơn ServiceLocator rồi. Thông thường, ta sử dụng DI Container. Chỉ việc định nghĩa một lần, DI Container sẽ tự thực hiện việc inject các module cấp thấp vào module cấp cao.

```
//Với môi Interface, ta define một Module tương ứng
DIContainer.SetModule<IDatabase, Database>();
DIContainer.SetModule<ILogger, Logger>();
DIContainer.SetModule<IEmailSender, EmailSender>();

DIContainer.SetModule<Cart, Cart>();

//DI Container sẽ tự inject Database, Logger vào Cart
var myCart = DIContainer.GetModule();

//Khi cần thay đổi, ta chỉ cần sửa code define
DIContainer.SetModule<IDatabase, XMLDatabase>();
```

Kết luận

Sau khi áp dụng Dependency Injection, code bạn sẽ dài hơn, có vẻ "phức tạp" hơn và sẽ khó debug hơn. Đổi lại, code sẽ linh hoạt, dễ thay đổi cũng như dễ test hơn.

Như mình đã nói ở bài trước, không phải lúc nào DI cũng là lựa chọn phù hợp mà ta cần phải cân nhắc các ưu khuyết điểm. DI được áp dụng trong nhiều framework back-end (ASP.MVC, Struts2) lẫn front-end (AngularJS, KnockoutJS). Đa phần các dự án lớn đều áp dụng DI, do đó những kiến thức về DI sẽ rất hữu ích khi phỏng vấn cũng như làm việc.

Vậy cái DI Container phía trên ở đâu ra? Ta có thể tự viết, hoặc sử dụng một số DI Container phổ biến trong C# như: *Unity, StructureMap, NInject*. Ở phần 3, mình sẽ hướng dẫn cách viết một DI Container đơn giản và dùng các DI Container sẵn có nhé.

Phần 3 - Viết DI Container và sử dụng thư viện có sẵn

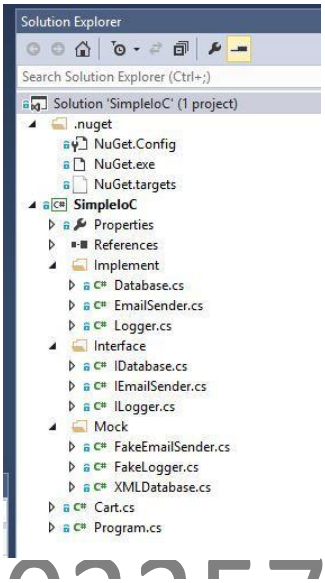
Sau 2 phần đầu, chắc các bạn đã có cái nhìn tổng quan về DI và cách áp dụng chúng vào code. Đa phần chúng ta không tự viết sử dụng các DI Container nổi tiếng như: Unity, NInject, StructureMap.

Để hiểu nguyên lý hoạt động của chúng, mình sẽ cùng các bạn cách viết một DI Container đơn giản (chúng cũng không quá "ghê gớm" hay phức tạp như bạn nghĩ đâu). Sau đó mình sẽ hướng dẫn cách sử dụng cái DI Container có sẵn, cũng như áp dụng IoC vào một project MVC.

Tự viết 1 DI Container đơn giản

Các bạn có thể dùng git để clone project về máy và bắt đầu làm theo mình: https://github.com/ToiDiCodeDaoSampleCode/SimpleIoC. Các class và

interface vẫn như trong phần 2, có điều mình đã bổ sung thêm 1 số class mock – module giả. Trong thực tế, ta sử dụng các class mock này để viết Unit Test.



DI Container thường có 1 function dùng để setup module và interface, một function khác để lấy module dựa theo interface. Ở đây mình gọi 2 function đó là *SetModule* và *GetModule*.

```
public class DIContainer
{
    public static void SetModule<TInterface, TModule>()
    {
        SetModule(typeof(TInterface), typeof(TModule));
    }

    public static T GetModule<T>()
    {
        return (T)GetModule(typeof(T));
    }
}
```

Code của hàm Main cũng rất đơn giản. Ta chỉ cài đặt các interface và module tương ứng thông qua function *SetModule*. Với class Cart, ta chỉ cần gọi hàm *GetModule*. *DIContainer* sẽ tự inject *IDatabase, ILogger* vào theo code ta đã viết.

```
//Với môi Interface, ta define một Module tương ứng
DIContainer.SetModule<IDatabase, Database>();
DIContainer.SetModule<ILogger, Logger>();
DIContainer.SetModule<IEmailSender, EmailSender>();

DIContainer.SetModule<Cart, Cart>();

//DI Container sẽ tự inject Database, Logger vào Cart
var myCart = DIContainer.GetModule<Cart>();
```

Class *DIContainer* sẽ có các đặc tính sau:

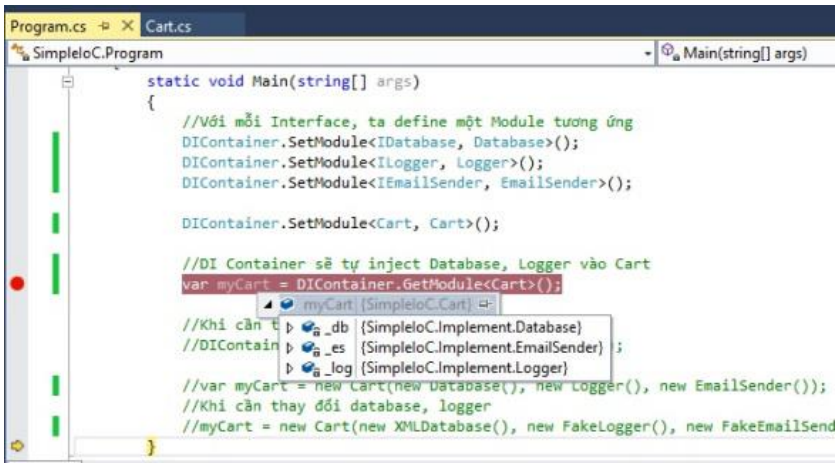
- Lưu trữ các Interface, Module tương ứng vào một Dictionary có Key là Interface, Value là Module. Để lấy một Module từ Container, ta cần đưa vào Interface của Module đó.
- Khi cài đặt một module, container sẽ tìm Constructor đầu tiên của module đó.
- Nếu contructor không có tham số (Module không có dependency), container sẽ gọi constructor này để khởi tạo module.
- Nếu constructor này có tham số (Có dependency), container sẽ khởi tạo các tham số này, gán chúng vào constructor của module. Đây là quá trình injection.

Việc implement cũng không phức tạp lắm, bạn đọc code và comment sẽ hiểu thôi.

```
public class DIContainer
{
    //Dictionary để chứa các interface và module tương ứng
    private static readonly Dictionary<Type, object>
              ResgisteredModules = new Dictionary<Type, object>();

    //Hai hàm cơ bản, ở đây mình chuyển <T> thành
    //dạng Type trong C# để dễ viết code
    public static void SetModule<TInterface, TModule>()
    {
        SetModule(typeof(TInterface), typeof(TModule));
    }
    public static T GetModule<T>()
    {
        return (T)GetModule(typeof(T));
    }

    …
}
```

```
public class DIContainer
{
    …

    private static void SetModule(Type interfaceType, Type moduleType)
    {
        //Kiểm tra module đã implement interface chưa
        if (!interfaceType.IsAssignableFrom(moduleType))
        {
            throw new Exception("Wrong Module type");
        }

        //Tìm constructor đầu tiên
        var firstConstructor = moduleType.GetConstructors()[0];
        object module = null;
        //Nếu như không có tham số
        if (!firstConstructor.GetParameters().Any())
        {
            //Khởi tạo module
            module = firstConstructor.Invoke(null);
            //new Database(), new Logger()
        }
        else
        {
            //Lấy các tham số của constructor
            var constructorParameters = firstConstructor.GetParameters();

            var moduleDependecies = new List<object>();//IDatebase, ILogger
            foreach (var parameter in constructorParameters)
            {
                var dependency = GetModule(parameter.ParameterType);
                //Lấy module tương ứng từ DIContainer
                moduleDependecies.Add(dependency);
            }

            //Inject các dependency vào constructor của module
            module = firstConstructor.Invoke(moduleDependecies.ToArray());
        }
        //Lưu trữ interface và module tương ứng
        ResgisteredModules.Add(interfaceType, module);
    }

    private static object GetModule(Type interfaceType)
    {
        if (ResgisteredModules.ContainsKey(interfaceType))
        {
            return ResgisteredModules[interfaceType];
        }
        throw new Exception("Module not register");
    }
}
```

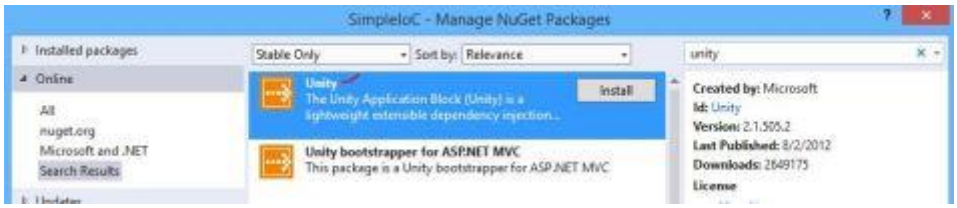
Kết quả khi chạy code

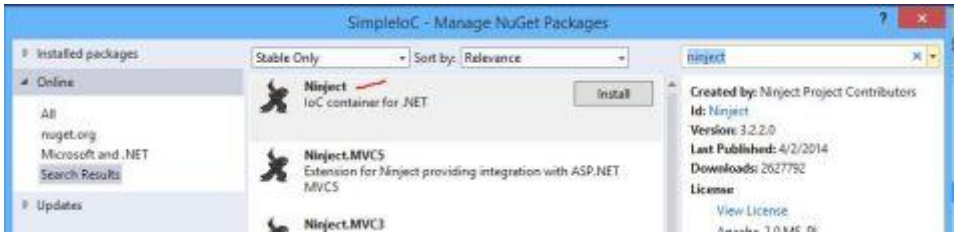


Sử dụng DI Container từ các framework có sẵn

Tất nhiên, nếu người khác đã viết sẵn, kiểm thử và sửa lỗi, chúng ta có thể tái sử dụng mà không cần phải viết lại từ đầu cho mệt[42]. Mình sẽ hướng dẫn các bạn sử dụng DI Container của *Ninject* và *Unity*.

Dùng Nuget (hoặc Package Manager Console) để cài đặt *Ninject* và *Unity*. Nhấp chuột phải vào project SimpleIoC, chọn Manage Nuget packages.





[42] Đọc lại bài "Đôi khi cắm đầu ngồi code là cách ngu nhất..."

Vì bạn đã chia code ra thành các module và interface rồi, ta chỉ cần thay code của DIContainer thành code Unity và Ninject là được.

Ninject

```
var kernel = new StandardKernel();
kernel.Bind<IDatabase>().To<Database>();
kernel.Bind<ILogger>().To<Logger>();
kernel.Bind<IEmailSender>().To<EmailSender>();
kernel.Bind<Cart>().To<Cart>();

//DI Container sẽ tự inject Database, Logger vào Cart
var myCart = kernel.Get<Cart>();
```

Unity

```
var container = new UnityContainer();
container.RegisterType<IDatabase, Database>();
container.RegisterType<ILogger, Logger>();
container.RegisterType<IEmailSender, EmailSender>();
container.RegisterType<Cart, Cart>();

//DI Container sẽ tự inject Database, Logger vào Cart
var myCart = container.Resolve<Cart>();
```


Tóm tắt:

- Khi nhắc tới DI, đa phần người ta nói đến Dependency Injection. Đây là một design pattern, các module phụ thuộc (dependency) được inject vào các module cấp cao.
- DI làm giảm sự kết dính giữa các module, giúp code dễ bảo trì và nâng cấp nhưng cũng làm nó khó debug hơn và phức tạp hơn.
- Có 3 dạng DI là: Constructor, Setter và Interface, trong đó Constructor DI được sử dụng phổ biến nhất.
- Trong các dự án thực tế, người ta thường sử dụng các DI Container đã viết sẵn như Unity, Inject, StructureMap (Tùy ngôn ngữ lập trình)

SAI LẦM HAY GẶP CỦA LẬP TRÌNH VIÊN MỚI VÀ NHỮNG MÁNH KHÓE CỦA CÁC LẬP TRÌNH VIÊN VĨ ĐẠI

Ở bài trước, mình đã có nói đến những sai lầm trong việc phát triển bản thân và thái độ làm việc[43]. Bài viết này sẽ đề cập đến những sai lầm trong chính chuyên môn lập trình.

Top 9 sai lầm hay gặp

Có thể tóm gọn các sai lầm này trong "5 chữ Không" và "4 chữ Thích".

- Không quan tâm đến việc đặt tên hàm, tên biến: Giữ thói quen từ khi ra trường, nhiều bạn chỉ đặt tên biến kiểu x1, x2. Rất ngắn gọn và dễ hiểu nhưng đọc … không hiểu gì.
- Không quan tâm đến cấu trúc code: Code chỉ cần chạy được là xong, còn cấu trúc code rối như đống giẻ lau cũng mặc.
- Không quan tâm đến bảo mật: Nghĩ rằng bảo mật là chuyện của bọn hạ tầng IT, mình chỉ lo code tính năng thôi.
- Không quan tâm đến testing: Code chạy đúng một trường hợp chính là được, mấy trường hợp còn lại thì kệ nó, không cần phải test thử, cho bên tester làm.
- Không quan tâm đến performance: Không sử dụng cấu trúc dữ liệu hợp lý, không optimize code.
- Thích code theo phong cách phức tạp lòng vòng, hoặc thông minh nhưng … khó hiểu.
- Thích cắm đầu ngay vào code cả khi chưa biết rõ yêu cầu hay business logic: Chưa xác định được vấn đề đã cắm đầu code tìm hướng giải quyết.
- Thích một ngôn ngữ/công nghệ đến mức mù quáng, lúc nào cũng nghĩ nó là nhất [44].
- Thích code lại từ đầu, tạo ra cái mới mà không quan tâm đến những cái đã có để tiết kiệm thời gian, công sức, tiền bạc.

