TẠP CHÍ LẬP TRÌNH
https://tapchilaptrinh.wordpress.com

TINH GIẢN

Vol.2

04/2013

trò chuyện về Tinh gọn

Tinh gọn hay tiếng Anh là Lean không thực sự là một khái niệm mới mẻ trên thế giới, nhưng ở Việt Nam người biết không nhiều, người hiểu cũng ít, người thực hành lại càng hiếm.

Không rõ Lean đã vào Việt Nam chính xác từ thời điểm nào, nhưng chúng tôi phỏng đoán cũng phải được hơn hai thập kỉ. Toyota vào Việt Nam năm 1995, và ít nhất thì họ cũng mang đến hơi thở Lean từ đấy.

Lean bắt nguồn từ Nhật Bản, đã giúp các công ty Nhật và ngày nay là các công ty trên toàn cầu trong việc tăng lợi nhuận, chất lượng, năng suất, tốc độ đồng thời giảm thiểu nỗ lực, thời gian và chi phí.

Thấy rõ những giá trị mà Lean có thể mang lại cho phát triển phần mềm nói riêng và cuộc sống nói chung, chúng tôi đã chọn Lean làm chủ đề của số 2 lần này. Tạp Chí Lập Trình hy vọng sẽ mang đến cho bạn những hiểu biết và gợi ý nhất định để thực hành Lean.

Chúc các bạn có những phút giây thú vị cùng các bài viết của Tạp Chí Lập Trình!

Hà Nội, tháng 4 năm 2013

Ban biên tập Tạp Chí Lập Trình

trong số này

Vol.2
Tháng 4
2013

SỰ KIỆN

4. ScrumDay Việt Nam 2012
5. Khóa học Lean Mindset
6. Summer Coderetreat 2013

TINH GIẢN

7. Bàn về sự đơn giản
10. 7 nguyên lí của Phát triển phần mềm Tinh gọn
14. 10 nguyên tắc của Lập trình Tinh gọn
21. Tái cấu trúc để đơn giản hơn
23. Hãy tinh gọn trong học tập và cuộc sống
25. Thiết kế không chết
28. Kanban cho mọi người
31. Kiểm thử đơn vị với PHPUnit trên Netbeans

CHIA SẺ

36. Thơ tình bên Computer
37. Bạn đọc với Tạp chí Lập trình

ScrumDay Việt Nam 2012

Ngày 09/12/2012, Đại học FPT và Hanoi Scrum tổ chức hội thảo về Phương pháp Phát triển Phần mềm Linh hoạt (Agile Software Development) - ScrumDay Việt Nam 2012. Đây là lần đầu tiên một hội thảo lớn về Agile được tổ chức tại Hà Nội. Hội thảo tập trung vào các nội dung phát triển phần mềm từ quản lý phát triển sản phẩm, kỹ thuật, công cụ, quy trình, cho đến chiến lược phát triển sản phẩm dựa trên triết lý Agile.

Chính thức xuất hiện từ năm 2001, đến nay Agile đã trở thành triết lý phổ biến nhất thế giới trong việc phát triển các phần mềm từ đơn giản đến phức tạp. Tuy vậy, ở Việt Nam, Agile mới chỉ xuất hiện trong một cộng đồng CNTT hẹp và còn là một ẩn số với đại đa số giới CNTT.

ScrumDay Việt Nam 2012, có sự tham gia của các diễn giả hàng đầu trong nước ở lĩnh vực phát triển phần mềm và Agile. Chủ đề của hội thảo tập trung vào các lĩnh vực phát triển phần mềm từ quản lý phát triển sản phẩm, kỹ thuật, công cụ, quy trình, cho đến chiến lược phát triển sản phẩm dựa trên triết lý Agile. Vì vậy, người tham dự có cơ hội tìm hiểu về phương pháp phát triển phần mềm tiên tiến đang làm thay đổi triết lý phát triển của ngành công nghiệp phần mềm, cũng như giao lưu, chia sẻ kinh nghiệm trong phát triển và ứng dụng Agile\Scrum với các chuyên gia và đồng nghiệp khác.

Mong rằng, ScrumDay Việt Nam cùng với AgileTour (sự kiện thường niên đã được tổ chức tại Sài Gòn từ năm 2011) sẽ góp phần thúc đẩy việc ứng dụng Agile\Scrum trong cộng đồng CNTT Việt Nam, đưa ngành công nghiệp phần mềm Việt Nam ngày càng phát triển.

KHÓA HỌC LEAN MINDSET

Ngày 07/03/2013, Đại học FPT đã tổ chức khóa học "Lean Mindset". Học viên của khóa học đã được làm việc với Tom Poppendieck và Marry Poppendieck, hai tác giả của "Phát triển Phần mềm Tinh gọn" (Lean Software Development). Đây là một phương pháp sản xuất phần mềm theo triết lý Agile, sử dụng Tư duy Tinh gọn và các nguyên lý đặc trưng của Tinh gọn (vốn xuất phát từ ngành sản xuất ô tô – Lean Manufacturing) được áp dụng cho lĩnh vực phát triển phần mềm.

Trong chuyến viếng thăm Việt Nam theo lời mời của AgileVietnam.org hai ông bà đã mang đến cho cộng đồng CNTT hai khóa học "Lean Mindset" tại hai thành phố lớn là Sài Gòn và Hà Nội. Khóa học chỉ diễn ra trong duy nhất một ngày. Những nội dung được đề cập tới trong khóa học xoay quanh các vấn đề như: tại sao các công ty lại có khả năng biến những mối nguy hiểm thành thành công? Cách để khám phá ra những gì khách hàng thực sự muốn? Làm thế nào để cung cấp sản phẩm đáng tin cậy hơn bằng cách quản lý công việc? v.v..

Theo đánh giá của nhiều học viên sau khóa học, kỹ thuật Value Stream Mapping (Ánh xạ Dòng Giá trị) là điều thú vị nhất mà họ học được từ khóa học này. Đây là kỹ thuật đã được triển khai nhiều trong "Sản xuất Tinh gọn", nó giúp tìm ra cách thức để gia tăng hiệu quả cho dòng sản xuất.

Khóa học tuy ngắn song nó đã cung cấp cho học viên nhiều ý tưởng mới, triết lý mới để TINH GỌN hơn trong công việc cũng như cuộc sống hằng ngày. Mong rằng sẽ có thêm những khóa học bổ ích như vậy dành cho cộng đồng CNTT tại Việt Nam.

Summer
Coderetreat 2013

Coderetreat là gì?

Coderetreat là một hoạt động được khởi xướng vào khoảng 4 năm trước tại hội thảo về mã nguồn ở Sandusky Ohio. Có rất nhiều những cuộc thảo luận xoay quanh việc chúng ta đã thực hành không đủ. Trong những lĩnh vực có yếu tố sáng tạo, người ta được rèn luyện thường xuyên. Nhưng hầu hết các lập trình viên lại được rèn luyện chủ yếu từ thực tế công việc. Vì vậy họ đã không học tập tốt theo cách thường thấy với khả năng vốn có như khi họ rời xa những áp lực từ thực tế.

Coderetreat lần đầu tiên được tổ chức tại Ann Arbor, Michigan. Sau nhiều nỗ lực, giờ đây Coderetreat đã có định dạng như chúng ta thấy hôm nay: đơn giản, hiệu quả và có tính lan tỏa.

Giờ đây hoạt động này đã trở thành thông lệ với nhiều người vào mỗi dịp cuối tuần.

Hình dung về Coderetreat

Coderetreat là một sự kiện thực hành chuyên sâu, tập trung vào các nguyên tắc cơ bản của phát triển phần mềm và thiết kế. Bằng cách cung cấp cho các nhà phát triển cơ hội để tham gia vào thực hành có chủ ý, tránh xa những áp lực của việc "hoàn thành nhiệm vụ". Coderetreat đã chứng tỏ là một hoạt động có hiệu quả cao giúp cải thiện kỹ năng viết mã nhằm giảm thiểu các chi phí thay đổi theo thời gian.

Sự kiện Coderetreat diễn ra trong thời gian một ngày với các đặc điểm cơ bản như sau:

- Chia phiên làm việc. Mỗi phiên 45 phút.
- Giải bài toán Game of life của Conway.
- Lập trình theo cặp (pair programming).
- Sau mỗi phiên thì đảo cặp.
- Xóa hết mã nguồn sau mỗi phiên.
- Thực hiện cải tiến sau mỗi phiên.

Hoạt động Coderetreat tại Việt Nam

Sự kiện "Global Day of Coderetreat 2012" thu hút hơn 3000 lập trình viên trên toàn thế giới trải dài trên 24 múi giờ và được truyền trực tiếp thông qua công cụ Hangout trên Google+.

Đây cũng là lần đầu tiên Việt Nam tham gia sự kiện này với hai đầu cầu là Hà Nội và Sài Gòn (do HanoiScrum và AgileVietnam đứng ra phối hợp tổ chức), kết nối trực tiếp với các điểm tổ chức khác tại các quốc gia Singapore, Trung Quốc, Nhật Bản.

Sau sự kiện đầu tiên này, ban tổ chức đã nhận được nhiều phản hồi tích cực từ phía người tham dự. Vì vậy, vào mùa hè này, Coderetreat sẽ tiếp tục phủ sóng tại Việt Nam với sự kiện Summer Coderetreat 2013. Phía Hà Nội, HanoiScrum và những thành viên CocoDojo sẽ phối hợp cùng các điểm cầu khác trong cả nước cùng tổ chức một ngày hội dành cho các lập trình viên và những người đã và đang rèn luyện các kỹ năng lập trình chuyên sâu.

Hy vọng với sức nóng của mùa hè, một ngày trọn vẹn với Coderetreat sẽ mang đến cho các lập trình viên những trải nghiệm đáng nhớ.

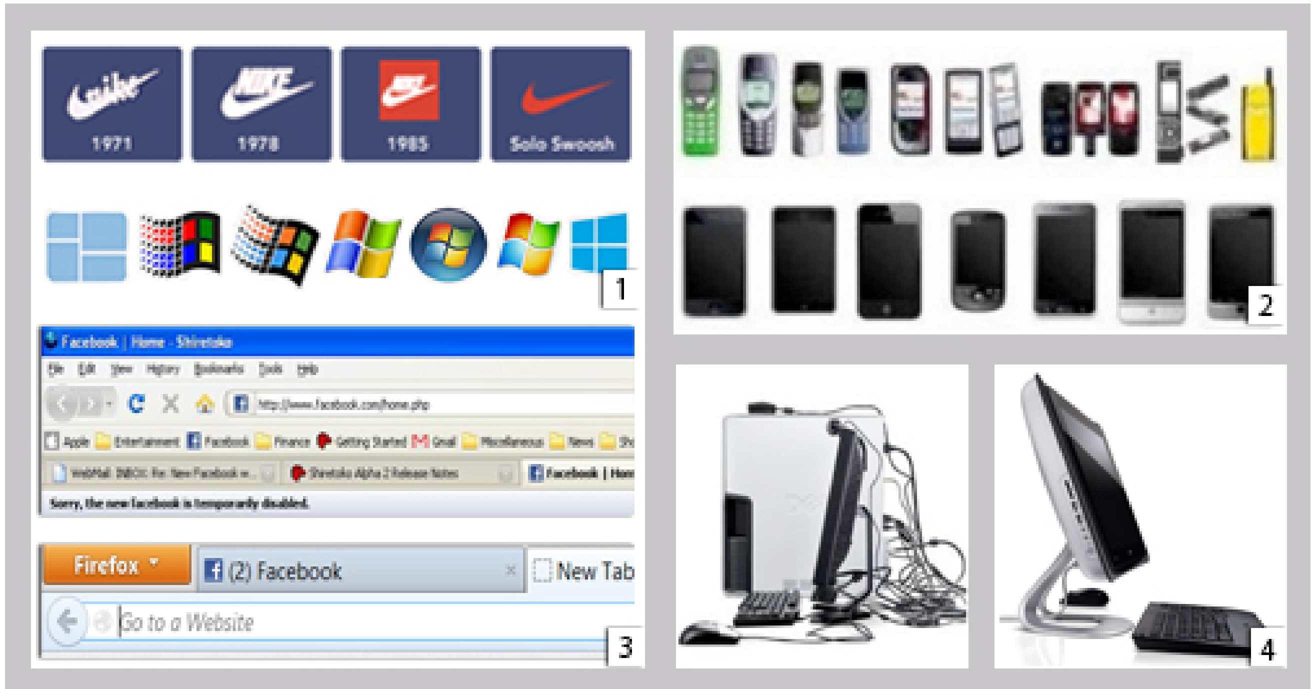
Địa điểm và thời gian chi tiết sẽ được ban tổ chức thông báo đến các bạn tại địa chỉ:

- hanoiscrum.net
- facebook.com/groups/cocodojo

Hẹn gặp tại Summer Coderetreat Việt Nam 2013!