[43] Xem lại bài "Top 20 sai lầm của lập trình viên"

[44] Xem lại bài "Đừng xem ngôn ngữ lập trình như tôn giáo"

Để sửa chữa những sai lầm này, ta có thể học hỏi từ những lập trình viên giỏi và nhiều kinh nghiệm. Mình đã tổng hợp và chọn lọc những ý tưởng và câu trả lời hay của câu hỏi: What are the best-kept secrets of great programmers? – Những mánh khóe "không bao giờ tiết lộ" của các lập trình viên vĩ đại.

Mánh khóe thứ nhất: Không bao giờ truyền dạy các "mánh khóe" này cho người khác. Hết!

Đùa thôi, các bạn tiếp tục đọc nhé!

Mánh khóe code và test

- Trong đa phần các trường hợp, sử dụng inheritance (kế thừa) là một design tệ, làm cho code khó test và khó bảo trì. Hãy chuyển qua composition (sở hữu) và kết hợp với interface. (Có thể đọc thêm về *prefer composition over inheritance*).
- Đừng sử dụng interface cho tới khi bạn hoàn toàn rõ ràng về domain của chương trình. (Mỗi khi cần thêm một function, bạn sẽ phải thêm nó vào interface và implement của interface đó, tốn gấp đôi công sức).
- Bảo mật/mã hóa rất khó. Đừng tự làm mà hãy tái sử dụng (sử dụng thư viện, thuật toán có sẵn v…v), trừ khi bạn biết rõ mình đang làm gì.
- Có vô vàn nguyên nhân làm crash một chương trình: deploy sai cách, dữ liệu bị lỗi, người dùng dùng sai cách, quá tải … Chuẩn bị sẵn sàng cho những điều đó: Ghi log những exception gặp phải, deploy thử lên server test, đặt giới hạn cho bộ nhớ…
- Kết nối mạng (HTTP, socket) rất dễ xảy ra vấn đề. Luôn nhớ đặt timeout cho các kết nối này, sử dụng thư viện để wrap chúng, kết nối lại khi kết nối có vấn đề.
- Mỗi dòng code thêm vào sẽ làm chương trình phức tạp thêm một chút và tăng khả năng có bug. Bỏ bớt code là cách hay nhất để giảm bớt số lượng bug.
- Kiểm tra lại những thứ người dùng nhập vào, vừa đảm bảo tính bảo mật, lại hạn chế được bug.
- Tái sử dụng code chưa chắc đã khiến code của bạn dễ bảo trì hơn. Tái sử dụng code giữa 2 domain khác nhau có thể làm chúng "dính chặt" với nhau hơn.
- Chỉ test những thứ cần test, test ít thì dễ sót bug, test nhiều thì sẽ mất thời gian và tốn công cập nhật lại test case mỗi khi đổi requirement.
- Mỗi khi commit code, hãy giữ số lượng code nhỏ, code chạy được, viết message rõ ràng bao gồm thứ bạn đã làm và lý do bạn làm thứ đó.

- Với kiến trúc tốt, bạn vẫn có thể viết code tệ. Tuy nhiên, với kiến trúc tốt, bạn có thể dễ dàng nâng cấp, thay thế phần code tệ đó. Tập trung xây dựng kiến trúc tốt, ít móc nối trước, về sau sẽ dễ thở hơn.
- Code để lâu cũng rất dễ hư hỏng, do đó cần được refactor thường xuyên. Tuy nhiên cần tránh refactor code quá độ.

Mánh khóe làm việc

- Rất khó để ước đoán thời gian cần làm để hoàn thành một module và dự án, đó là lý do người ta sử dụng Scrum.
- Viết code để cho chính mình và người khác đọc. Thêm comment để giải thích "Vì sao", thêm comment ở những nơi mà bạn nghĩ 1 năm sau bạn đọc code sẽ không hiểu gì.
- Hiểu rõ thư viện/framework mà mình sử dụng, đừng có gắng viết lại từ đầu những thứ người khác đã tốn công viết rồi.
- Cài đặt để việc build một project diễn ra nhanh chóng tiện lợi nhất có thể. Hãy chắc chắn bạn có thể build bằng command line, sẽ rất có ích (Có thể kích hoạt build từ xa, hoặc đưa project lên CI chẳng hạn).
- Hiểu rõ những tool bạn sử dụng (IDE, source control, build tool, Photoshop). Cố gắng tìm hiểu và làm quen với việc dùng các hotkey, hạn chế dùng chuột. Bạn sẽ làm việc nhanh hơn và "pro" hơn.
- Ngồi lâu rất có hại. Hãy tập một số thói quen để đảm bảo sức khỏe khi làm việc: Không ngồi nhiều, lâu lâu cho mắt nghỉ ngơi, sắp xếp bàn làm việc, bàn phím, chuột sao cho làm việc thoải mái…
- Đừng áp dụng lung tung các framework/ pattern vào dự án để "thể hiện". Không phải lúc nào Test-Driven Development cũng tốt, không phải lúc nào cũng nên áp dụng DI/IoC.

Mánh khóe phát triển bản thân

- Vọc code của các ứng dụng, framework Open Source là cách nhanh nhất để học hỏi và nâng cao trình độ.
- Code review là một trong những cách hay nhất giúp bạn tiến bộ. Có người đánh giá code của bạn sẽ giúp bạn phân biệt code giỏi và dở và tránh những lỗi lầm cơ bản (Ở Việt Nam mình thấy việc code review này làm khá qua loa, khá là chán).
- Học một ngôn ngữ mới sẽ giúp bạn hiểu những khái niệm mới, có cái nhìn mới, cách suy nghĩ sẽ linh hoạt hơn. (Thử chuyển từ C#/Java sang ngôn ngữ scripting như Python hoặc JavaScript, bạn sẽ thấy một chân trời mới).
- Học một ngôn ngữ hướng đối tượng là chuyện dễ. Biết cách thiết kế hệ thống theo hướng đối tượng là chuyện khó. Hãy tìm hiểu các nguyên lý SOLID và

một số Design Pattern, chúng sẽ nâng cao hiểu biết của bạn về thiết kế hướng đối tượng.

- Luôn giữ tinh thần học hỏi, nhưng đừng chạy theo công nghệ mới. Đừng chọn một công nghệ cho một dự án chỉ vì nó hot hay nó mới.

Tóm tắt

- Do thiếu kinh nghiệm, lập trình viên mới thường dễ mắc phải các sai lầm trong chuyên môn lập trình
- Để hạn chế các sai lầm này, nên học hỏi từ những đàn anh có kinh nghiệm (đàn anh trong công ty, developer nổi tiếng trên các trang lớn như Quora, Medium)

BÍ KÍP ĐỂ TRỞ THÀNH "CAO THỦ" TRONG VIỆC FIX BUG

Mình từng có bài viết để than phiền về sự lười biếng, ý lại của các sinh viên ngành lập trình hiện nay [45]. Ngoại trừ một số bạn hỏi về lý thuyết hoặc vấn đề công nghệ, phần nhiều các bạn sinh viên hay lên mạng hỏi khi "gặp lỗi không biết sửa".

Qua đó, có thể thấy các bạn sinh viên năm 2 năm 3 hoặc mới ra trường vẫn còn thiếu kĩ năng debug. Bài viết này là những kinh nghiệm giúp bạn debug và đặt câu hỏi hiệu quả hơn. Mỗi khi thấy ai hỏi bài hay nhờ sửa lỗi các bạn cứ đưa bài viết này để giúp ích cho người ta nhé.

Trời đã sinh dev sao còn sinh bug?

Người ta thường bảo là developer và QA (tester) là kẻ thù không đội trời chung, một bên ráng giấu bug đi còn một bên ráng chạy chương trình để moi bug ra. Tuy nhiên, sự thật không phải vậy. Cả dev và QA đều có kẻ thù chung là bug.

Thế mới có câu chuyện rằng: Một chàng developer gặp phải con bug rất khôn, lúc ẩn lúc hiện. Chàng phải OT hết cả tuần lễ để tìm bug, không có thời gian dắt gấu đi chơi nên gấu bỏ đi theo người khác. Uất hận, chàng ngửa mặt lên trời than "Trời đã sinh dev sao còn sinh bug", sau đó hộc máu mà chết.

Ở phần dưới, mình sẽ chia sẻ những bí kíp tìm lỗi từ sơ cấp đến cao cấp, cùng những điều cần lưu ý để đặt câu hỏi hiệu quả.

Bí kiếp sơ cấp – Lỗi cú pháp

Đây là các lỗi hay gặp khi mới học lập trình, viết sai cú pháp nên chương trình không chạy được: thiếu mở đóng ngoặc, nhầm dấu bằng, thiếu chấm phẩy.

Cách giải quyết: dùng IDE xịn (Visual Studio, Eclipse, Atom). Các IDE này đều hỗ trợ nhắc lỗi cú pháp (Chỉ cần các bạn chịu khó đọc thông báo lỗi bằng tiếng Anh là được).

[45] Xem lại bài "thực trang học lập trình của các sinh viên"

Các lỗi này chỉ cần code nhiều là quen, hầu như code lâu sẽ hết. Bạn có thể tập một số thói quen như: Khi mở ngoặc nhớ đóng ngoặc, cuối câu lệnh phải thêm chấm phẩy.

Bí kiếp trung cấp – Exception

Sau khi sửa các lỗi cú pháp, chương trình đã build được, nhưng khi chạy lại crash hoặc quăng Exception. Exception hay gặp nhất là *NullPointerException*, khi bạn muốn truy cập vào một biến có giá trị null.

Các lỗi này cũng không quá khó xử lý. Chỉ cần đọc tên Exception và message kèm theo là bạn có thể hiểu được nguyên nhân gây lỗi. Tiếp theo, hãy copy tên Exception và message vào Google để tìm, câu trả lời thường sẽ có ở một vài link đầu tiên.

Bí kiếp cao cấp – Lỗi framework và logic

Đây thường là những lỗi phức tạp, chương trình chạy sai mà không báo lỗi hay quăng Exception gì. Ví dụ, lỗi mà bạn nào cũng gặp khi mới học là viết = thay cho ==, chương trình chạy sai mà không rõ lỗi là gì.

```
if (x=3) {
  // Do something
}
```


Các lỗi này thường khó sửa vì bạn không biết rõ nguyên nhân gây lỗi. Với những lỗi dạng này, trước tiên bạn cần xác định:

1. Code của mình sẽ chạy các bước nào, gọi những hàm nào
2. Kết quả đúng mà các hàm nên trả về là gì
3. Kết quả thực sự các hàm trả về là gì
4. Kết quả nào bị sai? Hàm nào trả về kết quả đó? Nguyên nhân sai là gì?
5. Xác định hàm chạy sai, lặp lại bước 1

Nếu chưa quen debug, hãy in ra toàn bộ các giá trị và kiểm tra xem có giá trị nào sai hay không. Khi đã quen, bạn chỉ cần đặt breakpoint, cho chương trình chạy từng dòng lệnh và kiểm tra giá trị của từng biến.

Sau khi làm rõ những điều nói trên, nhiều khả năng bạn đã tìm được câu trả lời cho mình. Nếu không tìm được hàm gây lỗi, lục tung Google nhưng không tìm ra nguyên nhân hay cách sửa, bạn sẽ phải dùng đến cách cuối cùng: Vác đi hỏi.

Tàn quyển – Làm sao đặt câu hỏi một cách hiệu quả?

Đầu tiên, hãy đặt mình vào vị trí người được hỏi, liệu khi đọc câu hỏi họ có hiểu gì không. Nhiều bạn cứ hỏi chung chung kiểu: Code không chạy được!! Thánh cũng chả hiểu code của bạn tại sao lại không chạy được. Làm rõ điều cần hỏi là bạn đã trả lời được 50% câu hỏi rồi.

Khi bạn hỏi một câu hỏi stackoverflow, bạn thường được yêu cầu chỉ post đoạn code gây lỗi lên. Hãy tập thói quen này trước khi đi hỏi: Xác định đoạn code gây bug rồi tách riêng nó ra, cố gắng tái tạo lại bug. Việc xác định được đoạn code gây bug là đã giải quyết 50% vấn đề rồi, có khi xác định xong là bạn sửa được bug luôn rồi, chẳng cần phải đi hỏi nữa.

Hãy nhớ một điều, luôn luôn Google và tìm hiểu trước khi đặt câu hỏi. Người được hỏi thường rất sẵn lòng giúp đỡ, nhưng họ sẽ rất bực mình nếu bạn hỏi những câu đơn giản mà chỉ cần 30 giây tìm Google là ra. Việc không chịu tìm hiểu hay Google trước khi hỏi chứng tỏ bạn lười và không tôn trọng thời gian của người được hỏi.

Lời kết

Kĩ năng debug cũng như kĩ năng code đều cần có thời gian rèn luyện mới có thể thành thục. Do đó, đừng buồn hay nản lòng khi bạn tốn quá nhiều thời gian để sửa lỗi. Qua một thời gian bạn sẽ quen dần và nhanh hơn thôi. Nhớ nhé, phải thường xuyên luyện tập code và tự debug thì mới nâng cao được khả năng code lẫn kĩ năng debug nhé.

Đã nói ở phần trên rồi, nhưng mình vẫn nhắc lại thêm lần nữa. Thay vì cứ gặp khó khăn là vác đi hỏi lung tung, nhớ Google 7 lần trước khi hỏi. Về lâu dài, việc này sẽ nâng cao khả năng tìm lỗi và debug của bạn đấy. Đừng để tới lúc đi làm, cứ bí là phải lên Facebook hỏi hoặc quay sang hỏi đồng nghiệp nhé.

Tóm tắt

- Đa phần các bạn sinh viên thường thiếu kĩ năng debug – Tìm và sửa lỗi trong code của mình
- Google và Stackoverflow là bạn thân thiết nhất, hãy copy thông báo lỗi lên 2 trang này để tìm lời giải trước khi đi hỏi
- Hãy tập cách sử dụng debugger trong ngôn ngữ lập trình của mình
- Kĩ năng debug cũng như kĩ năng code, cần thời gian rèn luyện mới giỏi được

CHUYỆN VỀ NHỮNG "Ổ GÀ" TRÊN CON ĐƯỜNG LẬP TRÌNH

Trước khi nói về ngành lập trình, ta hãy nói về ngành cầu đường. Hai ngành này có kha khá điểm chung mà ít người để ý.

> Trên đời này làm gì có đường, người ta đi mãi cũng thành đường thôi. – Lỗ Tấn

Từ chuyện xây đường...

Thuở xưa, để đến nơi, người ta phải đi chân trần, trèo đèo lội suối, đập đá băng rừng. Dần dần người ta chặt cây, nhổ cỏ, làm đường đất để mà đi. Đường đất vừa bẩn vừa nhiều bụi, người ta lại lát đá, lát gạch thành đường đá, đường gạch cho sạch đẹp, dễ đi hơn.

Thế rồi khoa học tiến bộ, người ta không lát đá nữa mà ủi phẳng đường, trải nhựa hoặc bê tông lên. Trên đường nhựa, người ta xây cầu vượt, xây vòng xoay, xe cộ băng băng qua lại nhanh chóng và tiện lợi hơn nhiều so với ngày xưa.

Giờ ta hãy cùng xem lại một chút về lịch sử ngành lập trình. Ngày xưa, các cụ phải lập trình bằng cách bật tắt các công tắc switch để đổi chiều dòng điện. Thấy làm vậy cực quá, họ dùng bit (0 và 1) để thay thế cho các công tắc.

Tuy nhiên, làm việc với những chuỗi số 0110 dài ngoằn nghèo rất mệt mỏi và mất thời gian, do đó họ tạo ra các ngôn ngữ lập trình bậc thấp, vd như Assembly (tức mã máy). Assembly là chữ, do đó dễ đọc dễ sửa hơn. Sau khi viết, Assembly sẽ được dịch thành bytecode.

Dù vậy, việc sử dụng Assembly để viết một chương trình hoàn chỉnh vẫn vô cùng khó khăn. Thế rồi, một số ngôn ngữ lập trình bậc cao (C, C++, Java, C#) dần đần được tạo ra, giải quyết được nhiều công việc hơn với ít code hơn. Trên nền các ngôn ngữ này, họ xây các thư viện/framework (Zoomla, Ruby on Rail, ...). Sử dụng các thư viện này, việc viết một ứng dụng/website trở nên nhanh chóng và đơn giản hơn xưa nhiều.

Ta dễ thấy sự giống nhau đến kì lạ giữa việc làm đường và lập trình. Một chương trình được xây dựng dựa trên vô số nền tảng bên dưới (Framework => Code C#, Java => IL/Java bytecode => bytecode), cũng giống như một con đường được xây dựng dựa trên vô số lớp đường phía dưới. Các bác Khoa học Máy tính gọi những lớp (ngôn ngữ, thư viện, framework) này là lớp trừu tượng (Layer of abstraction).

Các lớp trừu tượng này che giấu các khái niệm phức tạp bên dưới, trừu tượng chúng bằng các khái niệm đơn giản hơn. Ví dụ như khi ta code bằng framework ASP.NET Web Form, Web Form sẽ che giấu các khái niệm HTML, CSS, JavaScript phía

dưới, "trừu tượng" nó bằng các khái niệm control, button, textbox để lập trình viên dễ sử dụng. Hoặc khi ta sử dụng jQuery, nó sẽ che giấu sự khác biệt giữa các trình duyệt, ta chỉ cần viết code 1 lần, chạy được ở mọi trình duyệt.

... cho tới chuyện ổ gà

Buồn thay, con đường của lập trình viên không hề bằng phẳng mà lại có khá nhiều ổ gà ổ voi. Đôi khi, các lớp trừu tượng này không che giấu được hết những ổ gà bên dưới nó. Trong nhiều trường hợp gặp lỗi hoặc performance chậm, ta cần phải bới các "lớp trừu tượng" này lên để tìm nguyên nhân và cách sửa. Dưới đây là một số ví dụ:

- Với WinForm, ai cũng có thể dễ dàng kéo, thả, tạo form, double click vào button để viết code vì việc kéo thả, đã "trừu tượng hóa" việc tạo giao diện bằng code. Tuy nhiên, nếu muốn viết một control hoàn toàn mới, hoặc tạo control động, ta cần phải hiểu rõ code WinForm được tự động sinh ra như thế nào (Khi double click vào button, ta gán một event handler vào event của một button đó, đại loại thế, ...).
- Hiện tại, có rất nhiều bạn đi làm, tham gia các project có sử dụng Dependency Injection với Ninject, Unity nhưng không hề biết bản chất nó là gì cũng như lý do phải sử dụng nó[46].
- LINQ là thứ rất dễ sử dụng, giúp code nhanh chóng hơn. Tuy nhiên, muốn viết một hàm tương tự Where, OrderBy của LINQ, ta phải nắm vững một số khái niệm lằng nhằng như Generic, Callback, Lambda Expression.
- EntityFramework hỗ trợ chúng ta tương tác với database rất đơn giản về dễ dàng, vì nó đã "trừu tượng hóa" SQL và database bên dưới. Tuy nhiên, các bug của EF thường khá khó fix. Để xử lý bug, ta cần phải hiểu rõ cơ chế hoạt động bên dưới của nó (SQL Query được sinh ra thế nào, các trường trong object được mapping ra sao).

Đơn cử như trong trường hợp sau:

[46] Xem lại bài viết về Dependency Injection và Inversion of Control

```
public string getUpperName(string name)
{
  return name.ToUpper();
}


// Hàm A này sẽ bị lỗi
var upperNames = _db.Users
                 .Select(x =>
getUpperName(x.Name)).ToList();

// Hàm B này không lỗi
var upperNames = _db.Users
                 .Select(x => x.Name.ToUpper()).ToList();
```

Hai hàm A và B trông có vẻ tương tự nhau, nhưng hàm A bị lỗi còn hàm B thì không. Nguyên do là: Entity Framework sẽ đọc Lambda Expression truyền vào, dịch ra SQL. Với hàm A, EF không thể dịch hàm *getUpperName* ra SQL được, trong khi hàm ToUpper thì lại dịch ra SQL được. Đấy, vì "ổ gà" trong EF, ta phải hiểu cơ chế hoạt động bên trong của nó thì mới trị được những bug kiểu này.

Làm sao "trám" ổ gà

Những kiến thức về các "ổ gà" như thế thường ít được đề cập trong sách vở mà thường phải tích lũy theo thời gian tiếp xúc và kinh nghiệm làm việc. Đây cũng là thứ để phân biệt trình độ giữa senior và junior. Vì lý do trên, chúng ta có thể dễ dàng học nhanh một ngôn ngữ/framework như WebForm, AngularJS, ... nhưng lại cần nhiều thời gian để tiếp xúc, sử dụng và thành thạo một ngôn ngôn ngữ đó.

Vậy phải làm sao để "trám" những ổ gà này, để đạt tới một "cảnh giới" cao hơn? Hãy đọc bài viết sau: "Điều gì ngăn cản bạn đạt tới cảnh giới "tối cao" trong code học" nhé.

Chú thích

Khái niệm này được bác Joel (đồng sáng lập Stackoverflow) đưa ra trong một bài viết trên blog, nguyên văn tiếng Anh là leaky abstraction: *All non-trivial abstractions, to some degree, are leaky*.

Bài viết này cũng gây khá nhiều tiếng vang khi mới ra mắt. Nếu có thể, các bạn nên đọc bài viết gốc để "thấm" điều tác giả muốn nói: http://www.joelonsoftware.com/articles/LeakyAbstractions.html

Nếu dịch nguyên văn "leaky abstraction" là "rò rỉ trừu tượng" thì nghe khá kì cục. Tìm khắp các bài viết ở Việt Nam cũng chẳng thấy ai nhắc đến khái niệm này, nên mình xin mạn phép Việt hóa khái niệm này bằng cái tên "ổ gà" cho dễ nhớ dễ thuộc vậy.

Tóm tắt

- Xây dựng phần mềm cũng giống như xây đường. Mỗi chương trình được xây dựng dựa trên vô số những lớp nền tảng bên dưới
- Các lớp trừu tượng này che giấu các khái niệm phức tạp bên dưới, trừu tượng chúng bằng các khái niệm đơn giản hơn
- Các lớp trừu tượng này đôi khi che giấu nhiều "ổ gà, ổ voi" nên ta phải hiểu bản chất của chúng thì mới có thể sửa code khi gặp lỗi

ĐIỀU GÌ NGĂN CẢN BẠN ĐẠT CẢNH GIỚI TỐI CAO TRONG "CODE HỌC"?

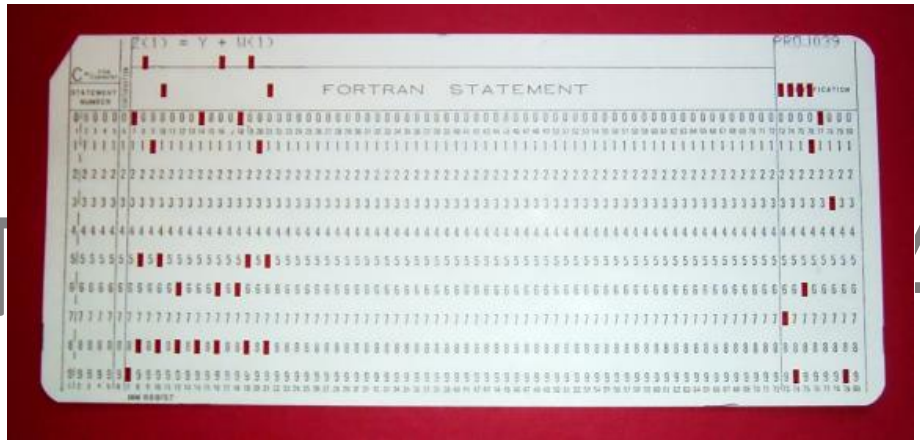
Chuyện ngày xưa

Đã từng có thời, code là một việc cực kỳ mệt nhọc và nhàm chán.

Đã từng có thời, lập trình việc phải làm việc với từng byte từng bit một.

Đã từng có thời, code phải được viết ra giấy, đóng thành thẻ rồi đút vào máy.

Đã từng có thời, ta phải mất cả năm trời để tạo giao diện, quản lý bộ nhớ, viết một chương trình đơn giản.



Ngày xưa, mấy bác lập trình viên code bằng cách "đóng lỗ" lên mấy tấm thẻ như thế này

Chuyện ngày nay

Tuy nhiên, những ngày tháng đó đã chìm vào quá khứ. Như mình đã nhắc tới ở bài trước, lập trình là việc kế thừa những nền tảng được xây dựng bởi những người đi trước. Hàng loạt thư viện/framework ra đời, giúp việc code trở nên nhàn hạ dễ dàng hơn.

Ngày xưa, để làm một website, ta phải học HTML/CSS, viết từng dòng code bằng tay, sau đó hì hục up lên mạng. Giờ đây, chỉ cần 1, 2 ngày với WordPress, Zoomla, Wix… ta đã có một website lung linh chễm chệ trên mạng. Với ứng dụng phức tạp hơn, ta cũng có thể viết nhanh chóng bằng Ruby on Rail, NodeJS, Meteor, …

Thế nhưng, cuộc đời rất công bằng, có được thì cũng có mất. Những thư viện/framework này làm cuộc sống của bạn dễ dàng hơn nhưng cũng là trở ngại của

bạn trên con đường truy cầu đỉnh cao "code đạo". Các framework này giúp bạn làm được nhiều thứ một cách nhanh chóng, nhưng cũng làm bạn trở nên "lười biếng" và "ngu ngốc" hơn.

Hiện đại hay... hại điện?

Hiện nay, đa phần việc lập trình đều dựa trên thư viện/framework do sự mạnh mẽ và tiện dụng của chúng. Có thể nói, sự xuất hiện tràn lan của các thư viện/framework đã sinh ra một thế hệ developer làm biếng (Trong đó cũng có mình), suốt ngày chỉ biết gọi API và dùng framework. Họ có thể học lướt, nắm được cách dùng của một framework rất nhanh, và nghĩ rằng mình đã "thành thạo" ngôn ngữ/framework đó. Tiếc thay, đời đâu có đơn giản như thế.