BÀN VỀ SỰ ĐƠN GIẢN

Mê Kim Dung



Nếu bạn thử nhìn vào sự phát triển logo qua các thời kì của Nike, Mercedes, hay là Microsoft Windows, bạn sẽ thấy có một điểm chung: sự đơn giản. Các logo càng ngày càng đơn giản. Xu hướng đơn giản hóa hiện không chỉ ở trong thiết kế đồ họa, mà còn ảnh hưởng rất nhiều tới thế giới công nghệ ngày nay

Xu hướng đơn giản hóa

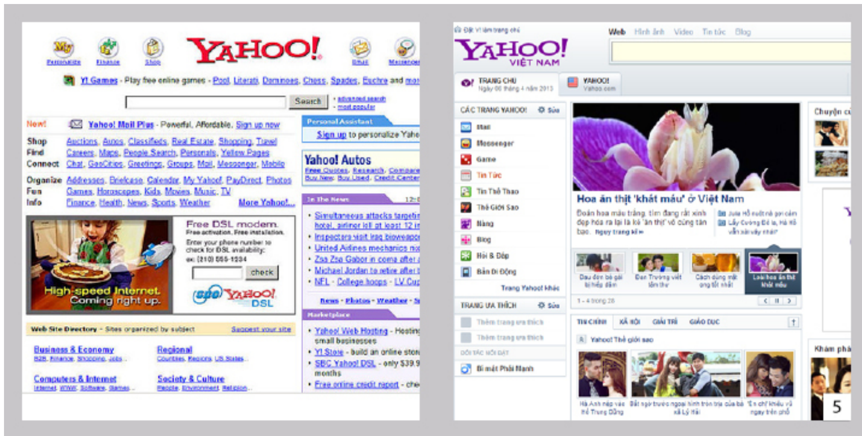
Trước khi iPhone đình đám của Apple xuất hiện, những chiếc điện thoại trông như thế nào? Hình vuông, hình bầu dục, gập, trượt, bút, nhiều nút, bàn phím đầy đủ, đủ loại đường nét góc cạnh, v.v.. Sau khi iPhone ra đời với đúng một nút bấm, hình dáng gần như phẳng lì, toàn đường nét cơ bản, thì điện thoại bây giờ trông như Hình 2.

Các hãng điện thoại đua nhau tìm kiếm những thiết kế đơn giản nhất có thể, nhiều đường thẳng, mặt phẳng, ít nút. Cả máy tính, phần mềm, hay giao diện website (hình 3, 4, 5), tất cả đều ngày một đơn giản hơn.

Ưu điểm của đơn giản

Đơn giản thì dễ dùng. Sản phẩm càng đơn giản càng dễ dùng. Chỉ trừ một số người chuyên sâu hay mày mò khám phá, còn lại đại đa số người sử dụng bình thường thích một sản phẩm động đến là biết cách dùng. Người ta kể rằng Steve Jobs đã yêu cầu đội ngũ kỹ sư thiết kế iPod làm thế nào để người nghe có thể chọn được bài hát ưa thích trong vòng 3 thao tác. Hoặc như Google, ngoài chất lượng tìm kiếm, một phần cũng quan trọng cho sự thành công của họ là thiết kế đơn giản, đơn giản đến mức hầu như bất kì ai cũng có thể sử dụng được mà không phải học hỏi gì nhiều. Kết quả là 70% số người dùng máy MP3 nghe nhạc bằng iPod và trên 80% lượng tìm kiếm Internet dùng Google.

Không chỉ dễ dùng, việc đơn giản hóa sản phẩm giúp cho các nhà phát triển được tập trung vào chất lượng. Những người theo chủ nghĩa đơn giản có câu "Ít hơn tức là nhiều hơn" (Less is More). Cùng một công sức bỏ ra, phần mềm đơn giản hơn, ít chức năng hơn đồng nghĩa với chất lượng mỗi chức năng sẽ nhiều hơn. Hãy nghe lời Steve Jobs khuyên Larry Page, CEO của Google: "Xác định 5 sản phẩm chính của anh là gì, rồi quăng hết những thứ còn lại đi. Những thứ đó chỉ có kìm hãm anh lại thôi". Chúng ta không thực sự biết có phải Larry Page nghe lời Jobs hay không, nhưng danh sách những sản phẩm, dịch vụ mà Google khai tử dưới thời Larry Page cũng khá dài: Bookmarks Lists, Friend Connect, Google Gears,



Trang chủ Yahoo

Google Search Timeline, Google Wave, Knol, CalDAV API, Google Building Maker, Google Cloud Connect, v.v.. Thay vào đó, Google Search, sản phẩm chủ lực của Google, mỗi ngày lại một thêm cải tiến, tiện lợi hơn cho người dùng và doanh thu của Google tăng đều trong mấy năm qua.

Dấu ấn của đơn giản in đậm nét trong tất cả các sản phẩm đặc sắc ở nhiều lĩnh vực khác nhau trong cuộc sống, không chỉ riêng công nghệ. Những chiếc siêu xe không bao giờ nhiều màu, thậm chí đa số chỉ một màu rất cơ bản là đen hoặc trắng. Những bộ trang phục sang trọng không bao giờ tua rua. Những văn bản chuyên nghiệp không có quá hai phông chữ.

Vậy tại sao người ta không làm mọi thứ đều đơn giản? Bởi vì làm đơn giản rất khó.

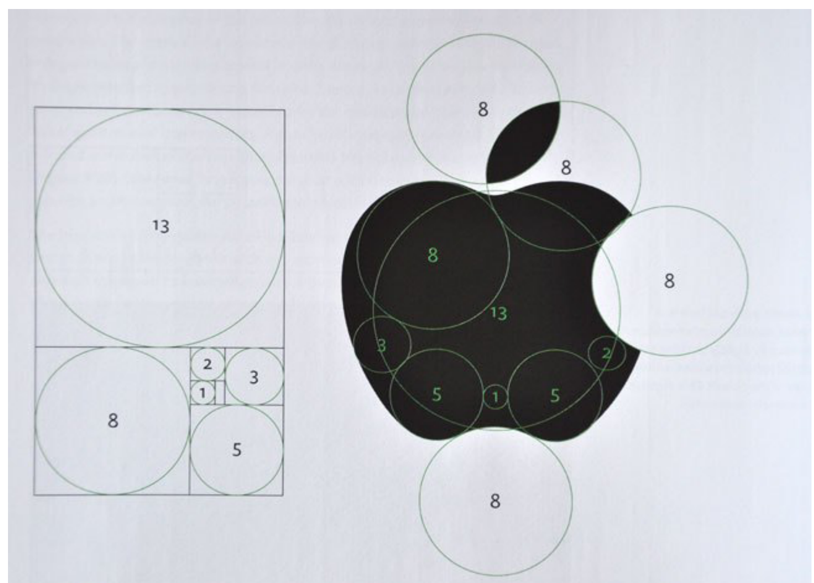
Sự phức tạp của đơn giản

Jonathan Ive, thiết kế trưởng Apple, tác giả những chiếc máy Mac đơn giản mà sành điệu, nói: Để làm ra một thứ đơn giản, người ta phải làm rất nhiều điều phức tạp phía sau.

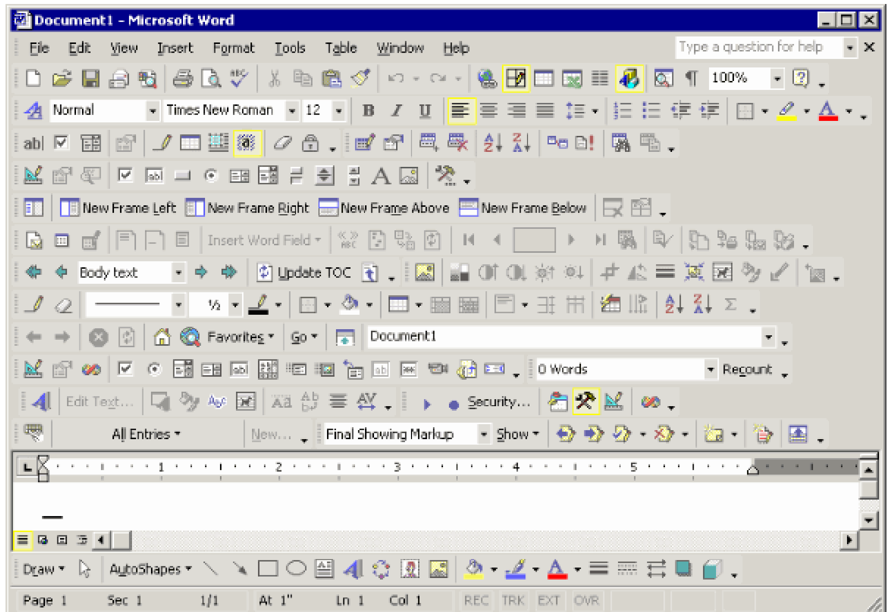
Chúng ta thử nhìn vào logo của Apple, một quả táo cắn dở trông rất vừa mắt được tạo nên bởi vài đường tròn đơn giản, thế nhưng ẩn sau vài cái đường tròn đơn giản đấy là cả một công trình phức tạp, tính toán kĩ càng áp dụng tỉ lệ vàng, tỉ lệ hài hòa nhất trong tự nhiên, như hình dưới.

Vậy đơn giản ở đây phải hiểu là đơn giản phía người dùng, không phải đơn giản cho người làm sản phẩm. Thực tế, càng đơn giản cho người dùng bao nhiêu, càng phức tạp cho người phát triển bấy nhiêu.

Hồi tôi còn học trong trường đại học, làm bài tập lập trình, để chương trình chạy ra được kết quả thì nhanh, nhưng lập trình để cho thầy giáo kiểm tra bài một cách thuận tiện nhất, ít thao tác nhất, trình bày ra màn hình một cách đẹp đẽ nhất có thể thì tốn rất nhiều thời gian và công sức. Tỉ lệ công sức đôi khi đến mức 20-80, với chỉ 20% dành cho việc chương trình chạy ra kết quả.



Thiết kế Logo của Apple



Giao diện Microsoft Office

Việc đơn giản hóa sản phẩm là một việc rất khó mà nhiều khi những hãng khổng lồ cũng không làm nổi. Hãy thử xem một người dùng bình thường muốn xem thông số DNS trong Windows 7 bằng giao diện đồ họa mất bao lâu: 10 cái nhấp chuột. Trong khi Mac OS chỉ mất 4, chưa đến một nửa. Đó là lí do tại sao người ta có câu "một khi đã dùng Mac sẽ không bao giờ muốn quay lại Windows".

Làm thế nào để đơn giản

Điều đầu tiên hãy mạnh dạn gạt bỏ bớt chức năng định làm trong sản phẩm. Chỉ để lại một số ít những chức năng tối cần thiết cho người dùng. Thống kê cho thấy trong một hệ thống thông thường có đến 45% số lượng chức năng không bao giờ dùng đến, 35% ít khi dùng, chỉ có 20% là thường sử dụng. Vậy đừng lo việc lược bớt chức năng sẽ làm phần mềm của bạn không đáp ứng được yêu cầu người sử dụng. Thực tế, nhiều chức năng chỉ làm đội chi phí và thêm bối rối cho phía khách hàng mà thôi.