Nếu một đứa trẻ lớn lên trong chăn ấm nệm êm, chỉ biết ăn sung mặc sướng thì nó sẽ không biết việc kiếm tiền cực khổ thế nào. Lập trình viên cũng thế, đôi khi ta mải hì hục cắm đầu vào viết code cho chạy mà không hiểu được nó làm những gì.

Mình đã gặp vài bạn có dùng Unity, StructureMap mà không hiểu IoC là gì, tại sao phải dùng. Mình cũng từng làm chung nhóm với vài bạn sử dụng Entity Framework nhoay nhoáy nhưng không biết lambda expression là gì, không hiểu câu SQL được sinh ra thế nào, mapping object ra sao, tại sao câu query viết ra lại chạy chậm.

Nếu không thật sự hiểu cách một framework hoạt động, không hiểu tường tận một ngôn ngữ, bạn sẽ không bao giờ đạt tới đỉnh cao, mãi mãi chỉ là thằng thợ code, cầm bàn phím đi code dạo (như mình) thôi.

Những thứ thuộc về bề mặt như cú pháp, cách gọi hàm,... có thể dễ dàng học được một cách nhanh chóng. Nhưng những thứ sâu xa hơn như: cơ chế hoạt động của framework, xử lý asynchronous, performance, tổ chức/thiết kế code và file hợp lý,... không thể học được trong một sớm một chiều.

Những kiến thức dạng này chỉ có thể đạt được sau một quá trình ăn dầm nằm dề lâu dài với ngôn ngữ/framework đó. Ví dụ như trong đoạn code dưới đây, javascript senior dev chỉ cần 10 đến 30 giây là có thể nhìn ra vấn đề, trong khi nhiều developer có thể phải mất cả 15-30 phút hoặc hơn để tìm hiểu và fix lỗi.

```
function addLinks() {
    for (var i = 0, link; i < 5; i++) {
        link = document.createElement("a");
        link.innerHTML = "Link " + i;
        link.onclick = function() {
            alert(i);
        };
        document.body.appendChild(link);
    }
}
// click vào 5 link đều ra số 5
window.onload = addLinks;
```

Con đường nào tới đỉnh cao "code đạo"

Trong một bài viết khác, mình đã từng nói rằng đôi khi "code là cách ngu nhất để giải quyết vấn đề"[47]. Thế nhưng, sẽ có nhiều lúc bạn gặp phải trường gặp "code là cách duy nhất để giải quyết vấn đề". Đôi khi, có vấn đề rất khó nhằn: cần optimize tốc độ, bug nằm trong framework. Để giải quyết những vấn đề này, ta phải có một nền tảng kiến thức vững, một cái nhìn thông suốt về bản chất framework/ngôn ngữ – thứ chỉ có thể đạt được thông qua quá trình tiếp xúc lâu dài.

Các bậc cao nhân xưa cũng từng nói: Không có đường tắt để đi tới đỉnh cao võ đạo (Đường tắt tới đỉnh Phan-xi-păng thì có, vì người ta vừa xây cáp treo xong). Các cụ nước ngoài cũng khuyên là: Để giỏi lập trình, hãy học trong 10 năm. Kiến thức có thể nhồi nhét vào đầu thông qua video hoặc sách vở. Song để biến nó thành kĩ năng, thành kinh nghiệm, bạn cần bỏ thời gian để tự mình tiêu hóa, trải nghiệm.

Kết luận

Tóm lại, điều mình muốn gửi gắm tới các bạn là: các thư viện/framework có thể giúp bạn làm việc nhanh chóng và hiệu quả hơn nhiều; dùng chúng thì dễ, nhưng để thật sự thông thạo chúng, bạn phải cố gắng tìm hiểu cơ chế hoạt động, bản chất thực sự, những cách sử dụng hợp lý nhất (best practice),... của chúng.

Nhớ đừng tự cho là mình giỏi, mình đã biết hết mà hãy luôn giữ thái độ hoài nghi, tò mò "tại sao code chạy được, tại sao lại được API thiết kế như vậy, ...".

[47] Xem lại bài viết cùng tên

Xin kết thúc bài viết bằng lời khuyên của thầy Khánh, một người thầy tại đại học FPT mà mình khá kính trọng:

> Cái quan trọng nhất là phải nắm rõ bản chất vấn đề. Các ông có thể dùng thư viện, dùng framework gì cũng được, nhưng phải hiểu rõ code tụi nó làm gì trong đó!

Tóm tắt

- Ngày nay, với sự trợ giúp của các thư viện/framework, việc code trở nên dễ hơn trước rất nhiều.
- Các thư viện/framework này tạo ra một thế hệ developer làm biếng, chỉ biết viết code mà không hiểu cách thức code hoạt động
- Để có một nền tảng kiến thức vững chắc, cần có thời gian tiếp xúc lâu dài với công nghệ, cũng như giữ sự tò mò để tìm hiểu cơ chế hoạt động và bản chất thật sự của chúng.

PHẦN 3 – KÍ SỰ CODE DẠO

Phần này là những câu chuyện, trải nghiệm và bài học mình rút ra được trong thời gian làm việc ở trong và ngoài nước.

Trung 0335740004

TẠM BIỆT ASWIG – ĐÔI DÒNG TÂM SỰ CỦA CHÀNG JUNIOR DEVELOPER

Ngày 4 tháng 9 năm 2015, mình chính thức nghỉ việc tại Aswig Solutions để bước trên một con đường mới – du học 2 năm ở UK tại đại học Lancaster. Bài viết này là đôi dòng tâm sự kể về một năm làm việc của mình ở đây: Buồn ít, vui nhiều, học được vô số điều bổ ích về technical và cách hành xử trong công việc.

Cuối tháng 9/2014 – Bị phỏng vấn như super junior

Sau khi nộp CV vào công ty với vị trí Junior Developer, mình "được" 2 anh team leader phỏng vấn tới tận hơn một tiếng rưỡi (100% tiếng Anh). Chả biết sao vị trí junior mà mình "được" hỏi đủ thứ từ C#, Linq, MVC, Entity Framework, Database, Front-end (HTML, CSS, Javascript) cho tới OOP, Design Pattern, … riết rồi cứ tưởng tuyển siêu nhân luôn chứ không phải junior gì hết.

Sau vòng 1 căng thẳng, mình được vào nói chuyện nhẹ nhàng với anh K. Manager (Trong lòng cứ chắc mẩm là đã đậu). Đến bây giờ mình vẫn ấn tượng với câu hỏi "Em nghĩ 3 điều gì là quan trọng nhất đối với developer?". Sau này, mình mới biết vòng 2 là vòng đánh giá thái độ làm việc và tính cách nhân viên, suýt nữa phát biểu lung tung là rớt rồi.

Sau khi thỏa thuận lương, mình được hẹn phỏng vấn thêm… lần 3 với Technical Manager nước ngoài – miếng ăn đến tận miệng còn chưa ăn được, thật là khổ. Cũng may là cuối cùng mọi chuyện đều suôn sẻ, đầu tháng 10 mình bắt đầu đi làm, với mức lương khá cao so với mặt bằng junior nói chung (8 chữ số).

Công bằng mà nói, công ty khá rộng rãi với nhân viên. Trong 2 tháng thử việc mình vẫn được tính 100% lương và được khám bệnh miễn phí. Cuối tháng 12 anh Manager vẫn xét thưởng cho mình tháng 13, được gần nửa tháng lương tiền thưởng (Làm nguyên năm là được 2 tháng).

Đầu tháng 10/2014 – Chập chững vào công ty, bị ma cũ "ăn hiếp"

Ngày đầu tiên đi làm, mình được anh Phóng – một anh developer cùng team hướng dẫn cài đặt máy, chỉ chỗ ăn uống (Công ty Aswig có hệ thống buddy, người mới gia nhập sẽ được "buddy" hướng dẫn và giới thiệu). Đến trưa, mình được anh H. Team Leader dắt đi ăn trưa miễn phí. Tính mình dễ lắm, ai bao mình ăn là mình thương liền à.

Quy định của công ty là phải sử dụng tiếng Anh trong công việc. Mới đầu mình cũng hơi ngộp vì thấy có vài bác du học về hoặc phát âm như gió, lúc họp hành thì chưa quen lắm, chẳng nghe được khách hàng nói gì. Sau 2 tuần mình mới dần tập thành thói quen và thích nghi được.

Mình lại ngồi ngay đối diện anh Manager nên không dám ho he nhiều. Ngày xưa đi làm hôm nào mình cũng ghé thăm webtretho, vozforum, kenh14… Từ hồi qua đây, sáng nào mình cũng chỉ dám vào simpleprogrammer.com, stackoverflow.com, codeproject.com, toidicodedao.com, …

Mình được vào team Foundation, một team lo việc đăng nhập và phân quyền cho các ứng dụng khác của công ty. Phase 1 của dự án không thành công lắm, và anh H. đã phải có một quyết định đau thương và táo bạo: Đập nguyên phase 1, làm phase 2 lại từ đầu. Team ban đầu có 3 người, về sau thêm được 1 người nữa là 4. Nhờ khả năng kĩ thuật xuất chúng của các thành viên cộng với khả năng chém gió "thần sầu" của anh team leader, dự án đã "tạm" thành công mĩ mãn.

Giữa tháng 4/2015 – Company trip, Thái Lan vẫy gọi

Mỗi năm công ty tổ chức company trip một lần và hầu như lần nào cũng đi nước ngoài. Kì này cả công ty được qua Thái Lan chơi, ngắm voi sẵn tiện chuyển giới luôn thể. Ở Bangkok 3 ngày nên mình cũng không đi chơi được gì nhiều. Đến hôm cuối cùng, mấy ông trong team rủ nhau đi mua quà cho vợ, ông nào ông nấy xách nguyên vali vài chục kg về. Ngẫm lại mới thấy có vợ cũng chẳng sung sướng gì.

Đầu tháng 5/2015 – Buồn thay chuyện kẻ ở người đi

Sau chuyến đi Thái Lan, nhờ khả năng ngoại giao xuất chúng, anh H. đã lấy một dự án khá béo bở về cho team mang tên Claimbook. Anh Sơn – thành viên thứ 4 của team lại quyết định nghỉ việc, qua Singapore làm… rửa chén kiêm lập trình viên cho một phòng lab ở bên đó (Nghe đồn lương cao đến hơn 4.000 đô Sing).

Anh Sơn là cựu sinh viên kiêm trợ giảng trường Bách Khoa. Khả năng kĩ thuật của anh cũng rất khá, nhưng lúc nào anh cũng bảo: Cỡ anh chỉ là cắc ké ở BK thôi. Mình nhủ thầm: Cắc ké ở BK mà cỡ này, dân BK chắc ghê gớm lắm. Tới lúc thấy ổng khoe bằng Giỏi mình mới biết mình bị lừa, hóa ra trên đời không tin ai cho được. Thiếu anh Sơn, đội ăn-nhậu mất đi một thành viên cộm cán, ai cũng buồn rười rượi….

Đầu tháng 6/2015 – Một tháng đầy biến động

Dự án thiếu người, mình bị "ép" phải dựng toàn bộ wireframe và giao diện cho dự án Claimbook. Đúng là làm super junior nó khổ thế. Một lần nữa, anh H. lại tỏa sáng với khả năng ngoại giao của mình. Anh đã "lôi kéo" được một bác Technical Lead và 2 developer mới vào team.

Anh còn dụ dỗ được team designer tham gia vào quá trình thiết kế cho dự án. Leader Team Designer là anh Sơn Đoàn – người yêu của Adrian Anh Tuấn. Trông anh Sơn đẹp trai và manly hơn ông Team Leader của mình cả chục lần.

Suốt 2 tháng trời, cả nhóm dồn hết tâm sức làm prototype cho dự án Claimbook. Team cũng tổn thất khá nhiều. Anh technical lead do thái độ làm việc không tốt, không phù hợp với văn hóa công ty, vào team mình được một tháng đã phải chuyển qua team khác rồi nghỉ việc ngay sau đó vài tuần. Design của dự án cũng bị thay đổi xoành xoạch theo yêu cầu của khách hàng, team mình cũng ngại không dám liên hệ lại với team designer vì sợ bị chửi do tự ý sửa lung tung design của họ.

Thay mặt anh H., xin gửi lời cảm ơn đến anh Sơn Đoàn đẹp trai, cùng toàn thể các bạn trong team designer đã góp phần vào thành công của dự án. Thấy team em được khen nhiều mà không nhắc đến các đóng góp của team anh nên cũng hơi cắn rứt.

Đầu tháng 8/2015 – Nghỉ việc, tăng lương và cách xử sự của người lãnh đạo

Vào khoảng tháng 5, mình cũng đã chia sẻ với anh H. Team Leader về ý định đi du học vào cuối năm để tiện cho việc sắp xếp nhân sự. Mặc dù khá tiếc nuối nhưng anh vẫn ủng hộ lựa chọn của mình.

Cuối tháng 7, đầu tháng 8, công ty bắt đầu xét tăng lương. Mấy đứa bạn mình ở team khác được tăng cũng kha khá. Anh H. Team Leader khá khó xử khi phải giải thích với mình rằng: Do sắp nghỉ việc nên mình chỉ được thăng chức từ Junior Developer lên Software Engineer mà không được tăng lương, để phần đó lại cho các bạn khác.

Cảm thấy vừa buồn vừa hụt hẫng, mình viết một bức tâm thư gửi anh Manager và anh Team Leader (Thư dài nên mình tóm tắt nhé):

> Chào hai anh, anh H. đã có chia sẻ với em về việc em được promote lên vị trí Software Engineer và không tăng lương.
>
> Em rất hiểu và thông cảm lý do mà hai anh đưa ra quyết định này: Em cũng hiểu anh H. rất khó xử khi phải giải thích chuyện này cho em; anh K. chắc cũng phải đắn đo khi đưa ra quyết định này. Tuy nhiên, bản thân em cảm thấy rất buồn và thất vọng vì nhiều lý do sau: …
>
> Vì vậy, em mong hai anh có thể xem lại kết quả Performance Review của em, hoặc có cách gì hỗ trợ cho em đỡ cảm thấy thiệt thòi. Cảm ơn hai anh vì đã đọc. Chúc hai anh có ngày cuối tuần vui vẻ.

Ngẫm lại, mình thấy mình kiềm chế cảm xúc khá tốt, không làm bù lu bù loa lên. Mình hiểu rằng ở vị trí Manager, có lúc phải đưa ra những quyết định thiệt thòi cho một cá nhân để giữ lợi ích công ty và tập thể. Do đó mình cố gắng lịch sự khi viết thư khi thử đặt mình vào vị trí của họ.

> Kinh nghiệm mình muốn chia sẻ là: Quan hệ và thương hiệu bản thân là những thứ rất khó xây dựng, đừng vì cảm xúc nhất thời mà hành động nông nổi rồi làm mất cả hai.

Ngay chiều hôm sau, mình được anh team lead và chị S. Manager rủ đi cafe nói chuyện và chia sẻ chân thành. Lần đầu mình được nhận lời xin lỗi chân thành của cấp trên "vì đã quên consider suy nghĩ của em khi đưa ra quyết định". Mình cũng được nghe chị S. Manager chia sẻ về việc các công ty VN đối xử không tốt với người sắp nghỉ.

Bản thân công ty cũng đã cố gắng thăng chức cho mình để khen ngợi những đóng góp của mình. Chỉ riêng việc Team Leader và Manager chịu bỏ thời gian ra giải thích, chia sẻ với junior như mình cũng làm mình cảm thấy được tôn trọng rất nhiều.

Tới hôm 1/9/2015, trong cuộc họp toàn công ty, mình được tuyên dương và tặng quà chia tay. Chưa bạn nào rời công ty lại có được đãi ngộ như vậy. Mình cũng hiểu là budget của công ty có hạn, anh H. Team Leader và anh K. Manager phải trích quỹ riêng để có quà cho mình, để mình cảm thấy được coi trọng, có thể ngẩng cao đầu tự hào khi rời khỏi công ty. Cảm ơn tình cảm và sự quan tâm mà 2 anh đã dành cho em. (Tai nghe AKG nghe sướng lắm các bạn ạ).

Kết

Hơn một năm đã qua, buồn vui cũng nhiều. Khả năng code, thái độ của mình cũng tiến bộ đáng kể. Mình cũng quan sát và học hỏi được rất nhiều về cung cách lãnh đạo, quản lý nhờ quan sát anh K., anh H. và các anh team leader khác. Cảm ơn các anh đã nâng đỡ em trên quãng đường vừa qua. ASWIG, hẹn gặp lại!

CHUYỆN ĐẦU NĂM – LẦN ĐẦU ĐI PHỎNG VẤN XIN VIỆC NƠI ĐẤT KHÁCH QUÊ NGƯỜI

Đây là một câu chuyện nho nhỏ về quá trình xin việc (làm lập trình viên part-time) ở UK, cũng là lần đầu mình đi phỏng vấn nơi xứ lạ quê người. Tựa đề khác của câu chuyện là "Tôi đã xin đi code dạo ở nước ngoài như thế nào".

Duyên trời đưa đẩy

Mình qua UK học Computer Science từ hồi tháng 9, trừ một tháng nghỉ hè ra thì tính ra cũng được hơn 4 tháng rồi. Ở giai đoạn đầu, do phải tập làm quen với cuộc sống và việc học nên mình không có thời gian rảnh để đi làm thêm (Vả lại có học bổng 500 củ cũng đủ tiền ăn tiền nhà rồi, cũng không cần xin gia đình gửi tiền qua).

Qua học kì mới, thấy các môn có vẻ nhẹ, ít bài tập nên mình cũng khá rảnh rỗi. Đang định giải quyết nốt vài series dang dở trên pluralsight thì nghe thằng bạn da đen cùng lớp nói "Phòng IT của trường (Infomation Systems Services) chỗ tao thực tập đang tuyển developer hay sao ấy, mài đăng ký thử đi, mài apply thì chắc là đậu".

Nói về anh da đen này một chút. Cha này cũng học Computer Science nhưng mà code không giỏi, lại mất căn bản Java. Do duyên trời đưa đẩy, ngồi cạnh mình trong phòng Lab môn Distributed System, lại còn chung nhóm với môn Advanced HCI. Chắc do thấy mình làm nhóm thì toàn hổ báo chỉ đạo phân việc, làm bài Lab thì lúc nào cũng hổ báo hỏi giáo viên nên anh có vẻ nể mình lắm. Thấy tội tội nên lâu lâu nhờ hướng dẫn mình cũng chịu khó giảng, có điều chắc ảnh không có khiếu code.

UX Developer là cái chi chi???

Quay lại chuyện xin việc. Mình lên trang rao việc xem thử thì thấy vị trí tuyển dụng là UX developer (part-time). Nghe cái title lạ hoắc lạ huơ, xem đến phần yêu cầu công việc thì thấy thế này:

Essential

- Working towards a degree (or equivalent) in a computer related discipline
- Experience of programming in HTML CSS3 / LESS and JQuery in a range of scenarios
- Experience of development in C# Application Form
- Ability to work within a team, and under own initiative

- Experience of achieving clear goals
- Ability to plan, organise and solve complex problems

Desirable

- Experience in Adobe Creative suite (Photoshop, Illustrator etc.), Sketch, Invision or similar prototype tool
- Experience with UX modelling techniques such as case diagrams, use-case descriptions
- A good understanding of website design and responsive UI and common frameworks, Foundation, Bootstrap

Hóa ra là tuyển người làm UI/UX. Chắc là ông trời biết mình từ đầu năm đến giờ mình tập trung vào mảng front-end đây mà. Mình đã từng chém ở bài trước là rành cả front-end và back-end thì quăng đi đâu cũng sống được cả[48], không ngờ linh nghiệm thật.

Nộp CV và chờ mòn mỏi

Khổ thay, bên này xin việc không chỉ cần CV mà còn cần cả thư xin việc (cover letter) đính kèm. Trước giờ mình có viết cái cover letter này bao giờ đâu, thế là phải lên Google tìm một mẫu hay hay, chế lại thành của mình rồi gửi.

Chờ 1 ngày, 2 ngày, rồi 1 tuần rồi mà vẫn chưa thấy đâu. Mình tưởng là rớt rồi nên chửi thầm "Mẹ, mình từng viết series hướng dẫn xin việc với phỏng vấn mà giờ lại trượt ngay vòng gửi CV, nhục vãi". May thay, sau 3-4 ngày, cuối cùng mình cũng nhận được cái email định mệnh.


[redacted] Interview Inbox x

Ashcroft, [redacted]
to me

Hi Hoang Pham,

I am emailing you to invite you to an interview for the role of [redacted] UX Developer.

The interview will be on Wednesday 10th February, in the ISS building.

Before this interview, you will be given a short test to demonstrate your ability. The test will begin at 10.30am, and last twenty minutes.

Your interview will then be at 11.00am and will finish at approximately 11.30am.

Please let me know if you are able to make this interview, and if not, your availability on the 10th February.

Công việc chỉ là developer bán thời gian thôi mà có test, lại còn phỏng vấn, sao mà rắc rối dài dòng thế không biết. Do trong mail không nói rõ nội dung test là gì, mình

[48] Xem lại bài "Kĩ năng cần có của một Web Developer"

dành ngồi lục và ôn dần đống tài liệu HTML/CSS/Javascript, jQuery, REST, API, thôi thì đủ thứ.

Thế rồi, cái ngày định mệnh ấy cũng đến. Liệu Hoàng sẽ phải làm bài test như thế nào, trả lời phỏng vấn ra sao? Liệu Hoàng có nhận được công việc "code dạo" mà anh hằng mong ước? Các bạn xem tiếp phần sau sẽ rõ.

Diện đồ nghiêm túc, lên đường

Mồng 3 Tết, ngày 10/2/2015, trong khi bạn bè ở Việt Nam đang vui vẻ ăn Tết, thoải mái vui chơi đập phá ăn nhậu thì mình phải đi phỏng vấn. Do tối hôm trước ôn kiến thức tới tận khuya, lại lo lắng hồi hộp nên đến tận 2h mình mới ngủ được. Cũng may buổi sáng mình dậy sớm, chẳng muốn ăn nhưng cũng ráng nuốt để buổi trưa khỏi đói.

Nghe nói bọn bên này nghiêm túc lắm nên mình cũng không dám mặc style Steve Job áo thun quần jean như thường lệ để đi phỏng vấn. May mà có mang theo một bộ vest với đôi giày Tây nên mình cũng có một bộ đồ khá tươm tất. Lịch hẹn là 10h30, mình ra đến tòa nhà ISS lúc 10h25 rồi báo với lễ tân. Chả hiểu các bạn ấy bận hay sao mà tới 10h35 hơn mới có một thằng ku xuống dắt mình lên làm test.

Ku này cũng khá trẻ nhưng trộng hơi dị. Dáng cao, người thon gầy, đeo kính, mặt ngô ngố, đúng kiểu nerd/otaku thường gặp (Lớp mình cũng có một thằng tương tự). Thằng ku dắt mình lên tầng 3, vào một căn phòng nhỏ cũng khá xinh, có 2 cái laptop đặt đối diện nhau. Chẳng lẽ đề test là solo mid Dota hay Lol với nó, có 20 phút làm sao đủ?

Hóa ra không phải, bài test cũng khá phù hợp với yêu cầu công việc

> Cho một trang web khá đơn giản với HTML/CSS/jQuery (Sublime Text mở sẵn, muốn dùng IDE gì cũng được). Hãy hiển thị thêm các trường trong dữ liệu, tinh chỉnh giao diện, chọn font cho đẹp mắt và thêm các hiệu ứng animation (UX developer nên yêu cầu phải có khả năng thẩm mĩ).

Đề bài không quá khó, nhưng chắc người ra đề sợ các bạn trẻ hồi hộp làm không kịp nên còn có vài dòng comment gợi ý code chỗ nào, lấy những trường gì. Mình đi phỏng vấn đi làm ở VN chưa lần nào bị bắt phải code, thế mà qua đây xin làm code dạo lại bị bắt viết code mới vui chứ.

Sau 20 phút làm bài, chưa thấy ai vào nên mình chém gió với thằng ku ngồi đối diện. Hóa ra nó là sinh viên năm nhất ngành Computer Science, hiện đang code back-end cho dự án được gần một năm rồi, đúng là tuổi trẻ tài cao.

Ngồi một hồi thì bé Alice (người gửi thư mời interview cho mình) bước vào phòng và dắt mình vào phòng khác phỏng vấn (Kết quả bài test sẽ được check sau, mình cứ tưởng rớt technical là khỏi phỏng vấn chứ).

Phỏng vấn gian truân như đánh đố

Bé Alice dắt mình đi lòng vòng gần 3 phút mới tới được phòng phỏng vấn. Team hình như không lớn (Chỉ 7, 8 người) mà có tới tận 3 người phỏng vấn mình.

- Bé Alice là Administrator kiêm BA, chuyên lo các vấn đề về account và business.
- Bác Brian (hơi lớn tuổi) là Product Owner, kiêm luôn quản lý group, từng code back-end.
- Anh Liam (khá trẻ) là front-end developer, người chịu trách nhiệm cho mảng front-end của dự án.

Cảm giác đầu tiên của mình là… hơi hố. Mình chơi nguyên bộ vest sơ mi quần tây nghiêm túc, còn 2 bác phỏng vấn mình thì… chơi style áo thun quần jean, thiệt là không nói nên lời. Mở đầu phỏng vấn cũng khá nhẹ nhàng, không có cảm giác căng thẳng mấy (may mà hồi xưa PV Aswig mình cũng phải dùng tiếng Anh nên giờ không bỡ ngỡ). Sau vài câu hỏi giới thiệu bản thân, vì sao em học trường này, ngành có gì hay, sở thích là gì (mình chém từ stackoverflow cho đến blog), các bác bắt đầu hỏi nghiêm túc.

Có vẻ là do bài test đã đánh giá được khả năng technical rồi, nên khi phỏng vấn họ không hỏi technical mà tập trung vào khả năng tư duy và giải quyết vấn đề hơn. Format khác hoàn toàn với khi mình phỏng vấn ở Việt Nam. Sau đây là một số câu hỏi xoáy đáp xoay và những câu trả lời của mình:

Anh Liam (Front-end developer)

- Bạn có góp ý gì về bài test? Theo em nó không khó nhưng nên bỏ những phần comment gợi ý, nếu là developer thì sẽ biết viết code ở đâu, như thế nào.
- Bạn hoàn thành bài test như thế nào? Dựa theo đề, em xác định những task cần làm, sắp xếp theo độ ưu tiên. Task nào ưu tiên cao thì làm trước, làm dần dần cho đến hết…
- Bạn có kinh nghiệm với jQuery hay framework gì ko? Mình chém về quá trình tự học jQuery, AngularJS, ionicFramework. Thấy các bác gật gù khí thế.
- Bạn ấn tượng bởi thiết kế của những trang web nào? Vì sao? Mình chọn trang của Apple vì nó tao nhã, nhiều khoảng trắng, đơn giản và một vài trang khác. Câu này mình không chuẩn bị, nên trả lời ko ok lắm.