Hình phía trên vẫn thường được đem ra để chỉ trích việc Microsoft ôm đồm quá nhiều chức năng vào trong Office khiến người sử dụng phải trả thêm tiền cho những chức năng mà họ hầu như không bao giờ động đến.

Hãy điều tra thị trường thật kỹ. Sắp xếp thứ tự ưu tiên các chức năng. Dũng cảm loại bỏ những chức năng không bao giờ hoặc thậm chí ít khi dùng. Ngày trước, khi Apple bỏ ổ đĩa mềm ra khỏi máy Mac, họ cũng bị phê phán rất nhiều. Nhưng chỉ ít lâu sau, ổ mềm đã biến mất sạch khỏi các PC mới xuất xưởng của những hãng sản xuất máy tính lớn còn lại. Thời gian sẽ cho câu trả lời.

Tối giản giao diện. Bỏ tất cả những chi tiết rườm rà. Giấu đi những thanh công cụ, những phím bấm ít khi dùng. Tối đa hóa không gian làm việc cho người sử dụng. Làm như Google khi thiết kế Chrome: "Chúng tôi muốn dành cho người dùng không gian lướt web rộng nhất có thể". Chrome đã bỏ hết các trình đơn truyền thống, gộp thanh tìm kiếm và địa chỉ vào thành một, tận dụng viền cửa sổ (window border) để đặt các tab, thanh trạng thái chỉ hiện lên khi cần. Hãy nhìn sự thay đổi giao diện của Firefox, đối thủ sừng sỏ trong trận chiến trình duyệt, ở trang trước để thấy sự thành công của giao diện tối giản như thế nào.

Thay đổi tư duy. Chú trọng vào chất hơn lượng của sản phẩm. Một trong những tư duy tiêu biểu theo hướng này là đường lối của những nhà phát triển hệ điều hành Unix: Làm đúng một việc, và làm thật tốt việc đó (Do one thing and do it well). Họ không khuyến khích người lập trình làm ra những phần mềm nhiều chức năng, nhiều hiệu ứng. Họ yêu cầu mỗi phần mềm chỉ cần làm đúng một việc và giải quyết rốt ráo tất cả vấn đề xung quanh nó. Copy chỉ làm đúng việc copy, không thêm "cut" hay chức năng khác. Kết quả là các hệ điều hành họ nix có một hệ thống phần mềm cơ bản hoạt động rất ổn định và vô cùng mạnh mẽ.

Để kết thúc, xin lại trích lời "quái kiệt đơn giản" Jonathan Ive: "Làm ra những sản phẩm đơn giản, đơn giản đến mức người sử dụng không hề nhận thấy sản phẩm đã giải quyết được những công việc phức tạp đến thế nào".

Tom và Mary Poppendieck

7 Nguyên lí của Phát triển Phần mềm Tinh gọn

Dương Trọng Tấn

Như là một trong các phương pháp thuộc họ "Phát triển Phần mềm Linh hoạt" (Agile Software Development), Phát triển Phần Mềm Tinh gọn (Lean Software Development, hay gọi tắt là Lean Development) là một phiên bản của phương pháp Sản xuất Tinh gọn (Lean Manufacturing) áp dụng cho lĩnh vực phát triển phần mềm. Thuật ngữ "Lean Software Development" xuất xứ từ cuốn sách cùng tên của Mary Poppendieck và Tom Poppendieck. Cuốn sách diễn dịch lại tư duy Tinh gọn với ý nghĩa mới kèm theo các công cụ hữu hiệu để triển khai thực tiễn.

Trong đó "Bảy nguyên lí" diễn giải Tư duy Tinh gọn trong việc phát triển phần mềm là linh hồn cho quá trình phát tiển phần mềm tinh gọn. Chúng được mô tả ngắn gọn như dưới đây:

1. Loại bỏ lãng phí

Tất cả mọi thứ không tăng thêm giá trị cho khách hàng được coi là lãng phí (Muda). Chúng bao gồm:

- Mã nguồn và chức năng không cần thiết
- Sự chậm trễ trong quá trình phát triển phần mềm
- Yêu cầu không rõ ràng
- Kiểm thử không đủ, để lặp lại quá trình có thể tránh được
- Quan liêu
- Giao tiếp nội bộ chậm chạp

Để có thể loại bỏ lãng phí, người ta phải nhận ra nó. Nếu một số hoạt động có thể được bỏ qua, hoặc kết quả có thể đạt được mà không cần có nó, đó chính là lãng phí. Ví dụ về lãng phí trong phát triển phần mềm có thể kể đến: làm ra các đoạn mã chưa hoàn thành rồi bỏ đi trong quá trình phát triển; các quy trình phụ và các tính năng không thường xuyên được sử dụng bởi khách hàng; chờ đợi các hoạt động khác, các đội khác, quy trình khác; sản phẩm có lỗi và chất lượng thấp; các công việc quản lí không tạo ra giá trị thực sự. Phát triển Tinh gọn tập trung loại bỏ các lãng phí để đạt trạng thái Tinh gọn, tạo điều kiện để đạt hiệu quả tối đa.

Để phân biệt và nhận ra lãng phí, kĩ thuật ánh xạ dòng giá trị được sử dụng. Trong đó, một dòng giá trị bắt đầu khi một đơn vị kinh doanh quyết định rằng thông tin tốt hơn sẽ giúp tăng doanh thu hoặc giảm chi phí. Đây là sự bắt đầu của dòng giá trị. Các dòng giá trị kết thúc khi phần mềm được triển khai bắt đầu tạo thêm doanh thu hoặc giảm chi phí. "Tồn kho" (inventory) trong dòng giá trị phát triển phần mềm là một phần công việc được thực hiện như: yêu cầu không được phân tích và thiết kế, bản thiết kế không được mã hóa, mã nguồn không được kiểm

thử và tích hợp, các tính năng không được triển khai, các tính năng đã triển khai nhưng không tiết kiệm tiền hoặc giảm chi phí. Khi dòng giá trị phần mềm có ít công việc kiểu như thế, rủi ro sẽ được giảm và năng suất được cải thiện rất nhiều. Bước tiếp theo là chỉ ra nguồn lãng phí và loại bỏ nó. Việc này cũng nên được thực hiện lặp đi lặp lại để loại bỏ hết các quy trình và thủ tục không cần thiết.

2. Khuếch trương việc học

Phát triển phần mềm là một quá trình học tập liên tục để tạo ra tri thức mới. Phương pháp tốt nhất để cải thiện một môi trường phát triển phần mềm là khuếch trương việc học tập.

Quá trình học tập được đẩy nhanh tiến độ bằng cách sử dụng các chu kỳ lặp đi lặp lại ngắn – cùng với kĩ thuật tái cấu trúc (refactoring) và kiểm thử tích hợp. Tăng cường thông tin phản hồi thông qua các buổi họp ngắn với khách hàng để xác định tình hình hiện tại và phác thảo những nỗ lực phát triển và điều chỉnh nhằm cải thiện trong tương lai. Trong những phiên làm việc này, cả hai đại diện khách hàng và đội ngũ phát triển tìm hiểu thêm về vấn đề và tìm ra các giải pháp để đi tiếp. Vì vậy, khách hàng hiểu rõ hơn về nhu cầu của họ, dựa trên kết quả hiện có của các nỗ lực phát triển, còn các nhà phát triển thì có thể tìm hiểu làm thế nào để đáp ứng tốt hơn những nhu cầu đó. Thay vì bổ sung thêm tài liệu hoặc kế hoạch chi tiết, các ý tưởng khác nhau có thể được thử nghiệm bằng cách viết mã và tạo ra các bản build. Quá trình thu thập yêu cầu người dùng có thể được đơn giản hóa bằng việc làm các bản mẫu: trình diễn các giao diện trên màn hình (screen) cho người dùng cuối và thu thập dữ liệu từ họ. Một ý tưởng quan trọng khác trong quá trình giao tiếp và học tập với khách hàng là phát triển dựa-theo-lô (set-based development) – kĩ thuật này giúp tập trung vào trao đổi về các ràng buộc của các giải pháp trong tương lai, những ý tưởng nào không thực sự là giải pháp; từ đó thúc đẩy việc đưa ra giải pháp thông qua đối thoại với khách hàng.

3. Quyết định càng muộn càng tốt

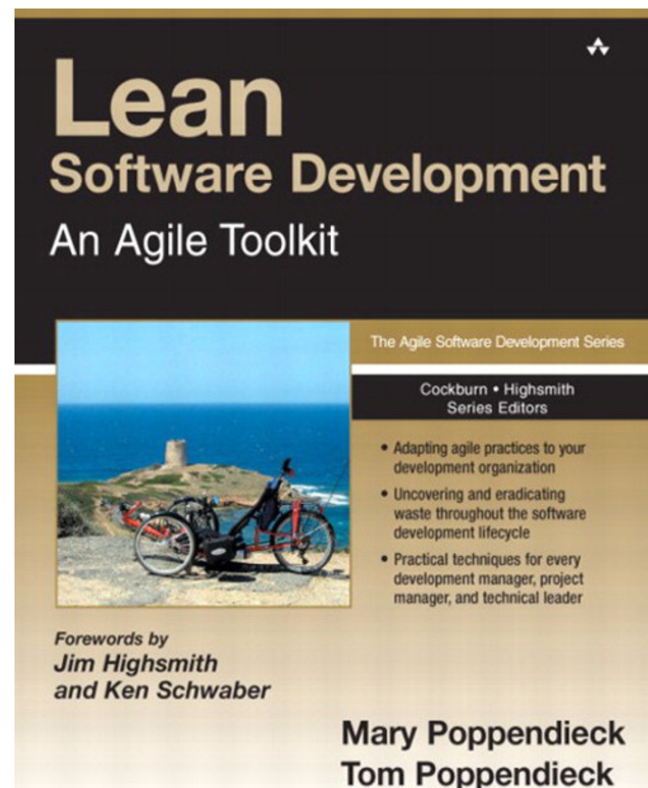
Do việc phát triển phần mềm luôn đi kèm với nhiều yếu tố không chắc chắn, nên để có kết quả tốt hơn, chúng ta phải quyết định dựa trên nhiều lựa chọn (options), trì hoãn quyết định càng lâu càng tốt cho đến khi chúng có thể được thực hiện dựa trên dữ kiện thực tiễn (facts) chứ không phải trên các giả định và dự đoán không chắc chắn. Một hệ thống càng phức tạp, thì càng chứa nhiều khả năng thay đổi, do đó cần thiết phải tạo ra các "độ trễ" cho các quyết định quan trọng. Cách tiếp cận lặp (iterative) thúc đẩy nguyên tắc này – khả năng thích ứng với những thay đổi và sửa chữa những sai lầm, những thứ có thể rất tốn kém nếu được phát hiện muộn, sau khi hệ thống đã được phát hành.

Cách tiếp cận trì hoãn này cũng có thể giúp khách hành nhận ra được những yêu cầu thực sự của họ vốn chỉ tường minh khi họ đã có những trải nghiệm rõ ràng về sản phẩm. Nó cũng sẽ giúp khách hàng loại bỏ các yêu cầu vốn không ổn trong các giả định ban đầu, giúp giảm thiểu lãng phí.

4. Chuyển giao càng nhanh càng tốt

Càng phát hành sớm các gói sản phẩm, bạn càng nhận lại được nhiều phản hồi sớm hơn về sản phẩm, và chúng có thể được đưa vào các cải tiến trong các phân đoạn tiếp theo của quá trình phát triển. Phân đoạn càng ngắn thì quá trình học tập và giao tiếp càng hiệu quả.

Khách hàng thì luôn đánh giá cao việc chuyển giao nhanh chóng các sản phẩm chất lượng vì chúng thường giúp gia tăng tính linh hoạt trong hoạt động của họ. Công ty có thể chuyển giao nhanh hơn tốc độ thay đổi tư duy của khách hàng.