Bé Alice (Administrator)

- Trong quá trình làm việc, làm sao bạn đảm bảo hoàn thành được mục tiêu? Vì dùng Scrum nên mình không cắm đầu code từ đầu tới cuối mà được

4-50% sẽ confirm lại với team leader hoặc.. Điều này giúp mình không bị chệch hướng hoặc hoàn thành sai mục tiêu.

- Là 1 UX developer, nếu có thể cải thiện UI của Facebook, bạn sẽ làm gì? Câu này rõ là hơi đánh đố, mình xin 20s xuy nghĩ, bác Product Owner cũng ngồi suy nghĩ theo.

Bác Brian (Product Owner)

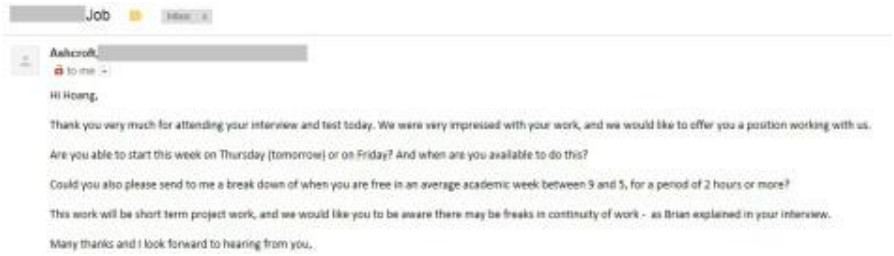
- Kể về một project mà bạn cảm thấy tự hào? Lôi cái project Claimbook từ thời làm ở ASWIG ra chém[49], mình dựng nguyên đống UI/ đưa ý tưởng, từ giai đoạn mock-up cho tới demo.
- Team đang làm ứng dụng iLancaster. Theo bạn cần thêm những chức năng hay cải tiến gì? Câu này thì 2 bên hỏi qua hỏi lại, mình thì chém là phải tìm hiểu người dùng cần gì trước, bên đấy lại đòi nghĩ ra chức năng luôn.

Một hồi sau mình cũng được nghe câu "Bạn có câu hỏi gì không?", mình chỉ hỏi sơ về quy trình làm việc, công việc thường ngày, quy mô team v...v để thể hiện sự nhiệt tình năng nổ thôi. Nhớ lời các cụ dạy, sau khi phỏng vấn mình gởi mail cảm ơn và hẹn liên lạc lại sau[50].

Kết quả

Chắc do thiếu người nên bên này làm ăn khá nhanh gọn: "Sau buổi phỏng vấn, chúng tôi sẽ xem kết quả bài test và thông báo kết quả cho bạn trong vòng chiều nay hoặc ngày mai".

Cũng may, sau một hồi mong ngóng thì ngay chiều hôm đó mình cũng nhận được một cái email định mệnh khác.


Job

Ashcroft

to me

Hi Hoang,

Thank you very much for attending your interview and test today. We were very impressed with your work, and we would like to offer you a position working with us.

Are you able to start this week on Thursday (tomorrow) or on Friday? And when are you available to do this?

Could you also please send to me a break down of when you are free in an average academic week between 9 and 5, for a period of 2 hours or more?

This work will be short term project work, and we would like you to be aware there may be freaks in continuity of work - as Brian explained in your interview.

Many thanks and I look forward to hearing from you.

[49] Đã nhắc đến trong bài viết về "Chia tay ASWIG"

[50] Xem lại chiêu này trong phần "Muôn nẻo đường tìm việc"

Chắc do ít ứng viên nên quy trình tuyển dụng diễn ra khá nhanh. Sáng ngày 10 mình phỏng vấn, chiều ngày 10 có kết quả, gửi mail xác nhận rồi ngay chiều hôm sau xách đít qua ISS đi làm hôm đầu tiên. Hôm đầu đi làm cũng có vài chuyện khá vui nên mình sẽ kể ở bài sau nhé.

NGÀY ĐẦU ĐI CODE DẠO NƠI ĐẤT KHÁCH QUÊ NGƯỜI

Ở phần trước, mình đã kể chuyện đi phỏng vấn xin… code dạo ở nước ngoài. Trong bài này, mình sẽ kể về ngày đầu tiên đi code dạo và những đồng nghiệp cùng dự án nhé.

Hôm đầu ngơ ngác

Như mình đã kể, chắc do thiếu người hay dự án đang cần gấp nên quy trình phỏng vấn và tuyển dụng diễn ra rẹt rẹt, mình vừa phỏng vấn sáng thứ 4 hôm 10/2 thì có offer ngay chiều hôm đó, và sáng thứ 5 hôm sau phải xách đít đi làm.

Team bên này làm ăn cũng đàng hoàng, gửi sẵn cho mình một file PDF hướng dẫn cách liên hệ với admin để request tài khoản và set vào các group. Do team cũng nhỏ nên trong file có ghi đầy đủ tên thành viên, chức vụ, sở thích (chắc để mọi người gần gũi với nhau hơn), đọc cũng khá là vui.

Bé Alice là cái bé phỏng vấn mình nhắc tới trong bài trước ý

Hôm đầu đi làm mình… ngơ ngác không biết gì. Cái tòa nhà ISS của trường khá to nên tới nơi rồi mình vẫn không biết mình làm ở chỗ nào, thế là đành phải nói với lễ tân rồi ngồi chờ người khác xuống dắt mình lên. Team của mình khá nhỏ, chỉ chiếm khoảng 2 dãy bàn.

Vì phần lớn thành viên đều là part-time trừ bác Brian Product Owner và anh Liam Leader team Front-end nên chỉ có khoảng 4 máy trống, không có máy riêng cho mỗi

người. Có vài hôm thành viên của team đi đông đủ quá, không đủ máy nên đành phải lôi laptop ra code.

Đồng nghiệp dễ thương

Team của mình phát triển ứng dụng di động iLancaster, một ứng dụng hỗ trợ sinh viên của đại học Lancaster và đang mở rộng ra cho cư dân sử dụng. Vì công việc của mình là code front-end nên mình làm chung team với anh Liam.

Anh này khá là đẹp trai, thân thiện và lịch sự. Mình gặp thì anh dắt ngay vào phòng họp để giới thiệu tổng quan về cấu trúc và công nghệ của project hiện tại. Sau khi về máy ngồi, anh còn tận tình hướng dẫn cách đăng nhập vào Visual Studio Team, lấy code từ TFS về, làm sao để build và chạy trên local, thiệt tận tình hết sức.

Anh Liam đang giải thích code cho mình thì bác Brian chen vào, bảo anh Liam là "Cùng là developer cả, code trước sau gì thì cũng show, dắt đi giới thiệu mấy chỗ khác trước đã". Thế là bác và anh Liam dẫn mình ra "góc giải trí" nho nhỏ gần đấy.

Chỗ này có phích nước, tủ lạnh, trà/cà phê/sữa miễn phí, làm mình nhớ tới những ngày ở ASWIG, sáng này cũng làm gói trà đào, trà chanh hoặc cafe free của công ty. Lần đầu mình hơi lóng ngóng hay sao mà làm rơi vãi cả đống đường, thiệt là mất hình tượng.

Anh Liam đứng bên trái (Đẹp trai hơn mình xí). Bạn bên phải là ai không nói các bạn cũng biết rồi

Đôi lời về văn hóa làm việc nhóm: Không hiểu có phải do văn hóa của dân Anh hay không mà lâu lâu các bạn lại hỏi *"Are you okay? Any problem?"* để xem mình có gặp

khó khăn hay cần giúp đỡ gì không. Các bạn ấy cũng không có vẻ khó chịu khi bị hỏi (Hoặc có thể khó chịu nhưng không thể hiện ra) mà luôn trả lời và giúp đỡ hết sức nhiệt tình, bản thân mình thấy mình nên học hỏi điều này.

Một điều nữa là có vẻ dân UK hơi… chê tiền. Các cửa hàng và tiệm cafe chỉ mở cửa từ 10-11h sáng cho tới… 5-6h chiều, vào thứ 7 chủ nhật còn đóng cửa sớm hơn, vào khoảng 4-5h chiều gì đó. Hôm đầu đi làm, vừa đúng 5h là anh Liam nhắc nhớ mình nhìn đồng hồ rồi chuẩn bị đi về, không có OT ô tiếc gì như VN đâu nhé[51]. Ở chỗ làm còn có một cái chuông "thần thánh", đúng 12h trưa sẽ có người đánh chuông để mọi người xung quanh biết mà nghỉ trưa.

Tuy thế, team lại làm việc rất nghiêm túc. Đặc biệt, các bác rất thích hiểu tường tận vấn đề, tập trung vào điều người dùng thật sự cần chứ không đơn thuần chỉ là chức năng. Đơn cử như câu chuyện nho nhỏ dưới đây:

Mình và anh Liam đang làm chức năng Adopt An Animal, cho phép các trung tâm đăng thông tin về chó mèo để người dùng vào nhận nuôi. Bác Liam muốn mỗi trung tâm có thể tạo một danh các thông số riêng (Màu lông, tuổi, cân nặng, thuộc giống gì).

Mình và anh Liam đề nghị rằng nên có một danh sách thông số chung để người dùng có thể dễ dàng tìm kiếm. Đây là nội dung cuộc nói chuyện (Mỗi lần bàn về requirement là bác Brian toàn hỏi Cái gì? Tại sao?):

> Liam: Thay vì mỗi trung tâm có 1 danh sách thông số riêng, ta nên có 1 danh sách chung, chuẩn hóa các thông số.
>
> Brian: Tại sao lại cần chuẩn hóa các thông số?
>
> Mình: Nếu không có các thông số chung thì người dùng không thể search/filter dựa theo thông số được.
>
> Brian: Tại sao người dùng cần chức năng search/filter.
>
> Liam & mình: …
>
> Brian: Với chức năng search/filter, họ chỉ tìm thứ họ muốn, chó mèo có thông số không hợp sẽ không được hiển thị. Mục tiêu của ứng dụng là để người dùng xem càng nhiều động vật càng tốt. Việc filter chỉ nên hỗ trợ một số thông số cơ bản như: Loại (chó/mèo), Kích cỡ (lớn/nhỏ)… mà thôi.

Sau một hồi thảo luận thì cuối cùng team cũng thống nhất là chỉ có 4, 5 thông số cơ bản là bắt buộc, các thông số khác có thể tùy chỉnh bởi mỗi trung tâm. Ở đây, mình

[51] Đã nhắc đến trong bài "Mặt tối của ngành công nghệ IT"

và anh Liam đã tập trung vào chức năng và quên mất mục đích của ứng dụng, may là có bác Brian nhắc nhở.

Đống code như mê hồn trận

Nói chuyện con người đủ rồi, giờ nói chuyện kĩ thuật một chút nào. Ứng dụng của team được xây dựng trên framework CampusM. Framework này là cũng tương tự Ionic Framework, hỗ trợ viết app di động kiểu hybrid-app bằng HTML, CSS, JavaScript. Vì nó khá mạnh, tích hợp được nhiều thứ nên được một số trường đại học ở UK sử dụng, trong đó có trường mình.

Ngày xưa, team sử dụng phiên bản 1 của framework (AEK 1) để viết các chức năng. Gần đây, framework ra phiên bản AEK 2 (Tích hợp React, Redux và ES6 khá hầm hố), một số chức năng mới lại viết bằng AEK 2. Thế là code chia ra làm "code cũ" và "code mới", cái nào cũng đủ chuyện nhức đầu.

Ở phiên bản AEK 1, mỗi chức năng được viết trong 1 file duy nhất, chứa hết cả HTML, CSS, JavaScript. Nhìn code rối một cục mà mình chỉ biết lắc đầu thở dài ngao ngán. Chưa kể, các bác developer cũ khá thần thánh khi để HTML trong String, sau đó chơi trò cộng chuỗi HTML trong JavaScript. Do framework AEK có sử dụng dấu {{}} nên các template Javascript như Handlebar bó tay, báo hại mình phải dùng Underscore template để làm code rõ ràng, trong sáng hơn.

Qua phiên bản AEK 2, mỗi module có chia folder riêng, tách component rõ ràng, cảm giác sướng hơn hẳn. Khổ nỗi mình trước giờ có biết React là cái khỉ gì đâu, thế là phải lên pluralsight cày gấp 2,3 khóa về React. Cũng may là nhớ có một vài kiến thức nền tảng và kinh nghiệm nên việc học cũng tương đối dễ dàng, mình chỉ mất thứ bảy chủ nhật để xem clip là hiểu và tham gia ngay việc code được[52].

Song vẫn có một vấn đề khiến mình bực mình: framework mới ra nên các bạn developerment viết document chưa đầy đủ rõ ràng. Nhiều chỗ không biết hỏi ai, lên Google cũng không thấy (vì framework ít người dùng), đành phải lục source code để xem chúng hoạt động thế nào, vừa lôi ông bà họ hàng của bọn viết framework ra mà thăm hỏi.

Tuy nhiên, mình học được vài điều khá thú vị thông qua dự án, đó là phải "đặt mình vào vị trí của user". Mỗi người thường implement đầy đủ một chức năng từ đầu tới cuối. Mỗi khi viết code xong, admin (bé Alice) sẽ xem xét và đưa lên môi trường production để người dùng có thể dùng ngay, đưa feedback để cải tiến. Nhờ vậy, mình có cảm giác là "Thứ mình code ra có người dùng, phải code có lương tâm

[52] Kinh nghiệm trong bài "Cách tiếp cận một ngôn ngữ/ công nghệ mới"

từ UI/UX cho đến function", chứ không chỉ là viết mấy dòng code xong để đó như ngày xưa nữa.

Lần đầu pair programming

Thế rồi một sáng thứ sáu nọ, khi mình ngơ ngác chuẩn bị đi làm thì nhận được mail từ bác Brian. Mail bảo rằng hôm này có một thằng cu mới tên Geogre gia nhập team, buổi sáng nó sẽ được training sơ về framework, chiều thì pair programming với mình để học hỏi. Trước giờ mình có pair programming đâu mà biết, mà "sếp" bảo thế thì thôi cứ làm đại vậy.

Lên tới chỗ làm mới thấy Geogre là một thằng nhóc mập mập tóc vàng, học năm nhất ngành Computer Science. Cơ mà thằng này cũng khá trâu so với tuổi, vừa ngồi pair programming vừa nói chuyện mới biết nó code từ năm lớp 9, đã kinh qua VB, PHP, Java, JavaScript hết cả. Bố khỉ! Anh mày mới ra trường đi làm được 2 năm mà mày code đã được 3 năm rồi.

Pair programming tính ra cũng vui, mỗi người code nửa tiếng, người bên cạnh xem và góp ý, sau đó đổi lượt. Thằng Geogre tính ra cũng khá thông minh sáng dạ, suy nghĩ logic tốt, chỉ gì hiểu nấy, không biết là hỏi liền. Ban đầu mình cứ nghĩ là vừa code vừa giải thích sẽ mệt và mất thời gian, không ngờ việc suy nghĩ, tìm cách giải thích cho nó lại giúp mình suy xét vấn đề cẩn thận hơn, nghĩ ra cách giải quyết tốt hơn.

Tuy nhiên để không mất mặt đàn anh, có vài lần mình dùng hotkey, viết JavaScript theo cú pháp ES6, dùng code đặt debugger, làm thằng ku ngơ ngác phải hỏi "How can you do that?", tính ra mấy trò lòe thiên hạ này cũng hữu dụng thật. Trước mình cứ nghĩ pair programming tốn thời gian mà không ngờ hiệu suất lại cao phết, code một mình nhiều lúc mệt mệt lười lười lại code chậm, có thêm người nữa suy nghĩ cùng vừa vui lại vừa nhanh.

Quẩy – hay còn gọi là Social

Ngày trước, team chỉ có khoảng 5-6 thành viên. Dạo gần đây, do tuyển thêm vài người nên team trở nên "hơi bự". Do đó, bé Alice tổ chức một bữa social (bên này gọi là social, ở Việt Nam mình gọi là nhậu hoặc teambulding). Ở bên này dân nó uống bia còn khiếp hơn cả mình, toàn uống bia cốc bự và không thèm uống đá, uống nhiều mà chẳng thấy say hay đỏ mặt. Team toàn dân kĩ thuật nên các bạn cũng ít chém gió, hầu như chỉ có bác Brian và anh Liam nói nhiều.

Đồ ăn bên này cũng chả có gì ngon nên mình cũng không chụp hình đăng facebook. Sau bữa ăn, các bạn còn ra làm thêm chầu bia trước khi chuyển qua pub quẩy. Vì tuổi già sức yếu, tửu lượng có hạn nên mình đành ngậm ngùi về sớm, không đi tiếp tăng 2 với team được. Chụp tấm hình tự sướng kỉ niệm sẵn giới thiệu bác Brian, anh Liam đập chai và bé Alice cho các bạn luôn.

Bác lớn tuổi áo sọc là Brian, anh đeo kính ngồi cạnh là Liam. Bé Alice góc trong cùng bên phải, mình đẹp trai qua nên khỏi giới thiệu lại…

TẠM BIỆT LANCASTER ISS – TẠM KẾT THÚC KIẾP CODE DẠO NƠI XỨ NGƯỜI

Đầu tháng 9 này, mình sẽ tạm kết thúc năm đầu tiên của chương trình học Master, về Việt Nam vi vu đú đởn khoảng một tháng. Đầu tháng 10 mình lại phải qua Trung Quốc bán thận,… nhầm, bán thân khoảng 6 tháng trước khi quay lại UK kết thúc chương trình học.

Vì lẽ đó, mình phải ngậm ngùi nói lời chia tay với team hiện tại. Lần này, mình viết một bài review nho nhỏ về những trải nghiệm bản thân trong thời gian gần đây.

Môi trường làm việc lý tưởng

Có thể nói, làm việc trong phòng IT của trường Lancaster là một trải nghiệm khá sung sướng với mình.

Ban đầu, công việc chính của mình là front-end: thiết kế, code giao diện và flow cho một số ứng dụng nhỏ trong ứng dụng iLancaster. Team có một nhóm nhỏ riêng chuyên lo về back-end. Tuy nhiên, do có kinh nghiệm viết C# nên đôi khi cần mình cùng nhảy qua team back-end viết RestAPI luôn. Từ UI developer nhảy lên full-stack developer mà không được tăng lương, nhưng được thêm trách nhiệm cũng vui rồi.

Do khả năng code cũng tốt nên mình hoàn thành công việc khá nhanh và đúng deadline, cũng tạo được chút tiếng tăm nho nhỏ trong team. Mỗi khi có gì thắc mắc về UI là mấy bạn intern lại hỏi mình, anh Liam lead team front-end mỗi khi fix bug khó cũng … rủ mình fix chung cho nhanh.

Công ty cũng khá thoải mái nên mình tranh thủ mượn được con Macbook về nhà để code và nghịch ngợm. Từ Win chuyển sang Mac quả là từ địa ngục lên thiên đường.

Máy mỏng, touchpad nhẹ êm, pin trâu mỗi lần xài được 7 tiếng. Đã thế lại chẳng cần tắt mở, đóng máy mở ra làm việc ngay, sướng như tiên ấy! Trước mình toàn chê Macbook, giờ mới biết người chê Macbook chắc chưa dùng Macbook bao giờ cả.

Từ thiên đường xuống địa ngục

Theo chương trình học, mình phải tham gia một công ty được trường chỉ định để hoàn thành một dự án. Chả hiểu may rủi thế nào, mình rơi vào dự án phần mềm của một công ty hóa chất tên The REACH Centre. Dự án này sử dụng PHP, cái thứ ngôn ngữ rẻ tiền mà mình ghét cay ghét đắng. Đúng là ghét của nào trời trao của đó!

Mọi chuyện chưa đừng ở đó! Càng trò chuyện với anh Team Leader mình càng đổ mồ hôi hột:

> **Mình**: Ủa, code để ở đâu vậy anh?
>
> **TL**: À, ở trên server em, dùng FTP để lấy file về code thôi.
>
> **Mình (giả bộ bình tĩnh)**: Ủa không có lưu trong SVN hay Git hả anh?
>
> **TL**: ... À khoảng 1 tuần thì commit vào Git 1 lần để backup.
>
> **Mình (cạn lời)**: Ủa vậy nếu 2 người cùng làm việc trên 1 file thì sao anh?
>
> **TL**: À cái đó lúc trước cũng hay bị đè file. File nào nhiều người làm chung thì tách riêng ra thành nhiều file.
>
> **Mình (đổ mồ hôi)**: Có dùng Framework gì không anh?
>
> **TL**: Không, code PHP thuần và dùng PDO Object thôi em.
>
> **Mình (hạn hán lời)**: ...
>
> ....
>
> **Mình**: Ủa code này đang chạy trên dev server hả anh. Có chạy được trên local không anh?
>
> **TL**: Không, database trên dev server rồi, sửa trực tiếp file PHP trên server rồi chạy thôi.

Mình định hỏi là "sửa file trực tiếp, không lưu Git, lỡ sửa xong không chạy được thì làm sao chuyển lại code cũ hả anh?" mà sợ ảnh không trả lời được nên thôi... Nói chung cảm giác của mình không khác gì từ thiên đường rơi tõm xuống địa ngục. Lại thêm khóa luận lơ lửng trên đầu làm mình buộc phải chạy như con thoi suốt cả tuần liền, cả tuần chẳng có ngày nào yên ổn:

- **Thứ 2**: Đi làm bên ISS từ 9h sáng tới 5h chiều. Tối viết khóa luận từ 9h tới 1h sáng

- **Thứ 3-5**: Đi làm bên The Reach Centre từ 9h sáng tới 5h chiều. Tối về học tiếng TQ, viết khóa luận tới 1-2h sáng
- **Thứ 6**: Tương tự thứ 2
- **Cuối tuần**: Tiếp tục đọc sách, viết khóa luận

Đến giờ ngẫm lại, chả hiểu sao mình vẫn còn sống sốt được, may mắn thật!!

Những điều học được

Dù hơi cực khổ, nhưng trải nghiệm làm 2 công ty một lúc này đã giúp mình học được khá nhiều điều.

Ở team Lancaster ISS, mình học được khá nhiều điều về phong thái cần có của một manager thông qua bác Brian: Biết cách giao đúng việc cho đúng người; Khi đã giao thì tin tưởng thành viên, nhưng thỉnh thoảng vẫn check "How's it going?" để nắm tiến độ, đưa ra giúp đỡ khi cần thiết.

Bác Brian đang thu thập yêu cầu từ khách hàng

Mình cũng học được vài điều khác khi làm việc với mấy anh lập trình viên bên The Reach Center. Có thể kĩ năng technical họ không giỏi (Kiến trúc code hơi rối, code thiếu qui trình) nhưng code rất có tâm và có trách nhiệm. Code của họ có comment đầy đủ, validate đầy đủ, họ cũng rất nhiệt tình trả lời các câu hỏi của mình. Mình chợt ngẫm ra được một điều: Đừng vội đánh giá một developer qua trình độ technical, mà hãy nhìn thái độ code, thái độ làm việc của họ.

Nói lời tạm biệt

Chia tay thì dĩ nhiên là không vui. Mình và bác Brian Product Manager cũng có một cuộc trò truyện nho nhỏ với nhau. Bác có feedback tốt về khả năng lập trình và tinh

thần làm việc của mình. Ngoài ra, bác còn sẵn sàng cho mình recommendation nữa (Có thể cho thông tin liên lạc của bác vào CV để khi xin việc họ có thể hỏi bác về mình), đúng là người tốt.

Bác Brian cũng sẵn sàng cho mình một chỗ trống khi mình quay lại UK vào năm sau, đúng là ở hiền gặp lành mà. Số mình cũng hên, nên lần nào nghỉ việc cũng được tặng quà chia tay cả.

Quà lưu niệm của Lancaster: Áo Alumni, huy hiệu và một hộp trà "đúng chất" UK

Về Việt Nam, hẳn mình sẽ nhớ những lần pair programming, những bữa ăn nhậu hoặc daily meeting bên này. Cảm ơn mọi người vì đã giúp đỡ mình trong thời gian qua. Lancaster, hẹn gặp lại!

LỜI CUỐI SÁCH

Quyển sách này là tổng hợp và biên tập của hơn 60 bài viết hay nhất về nghề nghiệp cũng như kĩ thuật lập trình của Blog Tôi Đi Code Dạo[53]. Nó không chỉ là thành quả nỗ lực của chính mình, mà còn là thành quả của những bạn đọc trung thành đã theo dõi, góp ý cho blog ngay từ những ngày đầu thành lập.

Ngành IT nói chung và ngành lập trình nói riêng là một ngành khó, đòi hỏi nhiều đam mê. Kiến thức trong ngành vô cùng rộng lớn nên không thể gói gọn hết trong một cuốn sách. Dẫu vậy, mình vẫn mong có nhiều bạn sinh viên và lập trình viên mới ra trường biết tới và đọc cuốn sách này. Hi vọng cuốn sách sẽ là người bạn tốt, đồng hành cùng bạn trên những bước chân đầu tiên trong sự nghiệp lập trình.

Do thời gian và kinh nghiệm còn hạn chế nên cuốn sách không tránh khỏi sai sót, kính mong bạn đọc đóng góp ý kiến để sách có thể trở nên hoàn thiện hơn. Mọi góp ý về kĩ thuật hay cách trình bày xin vui lòng gửi về địa chỉ email: huyhoang8a5@gmail.com.

LỜI CẢM ƠN

Chân thành cảm ơn anh Đỗ Kim Cơ và ban biên tập của Tri Thức Trẻ Books đã giúp đỡ và tạo điều kiện cho mình xuất bản cuốn sách này.

Cảm ơn thầy Kiều Trọng Khánh, bác Chris Harvey, anh Lê Văn Song, anh Huỳnh Xuân Thi, anh Nguyễn Cảnh Hưng, anh Lê Anh Bảo, anh Nguyễn Thanh Bình, anh Phạm Hồng Sang đã bỏ chút thời gian trong quỹ thời gian bận rộn để đọc và viết đôi dòng nhận xét cho cuốn sách.

Cảm ơn các anh/chị đồng nghiệp tại FPT Software, ASWIG Solution và Lancaster ISS đã dìu dắt, cho mình những bài học và kinh nghiệm quí giá trong ngành lập trình.