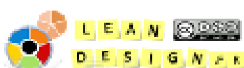
1. Loại bỏ lãng phí
2. Khuyếch trương việc học
3. Quyết định càng muộn càng tốt
4. Chuyển giao càng nhanh càng tốt
5. Trao quyền cho nhóm
6. Tạo ra tính toàn vẹn tự thân
7. Thấy toàn cảnh

Đây là nguyên lí bổ sung cho nguyên lí "Quyết định càng muộn càng tốt": càng chuyển giao nhanh, bạn càng có cơ hội để trì hoãn các quyết định. Ví dụ: bạn có thể thay đổi một chức năng trong một tuần, khi đó bạn không phải quyết định điều gì cho tới tuần sau, trước khi sự thay đổi là thực sự cần thiết.

Nguyên lí này cũng cho phép triển khai "hệ thống kéo" (pull system) trong phần mềm. Theo đó, bạn chỉ phát triển những gì khách hàng đang muốn có, chứ không phải là những gì họ muốn "từ ngày hôm qua", do vậy sẽ giảm được các "lãng phí" không cần thiết (là các chức năng sẽ không được dùng trong thực tế).

5. Trao quyền cho nhóm

Cách tiếp cận tinh gọn ủng hộ việc "tìm người giỏi và để cho họ làm công việc của riêng mình", quy trình khích lệ, tìm lỗi, loại bỏ những trở ngại, và không quản lý vi mô. Công việc chính của các cấp quản lí không phải là chỉ cho các nhà phát triển những gì họ phải làm, mà chủ yếu lắng nghe họ, đưa ra các phân tích, lời khuyên và tìm kiếm các cải tiến. Với niềm tin "tất cả là ở con người", Lean ủng hộ chủ trương động viên và khuyến khích để vươn cao hơn trong công việc của mình. Các nhà phát triển nên được trao quyền tiếp cận khách hàng, các lãnh đạo cung cấp hỗ trợ và giúp đỡ trong các tình huống khó khăn, cũng



PBI = Product Backlog Items PB = Product Backlog MVP= Minimum Viable Product

như đảm bảo rằng sự hoài nghi không hủy hoại tinh thần của đội. Các quyết định đi từ dưới lên (bottom-up) chứ không chỉ là từ cấp lãnh đạo (top-down).

6. Tạo ra tính toàn vẹn tự thân

Hơn cả "chất lượng", tính toàn vẹn (integrity) là điều khiến người dùng muốn dùng sản phẩm. Phần mềm cần được đặt trong một hệ thống đầy đủ để được phát triển cho đúng. Bản thân nó không phải là "cái đích", phần mềm luôn là phương tiện để người dùng đạt được "đích".

Toàn vẹn có hai dạng: toàn vẹn nhận thức (hay toàn vẹn ngoại tại- external integrity) và toàn vẹn khái niệm (hay toàn vẹn nội tại – internal integrity). Trong đó toàn vẹn ngoại tại phản ánh sự nhận thức từ phía khách hàng về sự cân bằng giữa chức năng, tính khả dụng, độ tin cậy và các yếu tố kinh tế.

Còn toàn vẹn nội tại có nghĩa là các thành phần riêng biệt của hệ thống làm việc tốt với nhau như một tổng thể với sự cân bằng giữa tính linh hoạt, khả năng bảo trì, sự hiệu quả và tính đáp ứng. Điều này có thể đạt được bằng sự hiểu biết các miền vấn đề và giải quyết nó cùng một lúc, không phải theo trình tự. Do đó cần có thông tin đầy đủ, đa chiều từ nhiều phía với các hình thức giao tiếp trực diện thay vì các mớ tài liệu dày cộm. Luồng thông tin cần phải là đa chiều: từ nhà phát triển tới khách hàng và phản hồi ngược lại.

Để đạt được toàn vẹn nhận thức, cần phải có được toàn vẹn khái niệm trước đã.

Một trong những cách làm đúng đắn để có được sự toàn vẹn kiến trúc là tái cấu trúc. Cùng với sự gia tăng tính năng là sự gia tăng sự phức tạp và lộn xộn trong cơ sở mã nguồn (codebase). Tái cấu trúc để giữ tính đơn giản (simplicity), rõ ràng, tối thiểu hóa các tính năng trong các đoạn mã nguồn. Sự lặp lại trong mã nguồn là những dấu hiệu cho thấy có thiết kế mã xấu và cần phải tránh. Quá trình xây dựng hoàn chỉnh và tự động nên được đi kèm với một bộ đầy đủ và tự động hóa các kiểm thử mức lập trình (developer testing) và mức khách hàng (customer testing), có quản lý phiên bản (versioning) thống nhất, đồng bộ và có ý nghĩa. Cuối cùng, sự toàn vẹn cần được xác nhận với các kiểm thử kỹ lưỡng để đảm bảo hệ thống có những gì khách hàng mong đợi. Kiểm thử tự động cũng được coi là một phần của quá trình sản xuất, và do đó nếu chúng không có giá trị thì có thể bị xem là lãng phí. Kiểm thử tự động không phải là một mục tiêu, mà là một phương tiện để hoàn thành công việc, đặc biệt là trong việc giảm thiểu các lỗi.

7. Thấy toàn cảnh

Hệ thống phần mềm ngày nay không đơn giản là tổng gộp của các bộ phận, mà còn là sản phẩm của sự tương tác giữa các thành phần đó. Vì vậy cần thiết phải tính đến các tác động toàn cục khi thực hiện các công việc tối ưu hóa cục bộ.

Lỗi trong phần mềm có xu hướng tích lũy trong quá trình phát triển và có liên hệ chặt chẽ với nhau. Cần dừng lại và phân tích các triệu chứng bất ổn khi bắt gặp. Hãy truy nguyên gốc rễ vấn đề. Bằng cách phân rã các nhiệm vụ lớn thành các nhiệm vụ nhỏ hơn, và tiêu chuẩn hóa các giai đoạn phát triển khác nhau, nguyên nhân gốc rễ của các lỗi này cần được phát hiện và loại bỏ.

Tư duy Tinh gọn (Lean Thinking) phải được thấu hiểu bởi tất cả các thành viên của dự án, trước khi triển khai thực hiện trong một tình huống cụ thể, trong thực tế cuộc sống, suy nghĩ. "Hãy suy nghĩ lớn, hành động nhỏ, thất bại nhanh chóng, và học ngay tức thì". Chỉ khi tất cả các nguyên tắc tinh gọn được tuân thủ đầy đủ, kết hợp với các triển khai phù hợp với môi trường làm việc, chúng ta mới có một cơ sở cho sự thành công trong phát triển phần mềm tinh gọn.

10 NGUYÊN TẮC CỦA LẬP TRÌNH TINH GỌN

Mary Poppendieck

Nghiên cứu gần đây đối với các Phương pháp Phát triển Linh hoạt (Agile), Phát triển Phần mềm Thích ứng (Adaptive Software Development), và Lập trình Cực hạn (XP - eXtreme Programming) đều có áp dụng các quy tắc đơn giản của Sản xuất Tinh gọn (Lean Manufacturing) trong phát triển phần mềm. Kết quả là, chúng ta có một thuật ngữ gọi là Lập trình Tinh gọn (Lean Programming), và cũng ấn tượng như những cải tiến trong sản xuất đã mang về các phong trào Just-in-Time và Quản lý Chất lượng Toàn diện (Total Quality Management) của những năm 1980.

Quy tắc 1: Loại bỏ lãng phí

Nguyên tắc đầu tiên của Lập trình Tinh gọn là: Loại bỏ lãng phí. Đó là, loại bỏ bất cứ điều gì mà không tạo ra giá trị gia tăng cho sản phẩm cuối cùng. Tất cả những tài liệu, bản vẽ, mô hình là một phần của dự án phát triển phần mềm, chúng giúp cho dự án hoàn thành, tuy nhiên lại là thành phần không nhất thiết của sản phẩm cuối cùng. Một khi sản phẩm hoàn chỉnh được bàn giao, người dùng cuối ít khi quan tâm đến những tài liệu đó. Các nguyên tắc Lean khuyến nghị rằng tất cả những phụ liệu đó là ứng viên để đưa ra phân tích. Việc phân tích phải đảm bảo không chỉ mang lại các giá trị gia tăng cho sản phẩm mà còn là cách làm hiệu quả nhất để đạt được những giá trị đó.

Quy tắc 2: Giảm thiểu tồn kho (Giảm thiểu thành phần trung gian)

Trong nhà máy của chúng tôi, chúng tôi luôn truyền nhau một thông điệp: Hàng tồn kho là chất thải. Tại sao vậy? Hàng tồn kho tiêu tốn các nguồn tài nguyên, làm chậm thời gian phản ứng. Hàng tồn kho ẩn đi các vấn đề về chất lượng, dễ bị mất, giá trị thấp và dễ lỗi thời. 'Lợi ích' của hàng tồn kho là luôn có nhiều hàng để bán. Nhưng 'chi phí' của hàng tồn kho lại luôn luôn cao hơn 'lợi ích' mà nó mang lại.

'Hàng tồn kho' của phát triển phần mềm là những tài liệu, cái không là một phần của sản phẩm cuối cùng. Ví dụ như tài liệu yêu cầu và thiết kế, thực sự chúng tăng được bao nhiêu giá trị cho sản phẩm cuối cùng? Và chúng quan trọng như thế nào cho sản phẩm? Nếu chúng ta so sánh các tài liệu đó với hàng tồn kho, thì điểm nổi bật cần lưu ý là thời gian cần thiết để tạo ra các tài liệu thiết kế gần bằng với thời gian của chu kỳ dự án. Cũng như hàng tồn kho phải được giảm thiểu để tăng cường tối đa lưu lượng sản xuất,

1. **Loại bỏ lãng phí**
2. **Giảm thiểu tồn kho**
3. **Tối đa hóa luồng**
4. **Kéo từ nhu cầu**
5. **Trao quyền cho người lao động**
6. **Đáp ứng yêu cầu của khách hàng**
7. **Làm đúng ngay từ đầu**
8. **Bãi bỏ tối ưu hóa cục bộ**
9. **Quan hệ đối tác với nhà cung cấp**
10. **Tạo ra văn hóa cải tiến liên tục**

> Nguyên tắc đầu tiên của Lập trình Tinh gọn: Loại bỏ lãng phí, loại bỏ bất cứ điều gì mà không tạo ra giá trị gia tăng cho sản phẩm cuối cùng

các tài liệu phân tích, thiết kế cũng cần được giữ ở mức tối thiểu để tối đa hóa luồng phát triển phần mềm.

Có rất nhiều sự lãng phí ở những tài liệu này. Lãng phí thời gian để viết ra các tài liệu. Lãng phí thời gian để xem xét, rà soát chúng, và những công việc liên quan đến thay đổi các yêu cầu, thiết lập độ ưu tiên và thay đổi hệ thống. Nhưng sự lãng phí lớn nhất đến từ việc xây dựng ra các hệ thống sai nếu các tài liệu không mô tả và phân tích chính xác các yêu cầu của khách hàng.

Phương pháp tốt nhất để giảm thiểu các công việc trung gian là nâng cao mức độ trừu tượng của các tài liệu. Thay vì viết 100 trang tài liệu mô tả chi tiết, hãy viết 10 trang thiết lập các quy tắc và hướng dẫn, và chỉ mô tả chi tiết những trường hợp ngoại lệ. Thay vì đưa ra một tập các đặc tả dày cả chục phân, hãy cung cấp 25 trang ngắn gọn tóm tắt những công việc phải làm.