Cảm ơn những bạn đọc trung thành của blog Tôi đi code dạo đã luôn động viên giúp đỡ mình trong quá trình viết blog. Cảm ơn hơn 1300 bạn đã hoàn thành khảo sát và cho mình góp ý về sách trước khi nó được phát hành.

[53] https://toidicodedao.com

GIẢI THÍCH CÁC THUẬT NGỮ HAY DÙNG

Ngành IT có khá nhiều thuật ngữ tiếng Anh phức tạp. Các thuật ngữ này được sử dụng nhiều trong công việc nên mình sử dụng nguyên văn trong bài viết. Điều này có thể sẽ gây khó khăn cho bạn đọc ngoài ngành hoặc các sinh viên mới.

Do vậy, phần này tổng hợp và giải thích một cách ngắn gọn các thuật ngữ hay dùng để bạn đọc dễ tra cứu. (Nếu có thời gian, mình khuyến khích các bạn tra cứu Google từng thuật ngữ để tìm hiểu kĩ hơn).

Thuật ngữ chuyên môn

- API: Viết tắt của Application Programming Interface. Có thể hiểu đây là danh sách những chức năng, dòng lệnh có sẵn mà ta có thể sử dụng.
- Bug: Lỗi của phần mềm. Quá trình tìm lỗi được gọi là debug. Sửa lỗi được gọi là fix bug.
- Build: Quá trình chuyển đổi mã nguồn (code) thành phần mềm (software)
- Code/Source Code: Mã nguồn của phần mềm. Lập trình viên viết mã nguồn để tạo ra phần mềm.
- Coder/Developer: Một cách gọi vui của lập trình viên
- Computer Science: Khoa học máy tính, một ngành học trong ngành IT thiên về nghiên cứu.
- CV: Viết tắt của Curriculum Vitae. CV được sử dụng để xin việc, chứa các thông tin như kĩ năng, kinh nghiệm làm việc
- Database: Cơ sở dữ liệu. Là nơi lưu trữ toàn bộ các dữ liệu của chương trình
- Deadline: Hạn chót để hoàn thành một dự án hoặc sản phẩm
- Debug: Quá trình tìm lỗi trong phần mềm
- Demo: Sản phẩm chưa hoàn chỉnh. Khi ta nói “demo cho khách hàng” nghĩa là giới thiệu một số chức năng cho khách hàng.
- Developer/Dev/Coder: Lập trình viên, còn gọi tắt là dev
- Fix bug: Quá trình sửa lỗi trong phần mềm
- Form: Biểu mẫu, nơi người dùng có thể nhập dữ liệu và gửi cho hệ thống
- Framework: Một bộ khung có sẵn, dựa vào đó để phát triển ứng dụng
- Git: Một hệ thống quản lý mã nguồn phần mềm
- Library/Thư viện: Tập hợp những đoạn code, chức năng đã được viết sẵn mà ta có thể sử dụng
- OT – Overtime: Chỉ việc làm thêm giờ của lập trình viên. Đôi khi có thể làm thêm tới 9-10h tối và cả thứ 7, chủ nhật.
- Outsouce: Gia công phần mềm. Sản xuất phần mềm dựa theo thiết kế, yêu cầu của khách hàng.

- Prototype: Có thể là giả lập hoặc phần mềm chưa hoàn thiện, thường dùng để mô tả phần mềm cho khách hàng
- Quora: Một trang hỏi đáp được khá nhiều lập trình viên nổi tiếng thế giới ưa thích
- Refactor code: Quá trình tinh chỉnh code, giúp code gọn nhẹ, dễ hiểu và dễ mở rộng hơn
- Requirement: Yêu cầu của khách hàng
- Server: Máy chủ. Với các ứng dụng Web, đây là nơi chứa ứng dụng và cung cấp dịch vụ
- Software: Phần mềm
- Software Engieering: Kĩ nghệ phần mềm, một ngành học trong ngành IT thiên về sản xuất phần mềm.
- Stackoverflow: Trang web hỏi đáp lớn nhất thế giới về lập trình.
- SVN: Một hệ thống quản lý mã nguồn phần mềm (tương tự git)
- Technical: Kĩ thuật. Lập trình viên thường thích dùng từ technical hơn từ "kĩ thuật".
- Test: Chạy thử chương trình. Các tester thường chạy thử chương trình để tìm ra bug.

Một số ngôn ngữ lập trình

- C#: Một ngôn ngữ lập trình được Microsoft phát triển, được dùng để lập trình các ứng dụng trên Windows hoặc ứng dụng Web, ứng dụng Window Phone.
- Java: Một ngôn ngữ lập trình miễn phí của Oracle, có thể viết ứng dụng web trên nhiều hệ điều hành. Các ứng dụng Android cũng được viết bằng ngôn ngữ này.
- JavaScript/JS: Khởi đầu là một ngôn ngữ chỉ chạy được trên trình duyệt Web. Hiện tại JS có thể dùng để viết cả ứng dụng web và ứng dụng di động.
- Objective-C và Swift: Ngôn ngữ lập trình do Apple phát triển, được dùng để lập trình ứng dụng cho hệ điều hành Mac và iOS
- PHP: Một ngôn ngữ rất phổ biến, được dùng để lập trình Web. Đa phần các trang web trên mạng được lập trình bằng ngôn ngữ này, trong đó có Facebook và Wikipedia.
- SQL: Viết tắt của SQL Query Language: Đây là ngôn ngữ được dùng để truy cập dữ liệu dưới database.

Về các vị trí trong ngành lập trình[54]

- Junior: Viết tắt của Junior Developer, chỉ các lập trình viên mới ra trường, chưa có nhiều kinh nghiệm. Đôi khi còn được gọi là Fresher.
- Senior: Viết tắt của Senior Developer, chỉ các lập trình viên lâu năm, nhiều kinh nghiệm
- Team Leader: Lập trình viên thường làm việc theo nhóm. Team Leader là người trưởng nhóm, chịu trách nhiệm phân chia công việc, hướng dẫn, đánh giá developer trong nhóm.
- Tester/QC (Quality Control): Nếu Developer là những người tạo ra phần mềm thì tester là những người kiểm tra các phần mềm để tìm ra lỗi nhằm đảm bảo phần mềm chạy đúng.
- BA - Business Analyst: Tìm hiểu hệ thống, lấy yêu cầu của khách hàng và viết thành tài liệu, giải thích cho lập trình viên
- PM - Project Manager: Người quản lý dự án, chịu trách nhiệm giao tiếp với khách hàng, ước đoán tiến độ, quản lý toàn bộ các thành viên trong dự án.

LINK ẢNH VÀ TÀI LIỆU THAM KHẢO

Dưới đây là link nguồn của các hình ảnh trong sách cũng như một số nội dung mà tác giả đã tham khảo trong quá trình viết.

Lập trình viên có cần học đại học hay không?

- https://www.quora.com/What-is-the-point-of-a-computer-science-degree-when-people-say-that-it-doesnt-prepare-you-to-become-a-software-engineer?srid=svRi&share=3848cadb
- https://simpleprogrammer.com/2016/08/15/learning-programming-by-going-to-college/

Tạo động lực học tập và làm việc, sức mạnh của thói quen

- http://jamesclear.com/why-is-it-so-hard-to-form-good-habits
- http://jamesclear.com/goals-systems
- http://zenhabits.net/no-goal/

[54] Đã giải thích trong bài "Con đường phát triển nghề nghiệp của developer"

Cách tiếp cận môt ngôn ngữ, công nghệ mới

Khóa học trên pluralsight:

http://www.pluralsight.com/courses/learning-technology-information-age

Con đường phát triển sự nghiệp (Careet Path) cho developer

Tìm hiểu về ngành BA: http://blog.itviec.com/2015/03/business-analyst/

Mặt tối của ngành công nghệ IT

Khóa học trên Plurasight:

https://www.pluralsight.com/courses/technology-careers-dark-side

Top 18 sai lầm mà các lập trình viên "non trẻ" hay mắc phải

- https://www.quora.com/What-are-mistakes-which-software-engineers-make-in-their-early-years
- https://www.quora.com/What-do-experienced-software-engineers-see-as-the-most-common-mistakes-young-software-engineers-make
- http://iflelectronic.com/19-mistakes-a-software-engineer-make-in-their-early-years/

Lập trình viên nên đọc những sách gì?

- Nguồn ảnh: books.google.com
- Một số sách được giới thiệu tại: http://blog.codinghorror.com/

Xóa mù về Agile và Scrum

Ảnh: https://en.wikipedia.org/wiki/Scrum_(software_development)

Tìm hiểu thêm:

- http://www.agilelearninglabs.com/resources/scrum-introduction/
- http://scrumtrainingseries.com/ <— Có video dễ hiểu
- http://iviettech.vn/blog/283-co-ban-ve-scrum.html
- http://www.scrum-institute.org/The_Scrum_Product_Backlog.php

- http://blog.itviec.com/agile-la-gi-scrum-la-gi/

Tổng quan về UI và UX trong ngành lập trình

- https://medium.com/ux-ui-design
- https://www.quora.com/What-is-the-difference-between-UX-and-UI-designer-and-web-designer

Một button trị giá 300 triệu đô

Nguồn ảnh:

- https://help.optimizely.com/Ideate_and_Hypothesize/Testing_ideas_for_E-commerce_and_retail_sites

Các bạn có thể đọc thêm về câu chuyện "kinh điển" về UI/UX này ở đây:

- https://www.uie.com/articles/three_hund_million_button/
- http://boingboing.net/2011/08/05/300-million-button-making-customers-create-logins-to-buy-cost-etailer-300myear.html
- http://www.fastcompany.com/1147825/300-million-continue-button

Luận về Technical Debt – Nợ kiếp này, duyên kiếp trước

Nguồn ảnh:

- www.slideshare.net/RevistaSG/4-deuda-tecnica
- www.boardofinnovation.com/resources-tools-for-paper-prototyping/

Dành cho các bạn muốn tìm hiểu thêm:

- https://www.techopedia.com/definition/27913/technical-debt
- http://martinfowler.com/bliki/TechnicalDebt.html
- https://blog.codinghorror.com/paying-down-your-technical-debt/

Giới thiệu tổng quan về CI – Continuos Integration

- http://www.ibm.com/developerworks/vn/library/rational/201301/continuous-integration-agile-development/)

- http://www.martinfowler.com/articles/continuousIntegration.html

Nhập môn Design Pattern

Ebook: Design Pattern Head First & Design Pattern For Dummies

Link ảnh

- https://www.codeproject.com/Articles/1009532/Learn-Csharp-Design-patterns-step-by-step-with-a-p
- https://github.com/cathyatseneca/btp600-w16/wiki/Singleton-Design-Pattern---By-Luv-Kapur

Ngoài ebook, các bạn có thể tìm hiểu thêm về design pattern ở đây.

- https://en.wikipedia.org/wiki/Software_design_pattern
- https://sourcemaking.com/design_patterns

SOLID là gì?

Nguồn ảnh

- http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

Lược dịch từ bản gốc

- http://blogs.msdn.com/b/cdndevs/archive/2009/07/15/the-solid-principles-explained-with-motivational-posters.aspx

Single Responsibility Principle

- http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- http://code.tutsplus.com/tutorials/solid-part-1-the-single-responsibility-principle–net-36074
- http://deviq.com/single-responsibility-principle/
- https://dzone.com/articles/single-responsibility

Open/Closed Principle

Nguồn ảnh

- http://preparedgunowners.com/2015/10/22/how-to-increase-handgun-capacity-with-magazine-base-pad-extensions/

Tham khảo thêm:

- http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- http://joelabrahamsson.com/a-simple-example-of-the-openclosed-principle/
- https://lostechies.com/gabrielschenker/2009/02/13/the-open-closed-principle/

Liskov Substitution Principle

- http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- http://stackoverflow.com/questions/56860/what-is-the-liskov-substitution-principle
- http://prasadhonrao.com/solid-principles-liskov-substitution-principle-lsp/

Interface Segregation Principle

- http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- http://code.tutsplus.com/tutorials/solid-part-3-liskov-substitution-interface-segregation-principles–net-36710
- http://www.codeproject.com/Tips/766045/Interface-Segregation-Principle-ISP-of-SOLID-in-Cs
- http://prasadhonrao.com/solid-principles-interface-segregation-principle-isp/

Dependency Injection và Inversion of Control

- http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- https://lostechies.com/derickbailey/2011/09/22/dependency-injection-is-not-the-same-as-the-dependency-inversion-principle/
- http://www.codeproject.com/Articles/538536/A-curry-of-Dependency-Inversion-Principle-DIP-Inve

- http://www.codeproject.com/Articles/615139/An-Absolute-Beginners-Tutorial-on-Dependency-Inver

Sai lầm hay gặp của các lập trình viên

- https://www.quora.com/What-are-the-best-secrets-of-great-programmers
- https://en.wikipedia.org/wiki/Composition_over_inheritance

Bí kíp để trở thành "cao thủ" trong việc debug

- http://blog.codeunion.io/2014/09/03/teaching-novices-how-to-debug-code/
- Sách Debug It! Find, Repair and Prevent Bugs in Your Code

Chuyện về những "ổ gà" trên con đường lập trình

- http://www.joelonsoftware.com/articles/LeakyAbstractions.html

Điều gì ngăn cản bạn đạt cảnh giới tối cao trong "code học"

Nguồn ảnh

- http://www.funnyjunk.com/funny_pictures/4500557/Bitch+please/26#26