Chúng ta biết rằng người dùng ít khi hình dung về chi tiết của hệ thống từ các tài liệu kỹ thuật, thậm chí ít có khả năng để nhận thức một cách chính xác làm cách nào hệ thống hoạt động trong môi trường của họ cho đến khi họ thật sự bắt tay vào sử dụng nó. Thậm chí nếu người dùng có thể dự đoán chính xác hệ thống hoạt động thế nào ở thời điểm hiện tại, cũng khó có khả năng nó hoạt động giống như người ta trông đợi trong cả phần đời còn lại của hệ thống. Tất cả những điều đó phải được xem xét khi chúng ta xác định những tài liệu đó có thật sự mang lại giá trị gia tăng cho sản phẩm hay không.

Quy tắc số 3: Tối đa hóa luồng (Giảm thời gian phát triển)

Trong những năm 80, chúng tôi đã học được cách tạo ra sản phẩm trong vòng vài giờ thay vì vài ngày hay vài tuần như trước đây. Chúng tôi thấy được rằng, dòng sản phẩm rất nhanh dẫn tới thời gian chu kỳ rất ngắn, thường là thấp hơn một hai cấp so với trước đây. Trong những năm 90, những dự án thương mại điện tử thường được hoàn thành chỉ trong vài tuần thay vì vài tháng hoặc vài năm như trong thế giới phần mềm truyền thống trước đây. Vâng, có thể có một chút gian dối ở đây. Nhưng mấu chốt là có một số lượng rất lớn các phần mềm được phát triển trong thời gian 5 năm trở lại đây với thời gian phát triển cực ngắn theo các tiêu chuẩn truyền thống.

Trong bài báo gần đây có tiêu đề "Giảm thời gian chu kỳ" (Reducing Cycle Time), Dennis Frailey đề xuất giảm thời gian chu kỳ phát triển phần mềm bằng cách sử dụng kỹ thuật tương tự như trong sản xuất. Ông đề nghị tìm kiếm và giảm việc chồng chéo của việc hiện thời (WIP - Work In Progress). Cũng giống như trong sản xuất, nếu WIP được giảm, thì chu kỳ sản xuất cũng giảm theo. Để giảm WIP, Frailey khuyến cáo sử dụng khái niệm "Lô nhỏ" (Small Batch) và "Nguyên tắc luồng thông suốt" (Smooth Flow Principle), những khái niệm trực tiếp từ Lean Manufacturing.

Phát triển lặp (Iterative development) về cơ bản là việc áp dụng những nguyên tắc này để lập trình. Tiền đề cơ bản của phát triển lặp là "phần nhỏ nhưng đầy đủ" của hệ thống được thiết kế và bàn giao trong suốt chu kỳ phát triển. Với mỗi phân đoạn (iteration), lại thêm một tập hợp các tính năng bổ sung. Thời gian chu kỳ từ khi bắt đầu đến khi kết thúc của mỗi phân đoạn là từ một vài tuần đến một vài tháng. Với mỗi phân đoạn, lại thực hiện các bước từ thu thập yêu cầu đến kiểm thử chấp nhận (acceptance testing).

Quy tắc 4: Kéo từ nhu cầu (Quyết định càng muộn càng tốt)

Trong nhà máy sản xuất băng hình của chúng tôi, chúng tôi từng nghĩ rằng sẽ là lý tưởng nếu phòng kinh doanh có thể dự báo chính xác nhu cầu của thị trường. Rất nhiều công việc cần đưa vào kỹ thuật dự báo tinh vi để có những tiên lượng chính xác hơn trong tương lai. Rồi một ngày, chúng tôi nhận ra rằng, chúng tôi đang cố gắng làm những điều sai. Việc dự đoán chính xác không phải là lý tưởng, mà lý tưởng là chúng ta có thể giảm sự phụ thuộc vào các dự báo bằng cách làm giảm đáng kể thời gian phản ứng để hệ thống có thể đáp ứng với thay đổi, thay vì dự đoán nó.

Trong thị trường công nghệ thay đổi nhanh, đòi hỏi phải liên tục nâng cấp sản phẩm, Dell Computer có một lợi thế rất lớn so với đối thủ cạnh tranh bởi vì họ không dự báo nhu cầu mà phản ứng lại với nhu cầu bằng cách làm theo đơn đặt hàng trung bình trong 6 ngày. Trong khi Dell chỉ nắm giữ hàng tồn kho trung bình 6 ngày, đối thủ cạnh tranh phải duy trì đến 6 tuần. Khả năng quyết định muộn của Dell tạo cho họ một lợi thế cạnh tranh rất lớn trong một thị trường biến động nhanh.

Trong thực hành phát triển phần mềm, hãy giữ sự thay đổi yêu cầu càng gần thời điểm bàn giao sản phẩm càng tốt để có thể cung cấp một lợi thế cạnh tranh đáng kể trong một thị trường biến động. Trong môi trường kinh doanh biến động, người sử dụng không có khả năng dự báo tương lai của họ một cách chính xác. Đóng băng sớm các thiết kế trong dự án phát triển phần mềm là một sự đầu cơ cho việc dự báo. Phần mềm nên được thiết kế để đáp ứng với sự thay đổi, chứ không phải dự đoán nó. Trong phát triển phần mềm, giống như việc sản xuất máy tính, khả năng quyết định càng muộn càng tốt là một lợi thế về cạnh tranh.

Quy tắc số 5: Trao quyền cho người lao động
(Quyết định ở cấp càng thấp càng tốt)

Một nguyên tắc cơ bản của Lean Manufacturing là trao quyền quyết định xuống mức thấp nhất có thể, cung cấp các công cụ và thẩm quyền cho những người "trong cuộc" (on the floor) để đưa ra quyết định. Khi Toyota tiếp quản nhà máy sản xuất của GM ở Fremont, California vào năm 1983, họ phải tiếp nhận những người lao động với năng suất và văn hóa xin nghỉ tồi tệ nhất trong ngành công nghiệp. Cũng những người lao động đó đã có năng suất lao động và chất lượng gấp đôi trong vòng hai năm sau. Điều này được thực hiện thông qua việc hình thành các đội được đào tạo về các kỹ thuật cải tiến và đo lường công việc, và được kỳ vọng để tự phát triển và liên tục cải tiến các tiêu chuẩn làm việc của bản thân.

Một trong những vấn đề của các tài liệu đồ sộ đó là chúng cố gắng đưa ra tất cả các quyết định thay cho các lập trình viên, thay vì cung cấp cho họ một bộ các hướng dẫn. Nói chung, việc tăng cường mức độ trừu tượng của tài liệu sẽ cung cấp chỉ dẫn cho các lập trình viên cũng như cho họ sự tự do để tạo ra các thiết kế chi tiết và các quyết định khi lập trình. Sẽ là tốt hơn nếu nói với các lập trình viên những gì cần phải làm, chứ không phải cách để làm chúng.

"Lập trình viên cần phải hiểu rõ mục đích của công việc và cách thức để nó phù hợp với luồng công việc chung."

Lập trình viên cần phải hiểu rõ mục đích của công việc và cách thức để nó phù hợp với luồng công việc chung. Họ cần biết những gì phải làm để đáp ứng yêu cầu của khách hàng và cần hiểu được kiến trúc và giao diện tiêu chuẩn của hệ thống. Họ cũng cần phải biết những gì họ phải thực hiện, khi nào phải hoàn thành, và có thể dự đoán khi nào nó có thể sẽ kết thúc. Cuối cùng, công việc của họ cần được nhìn thấy trong các phân đoạn nhỏ để cung cấp những thông tin phản hồi cần thiết cho việc cải tiến liên tục.

Quy tắc số 6: Đáp ứng yêu cầu của khách hàng (Bây giờ và trong tương lai)

Trong cuốn "Chất lượng là miễn phí" (Quality is Free), Philip Crosby định nghĩa chất lượng là "đáp ứng yêu cầu của khách hàng". Nghiên cứu của nhóm Standish Group vào năm 1994 lưu ý rằng, nguyên nhân phổ biến nhất của các dự án bị thất bại là do các yêu cầu bị thiếu, không đầy đủ hoặc không đúng. Thế giới phát triển phần mềm đã đối phó với nguy cơ này bằng cách mở rộng việc thu thập yêu cầu người dùng một cách thật chi tiết trước khi tiến hành thiết kế hệ thống. Tuy nhiên việc xác định các yêu cầu của người dùng theo hướng này có những khiếm khuyết trầm trọng.

Tôi đã từng tham gia một dự án trong đó khách hàng muốn một có một hệ thống phức tạp trong vòng 10 tháng. Thời gian là điều tối quan trọng, 10 tháng hoặc là không gì hết. Và tất nhiên, là một cơ quan nhà nước, nên trong hợp đồng yêu cầu phải có một tài liệu thiết kế sơ bộ trước khi có thiết kế chi tiết và viết mã nguồn. Rất nhiều người dùng đã phải tham gia, và rất khó khăn để có chữ ký của họ vào trong tài liệu thiết kế đó. Tại sao? Họ lo ngại rằng họ đang chấp thuận cho một thiết kế mà sau đó sẽ trở thành một sai sót của hệ thống. Bởi vì sẽ không đơn giản để thay đổi các thiết kế sau khi nó được ký chấp thuận. Chúng tôi phải mất 2 tháng để hoàn thành bản thiết kế đó. Và làm sao có thể đổ lỗi cho họ đây? Công việc của họ là phụ thuộc vào việc làm sao để có sự chuẩn xác. Vì vậy, chúng tôi đã lãng phí hơn hai tháng làm việc và rất nhiều giấy tờ để lấy chữ ký của những người dùng.

Thay vì khuyến khích sự tham gia của người dùng, việc ký vào các giấy tờ đó đã tạo ra một môi trường thù địch giữa họ với các nhà phát triển phần mềm. Người dùng bị yêu cầu phải ra quyết định sớm trong quá trình phát triển, và không được phép thay đổi nó, trong khi họ còn không biết hệ thống sẽ hoạt động ra sao, hoặc tình hình kinh doanh của họ có thể thay đổi như thế nào trong tương lai. Có thể dễ dàng hiểu được tại sao họ lại ngại ký vào các bản thiết kế đó, và muốn trì hoãn việc ký càng muộn càng tốt. Lưu ý rằng, bản năng này là phù hợp với quy tắc số 4 đã nêu ở trên.

Cách thức hiệu quả nhất để nắm bắt chính xác các yêu cầu của người sử dụng được tìm thấy trong phương pháp phát triển hệ thống theo cách thức lặp. Bằng cách phát triển các tính năng cốt lõi sớm và có được các thông tin phản hồi của khách hàng trong một buổi demo của mỗi phân đoạn, chúng ta có thể thu thập được các yêu cầu của khách hàng một cách chính xác hơn rất nhiều. Thêm vào đó, nếu chúng ta chấp nhận rằng các yêu cầu có thể thay đổi theo thời gian, chúng ta phải bắt đầu với những yêu cầu quan trọng nhất mà hệ thống cần phải được thiết kế để dễ dàng thích ứng với những thay đổi theo vòng đời của nó.

Quy tắc số 7: Làm đúng ngay từ đầu (Kết hợp với thông tin phản hồi)

Trước khi Lean Manufacturing đến với nhà máy của chúng tôi vào đầu những năm 80 chúng tôi thường sản xuất ra những sản phẩm chất lượng biên (chất lượng được tạo ra từ sự so sánh giữa bad và good quality). Chúng tôi kiểm thử để tìm ra sản phẩm tốt và gia công lại những sản phẩm kém. Sau khi hiểu được nguyên tắc "Làm đúng ngay từ đầu", chúng tôi đã đóng cửa các trạm gia công và thôi không kiểm tra chất lượng của sản phẩm nữa. Thay vào đó, chúng tôi đảm bảo rằng mỗi thành phần của sản phẩm được sản xuất ra đều tốt trước khi được bàn giao sang bộ phận khác. Điều này bao gồm việc kiểm tra và kiểm soát tại tất cả các điểm sản xuất để phát hiện ra các bộ phận lỗi và dừng ngay việc sản xuất của bộ phận đó lại khi lỗi được phát hiện.

"Làm đúng ngay từ đầu" không có nghĩa là "Đóng

băng các Đặc tả của sản phẩm". Ngược lại, các đặc tả này thay đổi liên tục. Và nguyên tắc "lean" nghĩa là khả năng thích ứng một cách hoàn hảo với điều kiện thị trường thay đổi. Điều này được thực hiện thông qua việc tạo ra một kiến trúc sản phẩm cho phép sự thay đổi trong sản xuất, các kỹ thuật giám sát nhằm phát hiện lỗi trước khi nó xảy ra, và có thể kiểm tra ngay lập tức những thứ đã được thiết kế trước khi đưa vào sản xuất.

Vào năm 1987, Barry Boehm đã quan sát thấy rằng, chi phí để tìm và sửa lỗi sau khi phần mềm được phân phối đắt gấp 100 lần so với khi lỗi được tìm và sửa trong giai đoạn thiết kế ban đầu. Sự quan sát này cùng với quy tắc "Làm đúng ngay từ đầu" đã được sử dụng rộng rãi để biện minh cho việc tăng chi phí thiết kế hệ thống chi tiết trước khi viết mã.

Vấn đề nằm ở chỗ, người ta giả định rằng có thể tạo ra bộ tài liệu mô tả chi tiết, chính xác yêu cầu của khách hàng, và những yêu cầu này sẽ không thay đổi. Nhưng thực tế thì các yêu cầu thường xuyên thay đổi trong vòng đời của hầu hết các phần mềm. "Làm đúng ngay" đã bị diễn giải sai thành "không cho phép thay đổi". Một khi chúng ta thừa nhận thay đổi là một yêu cầu cơ bản của khách hàng, điều này trở nên rõ ràng rằng "làm đúng ngay" đòi hỏi chúng ta cần chuẩn bị cho sự thay đổi.

Nếu chúng ta muốn đáp ứng yêu cầu của khách hàng, chúng ta phải thừa nhận rằng khách hàng không thực sự biết họ muốn gì trong giai đoạn đầu của sự phát triển. Vì thế chúng ta cần phải kết hợp với phương pháp lấy phản hồi của khách hàng trong suốt quá trình phát triển. Thay vào đó, hầu hết các hoạt động phát triển phần mềm đều bao gồm một quy trình "kiểm soát thay đổi", điều này làm cho việc đáp ứng các phản hồi của khách hàng trở nên rất khó khăn, khiến cho các nhà phát triển rất ngại hỏi. Những công việc này thực chất làm cản trở sự thay đổi, và không hề chú trọng đến việc đảm bảo chất lượng, đã khiến cho việc "làm đúng ngay" bị hiểu sai.

Lean Programming sử dụng hai kỹ thuật quan trọng để cho việc thay đổi trở nên dễ dàng. Cũng như Lean Manufacturing đưa hệ thống kiểm thử vào trong từng quá trình để phát hiện lỗi, Lean Programming cũng xây dựng những kiểm thử trong quá trình phát triển để đảm bảo rằng sự thay đổi sẽ không vô tình phá hỏng những mã nguồn đã viết trước. Thực tế, cách tốt nhất là viết các bài kiểm thử trước, và sau đó mới viết mã nguồn. Nếu kiểm thử đơn vị (Unit testing) và kiểm thử hồi quy (Regression testing) tốt sẽ tạo điều kiện cho những thay đổi trong yêu cầu ở cuối quá trình phát triển.

Kỹ thuật thứ hai cho phép sự thay đổi muộn trong quá trình phát triển là tái cấu trúc (Refactoring), hay còn gọi là cải tiến thiết kế của những phần mềm đã có sẵn bằng một cách thức nhanh chóng và có kiểm soát. Trong khi refactoring là một phương pháp đã được chấp nhận, những thiết kế cần tập trung từ sớm vào những vấn đề có sẵn hơn là ngồi suy đoán những thành phần mở rộng sẽ cần đến trong tương lai. Khi các tính năng bổ sung được thực sự đưa vào, refactoring cung cấp một thiết kế mới, đơn giản để xử lý những vấn đề mới xuất hiện. Một khi refactoring trở thành một phần trong quy trình, chúng ta sẽ giảm bớt sự suy đoán về những gì cần thiết trong tương lai bằng cách làm cho những thứ sẽ đến trong tương lai dễ dàng thích ứng với hệ thống.

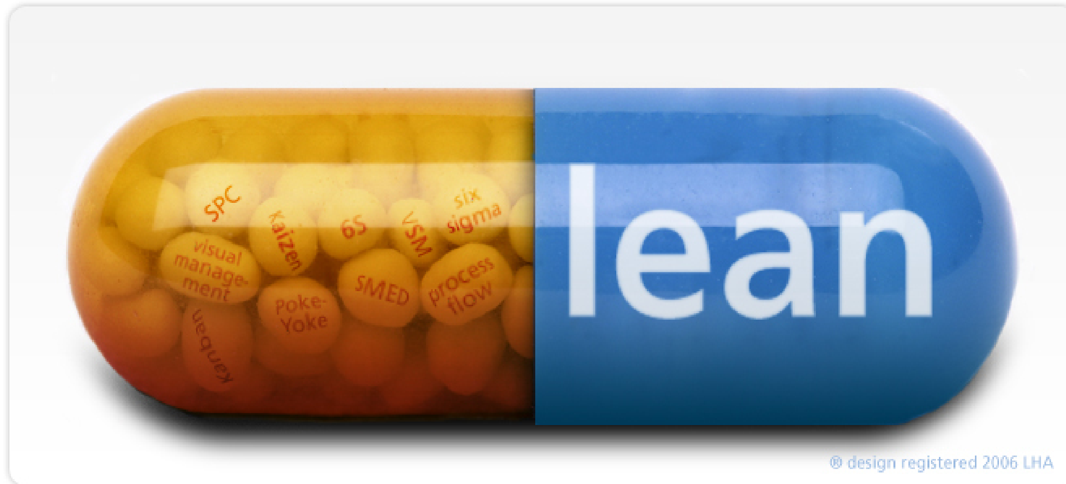
Quy tắc số 8: Bãi bỏ tối ưu hóa cục bộ (Đo lường tối ưu cục bộ là kẻ thù)

Trong những năm 80, kẻ thù lớn nhất của Lean Manufacturing chính là bộ phận kế toán. Chúng tôi đã có những máy móc lớn và rất đắt tiền trong nhà máy, và ý nghĩ rằng chúng không được sử dụng hết công suất là rất cực đoan (nói một cách nhẹ nhàng). Chúng tôi soạn ra các báo cáo kê khai công việc hàng ngày, và các kế toán không muốn những báo cáo đó bị bỏ xó bởi vì ở đó không có bất cứ một WIP (Work in Progress) để báo cáo.

Cả một thế hệ kế toán đã phải nghỉ hưu trước khi việc các máy móc được phép hoạt động dưới công suất được chấp nhận. Thiết kế các máy móc để cho phép việc chuyển đổi nhanh chóng thay vì công suất cao nhất cao nhất vẫn còn là một vấn đề khó thuyết phục thậm chí là đến tận bây giờ. Sau 20 năm, Lean Manufacturing vẫn bị coi là phản trực quan đối với những người thiếu cái nhìn rộng trong môi trường kinh doanh.

Trong bối cảnh đó, chúng ta hãy xem xét vai trò của quản lý phạm vi (scope) trong dự án phát triển phần mềm. Những nhà quản trị dự án đã được đào tạo để tập trung vào việc quản lý phạm vi, cũng giống như chúng ta được đào tạo để tập trung vào việc tối đa hóa năng suất thiết bị. Tuy nhiên, Lean Programming về cơ bản được định hướng bởi thời gian và các phản hồi. Việc tối ưu hóa năng suất cục bộ làm cho quá trình tổng thể được tối ưu hóa, và cũng tương tự như vậy, việc tập trung vào quản lý phạm vi sẽ giúp cho cả quá trình quản lý dự án tổng thể được tối ưu hóa.

Khi nghĩ về điều đó, việc giữ các phạm vi sao cho nó chính xác với những gì người ta hình dung về nó



từ đầu thì ít có giá trị trong môi trường kinh doanh luôn thay đổi. Trong thực tế, việc này lại tăng thêm sự lo lắng và làm tê liệt việc ra quyết định. Nó không có nhiều giá trị cho hệ thống cuối cùng, mà những hệ thống đó thông thường đã bị lỗi thời trước khi chúng được bàn giao. Việc quản lý những mục tiêu mà không còn giá trị sẽ là lãng phí thời gian và tạo ra một danh sách rất dài các vấn đề, gây khó khăn cho sự thỏa hiệp, và khó khăn cho việc sửa chữa hệ thống để nó đáp ứng đúng vào giai đoạn cuối. Tuy nhiên, việc giữ cho hệ thống chạy theo đúng phạm vi ban đầu lại là một mục tiêu chủ chốt nhất của quản trị dự án, việc đo lường này sẽ vẫn tiếp tục được tối ưu hóa trong giá trị tổng thể mà dự án đem lại.

Phạm vi sẽ biết tự điều chỉnh nếu lĩnh vực của nó được hiểu cặn kẽ, và có một thỏa thuận mức cao về những gì hệ thống sẽ làm trong lĩnh vực đó. Phạm vi sẽ tự điều chỉnh nếu như cả hai phía cùng tập trung vào việc phát triển nhanh và giải quyết các vấn đề thực tế của người dùng, và thông qua các phương pháp loại bỏ dư thừa để đạt được các mục tiêu đó.

Quy tắc số 9: Quan hệ đối tác với các nhà cung cấp (sử dụng mua sắm tiến hóa)

Lean Manufacturing không chỉ duy trì trong nhà máy sản xuất. Một khi ý tưởng về việc hợp tác với các nhà cung cấp được kết hợp với sự hiểu biết về giá trị của dòng sản phẩm nhanh chóng (rapid product flow). Quản lý Chuỗi Cung ứng (Supply Chain Management) đã được ra đời. Người ta bắt đầu nhận ra rằng nó đã sử dụng hàng tấn giấy tờ cho việc di chuyển vật liệu giữa các công ty, và điều này không hề tạo thêm giá trị cho sản phẩm. Hơn nữa, các công việc giấy tờ thường tốn kém hơn nhiều so với người ta nghĩ, chưa kể đến sự chậm trễ trong dòng sản phẩm mà nó gây ra. Thậm chí ngày nay, dự đoán rằng hàng tỷ đô la có thể được tiết kiệm từ cổng thông tin kinh doanh dựa vào việc cắt giảm chi phí giao dịch cần thiết để di chuyển hàng hóa giữa các công ty.

Quản lý Chuỗi Cung ứng dẫn tới các công ty phải để ý hơn tới các hợp đồng với các công ty khác. Và các hợp đồng này thường xuyên dùng để giữ cho các công ty không gian lận lẫn nhau. Thêm vào đó, nó còn khiến cho nhà cung cấp đối đầu nhau để làm sao họ đạt được việc cung ứng với chi phí thấp nhất. Thêm một lần nữa, Lean Manufacturing biến đổi mô hình này. Demming đã dạy rằng, việc tin tưởng mối quan hệ với một nhà cung cấp duy nhất tạo ra một môi trường cho phép việc tối ưu hóa giá trị tổng thể cho cả hai công ty.

Trong suốt những năm 1980, các công ty đã đạt được các sản phẩm với chất lượng cao nhất và chi phí thấp nhất trong chuỗi cung ứng bằng việc giảm số lượng của các nhà cung cấp và chỉ làm việc với một số nhà cung cấp như là đối tác. Chất lượng và sự sáng tạo từ việc kết hợp với chuỗi cung ứng đã được chứng minh vượt xa những lợi ích đến từ môi trường tranh chấp và doanh thu nhanh chóng của các nhà cung cấp. Hợp tác với các công ty giúp cả hai bên cải thiện thiết kế sản phẩm và các dòng chảy sản phẩm. Chúng liên kết các hệ thống để cho phép sự vận chuyển hàng hóa ngay lập tức giữa các nhà cung ứng với rất ít các công việc liên quan đến giấy tờ. Những lợi ích lâu dài của việc quan hệ hợp tác với các nhà cung ứng đã được tài liệu hóa.

Các công ty khôn ngoan nhận ra rằng, các hợp đồng phát triển phần mềm truyền thống sinh ra các lãng phí ẩn. Năm 1980, các nhà sản xuất phát hiện ra rằng

các mối quan hệ đối tác với chỉ một vài nhà cung ứng có thể tạo ra rất nhiều lợi ích. Không có mối quan hệ thù địch được tạo ra bằng cách tập trung kiểm soát phạm vi và chi phí liên tục, các nhà cung cấp và phát triển phần mềm có thể tập trung vào việc cung cấp những phần mềm tốt nhất có thể cho khách hàng, sửa chữa yêu cầu càng muộn càng tốt trong quá trình phát triển và cung cấp ra rất nhiều giá trị với giá phải chăng.

Quy tắc số 10: Tạo ra văn hóa cải tiến liên tục

Khi việc phát triển phần mềm có vẻ như không kiểm soát nổi, một cách để giải quyết là tăng mức độ "trưởng thành phần mềm" của tổ chức. Dường như điều này cũng giống như việc có cách thức để sản xuất tốt, khi mà chứng nhận ISO 9000 và giải thưởng Malcom Baldridge thỉnh thoảng bị đánh đồng với sự xuất sắc. Tuy nhiên những chương trình tài liệu hóa quy trình này được xác định xuất sắc chỉ khi trong thực tế áp dụng chúng thực sự xuất sắc.

Trong rất nhiều dự án phát triển phần mềm hiện tại, xuất sắc có nghĩa là khả năng thích ứng với biến động và sự thay đổi nhanh chóng của môi trường. Các phương pháp Tiếp cận Quá trình Chuyên sâu (Process-intensive) như các cấp độ cao trong CMM (Capability Maturity Model của Software Engineering Institute – SEI) có thể thiếu sự linh hoạt để đáp ứng nhanh chóng với thay đổi. Trong một email tư vấn gần đây từ Cutter Consortium, Jim Highsmith đã nhấn mạnh về sự giằng co giữa các phương pháp nặng nề và phương pháp nhẹ nhàng như Lean Programming.

Câu hỏi đặt ra là liệu có phải các chương trình chứng nhận tài liệu hóa quy trình bị bóp nghẹt thay vì nuôi dưỡng một văn hóa cải tiến liên tục? Deming có lẽ sẽ nằm trong mộ xem xét và viết những pho sách về quy trình thay thế cho phương pháp đơn giản của ông: Lên kế hoạch – Thực hiện – Kiểm tra - Hành động (Plan – Do – Check – Act).

Plan: Chọn một vấn đề, phân tích nó để tìm ra nguyên nhân có thể.

Do: Chạy một thử nghiệm để điều tra nguyên nhân có thể xảy ra.

Check: Phân tích dữ liệu từ thí nghiệm để kiểm chứng nguyên nhân.

Act: Tinh chỉnh và chuẩn hóa dựa trên kết quả.

Phát triển lặp cho phép sử dụng phương pháp Plan-Do-Check-Act trong một dự án. Trong suốt phân đoạn đầu tiên, việc chuyển giao từ thiết kế sang lập trình hoặc từ lập trình đến kiểm thử có thể chưa được trơn tru. Điều đó không vấn đề gì nếu như phân đoạn đầu tiên cung cấp một kinh nghiệm học tập cho các nhóm dự án, bởi vì còn có những phân đoạn tiếp theo, cả nhóm có thể cải tiến quy trình. Theo một cách nào đó, một môi trường dự án lặp trở thành một môi trường hoạt động, bởi vì quá trình này được lặp lại trong kỹ thuật cải tiến quy trình của Deming có thể được áp dụng từ phân đoạn này tới phân đoạn khác.

"Thực hành phát triển phần mềm theo Lean sẽ mang lại những phần mềm chất lượng cao nhất, chi phí thấp nhất, thời gian triển khai ngắn nhất có thể."

Cải tiến sản phẩm cũng có thể lặp đi lặp lại, đặc biệt nếu áp dụng tái cấu trúc. Trong thực tế, tái cấu trúc cung cấp một phương tiện rất tốt để áp dụng các nguyên tắc cải tiến liên tục vào trong môi trường lập trình.

Tuy nhiên, việc cải tiến cần được thực hiện vượt ngoài phạm vi một dự án. Chúng ta phải cải tiến các dự án trong tương lai bằng cách học từ các dự án đã qua. Một lần nữa, Lean Manufacturing có thể chỉ ra con đường. Trong suốt những năm 80, một bộ các phương pháp thực hành đã được tổng kết lại trong 10 nguyên tắc của Lean và được áp dụng rộng rãi trên hầu hết các nhà máy ở phương Tây. Những phương pháp này sau đó lan sang các tổ chức dịch vụ, các tổ chức logistics, chuỗi cung ứng, và xa hơn nữa. Chúng đã được thử thách trong rất nhiều lĩnh vực.

Các nguyên tắc đơn giản của Lean Manufacturing đã mang lại những cải tiến đáng kể trong nhiều ngành công nghiệp. Những nguyên tắc này có thể và nên được ứng dụng trong phát triển phần mềm. Thực hành phát triển phần mềm theo Lean sẽ mang lại những phần mềm chất lượng cao nhất, chi phí thấp nhất, thời gian triển khai ngắn nhất có thể.

Khái niệm

Tái cấu trúc (refactoring) theo định nghĩa của Martin Fowler:

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

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

Hiệu quả của tái cấu trúc

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

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

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

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

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

Thời điểm thực hiện

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

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

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

Các "mã bẩn" thường gặp

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

- Mã lặp
- Hàm dài
- Lớp lớn
- Hàm có nhiều tham số đầu vào
- Tính năng không phải của lớp
- Lớp có quan hệ quá gần gũi
- Lớp quá nhỏ
- Lệnh switch
- Từ chối kế thừa
- Định danh quá dài hoặc quá ngắn
- Dùng quá nhiều giá trị

Các kỹ thuật tái cấu trúc cơ bản

Các kỹ thuật tái cấu trúc được chia thành các nhóm tùy theo tiêu chí. Ở đây chúng ta sẽ chia theo mục đích và các bạn có thể tìm hiểu nội dung chi tiết trên tapchilaptrinh.wordpress.com

- Trừu tượng hóa
- Chia nhỏ mã
- Chuẩn hóa mã

Kết luận

Cải tiến là công việc cần thiết và đem lại hiệu quả cao. Nhưng số lượng các kỹ thuật mà ta áp dụng được cho dự án của mình thì rất nhiều và nó cũng gây tốn kém thời gian. Trong bài viết này tôi không hy vọng sẽ trình bày được tất cả hay nhiều kỹ thuật mà chỉ là một số kỹ thuật căn bản nhất.

Hãy tinh gọn trong học tập và cuộc sống

Nguyễn Ngọc Anh

Sau khi tham dự Lean Mindset Workshop của ông bà Mary & Tom tại ĐH FPT tôi hiểu rõ hơn tầm quan trọng của hai chữ "tinh gọn" trong cuộc sống ngày nay. Tôi nghĩ để thành công và phát triển trong thời nay thì tính tinh gọn mang ý nghĩa quyết định, mọi sự cồng kềnh và phức tạp đều gây cản trở, lạc hậu. Tôi xin đề cập đến một số nguyên lý cơ bản của "Phát triển tinh gọn" (Lean Development) và sự liên hệ với học tập và cuộc sống.

Loại bỏ lãng phí

Lãng phí là bất cứ thứ gì không đem lại giá trị. Nói về sự lãng phí, tôi nhớ lại câu chuyện: một người thầy bỏ các viên sỏi to vào một cái lọ, rồi hỏi các sinh viên cái lọ đã đầy chưa. Ai cũng bảo là đầy nhưng thầy tiếp tục cho thêm vào các viên sỏi nhỏ, rồi các hạt cát và cuối cùng vẫn đổ thêm nước vào được. Nếu chúng ta coi viên sỏi to là những thứ có giá trị nhất với cuộc sống của mình vậy hãy bỏ nó vào trước, bằng không sự lãng phí như những hạt cát nhỏ đã nằm sẵn trong bình làm cho bạn không cho vào được dù chỉ là một viên sỏi.

Muốn loại bỏ sự lãng phí thì chúng ta phải có cách nhìn ra nó. Hãy xác định những thứ quan trọng và có giá trị nhất với mình, coi đó là các nguyên tắc sống để làm kim chỉ nam cho mọi hành động của mình.

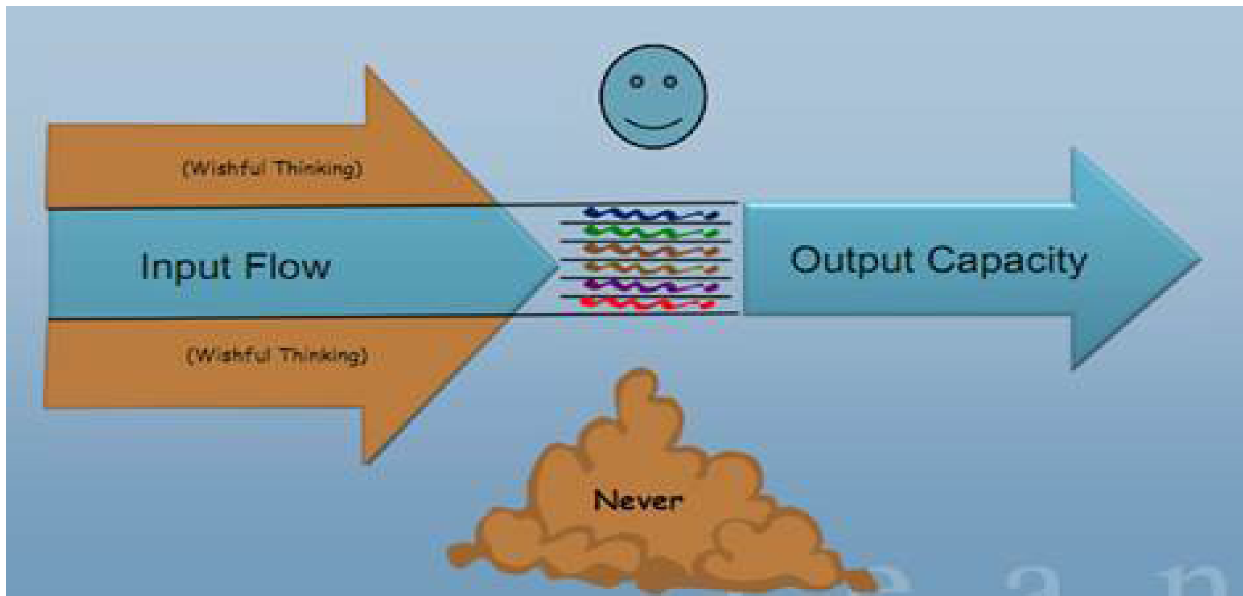
Những lãng phí

Sự lãng phí lớn nhất là sử dụng thời gian không hiệu quả. Trong đó kẻ thù chính là sự lười biếng. Theo tôi ai cũng lười, chúng ta càng đầy đủ thì càng lười. Nhưng mọi sự thành công đều trả giá bằng sự cố gắng tột bậc, vậy nếu bạn không quản lý được thời gian của mình và luôn vướng vào những chuyện vô bổ không đem lại giá trị thì đó là một sự thất bại, và hãy dành sự ngưỡng mộ cho người thành công. Tất nhiên nói và làm là hai chuyện khác nhau. Theo tôi bạn nên chọn một phần kiến thức bạn cho là quan trọng nhất, hãy bắt đầu bằng phần dễ nhất, làm và nhìn thấy kết quả sẽ là sự động viên cho bạn tiếp tục, sự hứng thú và đam mê xuất hiện sẽ đẩy lùi dần sự lười biếng. Hãy tham gia vào các hoạt động thú vị như Coding Dojo, nơi bạn được code, được cười và được chia sẻ. Nếu muốn thành Lý Tiểu Long trong ngành phần mềm bạn phải bắt đầu như thế, vì để thành chuyên gia, số giờ luyện tập phải tính bằng hàng nghìn.

Sự lãng phí còn xảy ra khi chúng ta muốn làm hoặc học quá nhiều thứ cùng lúc mà không thứ nào thật sự tập trung và quyết tâm. Theo tôi đây là biểu hiện của delay, extra features, queue, task switching… dẫn đến các công việc đều chỉ được làm một phần (partially done) và sự hỏng hóc, nông cạn (defect) trong nhận thức xuất hiện. Hãy lập ra danh sách

"Chúng tôi xây dựng con người trước khi xây dựng những chiếc xe"

~ Toyota~



ưu tiên những việc quan trọng và chỉ làm rất ít việc trong số đó tùy theo khả năng của mình.

Học tập không ngừng

Tôi ngưỡng mộ cuộc sống của những người như ông bà Mary và Tom. Tóc đã bạc trắng nhưng lịch làm việc của họ thật đáng kinh ngạc. Ngoài những ngày đi ngao du khắp thế giới để "truyền đạo" là viết, viết và viết. Bao năm rồi họ sống như vậy, càng đi nhiều, viết nhiều, dạy nhiều họ lại càng giỏi. Sự đóng góp cho cộng đồng là khó để có thể kể hết và phần thưởng họ nhận được cũng thật xứng đáng.

Theo tôi ai đó coi việc học là gánh nặng và trách nhiệm để đạt mục đích trong cuộc sống thì khó có thể thành công, nếu có chỉ là cái vỏ bọc bên ngoài. Việc học tập suốt đời một cách thật sự, liên tục làm cho bạn trở nên có ích, có giá trị, có bản sắc và cuộc sống tốt đẹp.

Vậy làm thế nào để "học tập suốt đời" được hiệu quả? Theo tôi trước hết ta cần biến việc học thành niềm đam mê. Trong các loại đam mê tôi đánh giá đam mê tri thức và sự hiểu biết là đáng trân trọng và bền vững nhất. Quá trình học tập nên được sắp xếp thành các chu kỳ lặp đi lặp lại ngắn, mỗi chu kỳ bao gồm: học -> áp dụng -> nhận phản hồi -> cải tiến. Điều này áp dụng đúng tư tưởng tích hợp liên tục (continuous integration) của Agile, làm cho khách hàng (là chính bạn) hài lòng.

Tôn trọng con người

"Chúng tôi xây dựng con người trước khi xây dựng những chiếc xe" – Toyota

Con người là cốt lõi của sự thành công trong bất kỳ công ty nào. Điều này đã được khẳng định ở những công ty hàng đầu như Toyota, Google, Facebook,… Một chuyên gia thật sự có sức làm việc và sáng tạo bằng cả trăm người hoặc hơn nữa. Vậy làm sao để trở thành chuyên gia trong lĩnh vực nào đó? Chỉ bằng cách luyện tập có chủ ý (Deliberate Practice) thường xuyên và liên tục bạn mới thành chuyên gia được.

Trong luyện tập có chủ ý bạn cần:

- Xác định một kỹ năng cụ thể cần cải tiến
- Thiết kế bài tập phù hợp
- Luyện tập thường xuyên và cật lực
- Nhận phản hồi sớm nhất có thể và điều chỉnh.

Trong ngành phần mềm, Coding Dojo là một phương pháp hay để luyện tập có chủ ý. Ở đó mọi người cùng luyện tập các "bài quyền" trong sự thoải mái và như vậy kiến thức được lan truyền.

Tóm lại mấy điểm chính về tinh gọn trong học tập là: loại bỏ lãng phí, học tập suốt đời để trở thành chuyên gia, luyện tập có chủ ý không ngừng. Trên đây là những suy nghĩ của tôi về tính "tinh gọn" dưới lăng kính của một người làm nghề giáo. Hi vọng có thể sẽ giúp ích cho ai đó biết phải bắt đầu như thế nào và làm gì để thành công.

Martin Fowler
Phạm Anh Đới (*lược dịch*)

Martin Fowler là diễn giả, nhà tư vấn và tác giả của rất nhiều sách có ảnh hưởng lớn trong phát triển phần mềm, thiết kế và phân tích hướng đối tượng, UML, mẫu thiết kế, các phương pháp phát triển phần mềm linh hoạt, và cả XP. Thấy được những băn khoăn của nhiều người mới bắt đầu thực hành XP và Agile về vai trò của thiết kế nên chúng tôi quyết định dịch một bài viết rất quan trọng của ông có tên "Có phải thiết kế đã chết?"

Nhiều người mới tiếp xúc với Extreme Programming (XP) cho rằng XP muốn giết chết thiết kế phần mềm. XP không chỉ chế nhạo việc thiết kế là "Big Up Front Design" (thiết kế hoàn hảo trước, viết mã chương trình sau) mà cả những kỹ thuật như UML, khung làm việc (framework) linh hoạt, mẫu thiết kế (pattern) cũng không được nhấn mạnh, thậm chí gần như là bỏ qua. Thực tế XP chứa rất nhiều thiết kế, nhưng ở một cách khác so với quy trình phát triển phần mềm đã xác lập. XP làm trẻ lại ý niệm về thiết kế tiến hóa bằng những kỹ thuật để chiến thuật thiết kế tiến hóa có thể thành công. XP cũng đưa ra những thử thách và kỹ năng mới mà một nhà thiết kế phải học, đó là làm theo thiết kế đơn giản, dùng tái cấu trúc để giữ cho thiết kế trong sáng và sử dụng mẫu theo cách tiến hóa.

Thiết kế có kế hoạch và thiết kế tiến hóa

Bản chất, thiết kế tiến hóa được xây dựng cùng với cài đặt hệ thống. Trong những trường hợp thường gặp, thiết kế tiến hóa là thảm họa. Thiết kế cuối là một mớ các thiết kế rời rạc của những quyết định cục bộ. Thiết kế có kế hoạch được sinh ra từ đây, nó chứa một quan điểm ở một nhánh khác của kỹ thuật. Thiết kế được hoàn thành ở một đơn vị, sau đó sẽ được giao cho đơn vị khác để cài đặt.

Ở nhiều điểm, cách tiếp cận này tốt hơn thiết kế tiến hóa viết mã và sửa. Nhưng nó cũng có một số vấn đề như nhà thiết kế không thể nghĩ ra hết mọi vấn đề mà bạn phải đối mặt khi lập trình hoặc khi thay đổi yêu cầu. Việc đưa tính mềm dẻo vào thiết kế giúp bạn có thể dễ dàng thay đổi, nhưng chỉ trước những thay đổi nhất định. Nhưng khi thiết kế hỗ trợ những thay đổi tiên đoán được thì sẽ không hỗ trợ và có thể gây tổn hại cho những thay đổi không biết trước.

Tất cả những điều này làm cho thiết kế có kế hoạch có vẻ không khả thi. Chắc chắn đó là thử thách lớn.

Triển khai những kỹ thuật của XP

Đường cong thay đổi nói rằng khi dự án đang tiến hành, cái giá của thay đổi tăng theo cấp số mũ. Điều khá hài ước là nhiều dự án vẫn chạy với quy trình cục bộ mà không có công đoạn phân tích. Điều này giải thích tại sao thiết kế tiến hóa không thể hoạt động.

Giả định cơ bản của XP là có thể làm cho đường cong thay đổi đủ "phẳng" để thiết kế tiến hóa hoạt động và chỉ có thể đạt được điều này khi áp dụng các kỹ thuật của XP. Có rất nhiều kỹ thuật sử dụng trong XP, ở trung tâm là kỹ thuật Kiểm thử và Tích hợp liên tục và tái cấu trúc.

Tôi chắc chắn rằng vẫn có chỗ cho thiết kế trước khi lập trình. Hầu hết nằm ở các phân đoạn trước khi lập trình một tác vụ cụ thể. Nhưng có một điểm cân bằng mới giữa thiết kế up- front và tái cấu trúc.

Giá trị của sự Đơn giản

Thông thường YAGNI (You Aren't Going to Need It - Bạn sẽ không cần nó) được môt tả là: hôm nay bạn không nên thêm mã chỉ được dùng bởi các tính năng mà ngày mai mới cần. Bề ngoài khẩu hiệu này tưởng rất đơn giản. Vấn đề phát sinh khi làm việc với những thứ như khung làm việc, bộ phận tái sử dụng, và thiết kế linh hoạt. Xây dựng chúng rất phức tạp. Nhưng nếu bạn xây chúng ngay từ đầu sẽ làm tăng chi phí thay đổi và có thể là không bao giờ sử dụng tính năng đó.

Tuy nhiên bạn chỉ có thể làm như thế một cách hợp lý khi bạn dùng XP hoặc một kỹ thuật tương tự với chi phí thay đổi nhỏ hơn.

Mẫu thiết kế và XP

XP có thể là một quy trình phát triển, nhưng mẫu thiết kế là kiến thức xương sống của thiết kế, kiến thức này có giá trị dù quy trình của bạn là gì đi chăng nữa. XP nhấn mạnh vào cả việc không dùng mẫu thiết kế tới khi thực sự cần và áp dụng một mẫu thiết kế theo một cài đặt đơn giản. Nhưng mẫu thiết kế vẫn là một kiến thức then chốt.

Nuôi lớn một kiến trúc

Đối với chúng ta kiến trúc phần mềm là gì? Với tôi kiến trúc bao gồm ý tưởng về những phần nòng cốt của hệ thống, những bộ phận khó thay đổi. Đó là nền tảng để từ đó xây phần còn lại.

Tôi nghĩ kiến trúc khởi điểm có một vai trò nhất định.

Đó là những thứ ở tầng ứng dụng, cách tương tác với cơ sở dữ liệu (nếu bạn cần), cách để làm việc với web server.

UML và XP

XP và UML có tương thích với nhau không? Chắc chắn XP không nhấn mạnh rằng sơ đồ là tốt trong phạm vi lớn, mặc dù có cho rằng "hãy dùng chúng nếu thấy hữu ích".

Giá trị chính của sơ đồ là giao tiếp. Do đó hãy chọn những thứ quan trọng và bỏ qua những thứ ít quan trọng. Mã là đầy đủ nhất, dễ nhất để đồng bộ với mã.

Bạn có muốn trở thành một Kiến trúc sư khi lớn lên?

Trong XP, một kiến trúc sư thay vì đưa ra mọi quyết định thì, họ lãnh đạo về kỹ thuật tức là dạy và giúp lập trình viên ra quyết định. Những lãnh đạo kỹ thuật phải có kỹ năng với con người tốt và giỏi kỹ thuật.

Tính có khả năng đảo ngược

Một trong những khía cạnh quan trọng nhất của cả hai cách tiếp cận là giải quyết tính phức tạp bằng cách giảm những quyết định không thể đảo ngược được trong qui trình.

Do đó trong thiết kế tiến hóa, nhà thiết kế cần nghĩ về cách để tránh tính không đảo ngược trong quyết định. Thay vì cố gắng để có quyết định ngay, hãy tìm một cách để ra quyết định muộn hơn (khi bạn có nhiều thông tin hơn) hoặc ra quyết định để sau đó bạn có thể làm lại mà không quá khó.

Ý chí để thiết kế

Để hoạt động được, thiết kế tiến hóa cần một lực để hội tụ. Lực này chỉ có thể từ con người – một vài người trong đội có trách nhiệm đảm bảo thiết kế có chất lượng cao.

Trách nhiệm của họ là quan sát bộ mã, phát hiện những vùng đang trở nên lộn xộn, và nhanh chóng hành động để sửa chữa trước khi chúng vượt khỏi tầm kiểm soát. Thiếu ý chí để thiết kế có vẻ như là nguyên nhân lớn để thiết kế tiến hóa thất bại.

Những thứ khó tái cấu trúc

Chúng ta có thể dùng tái cấu trúc để giải quyết mọi quyết định thiết kế, hoặc liệu có vấn đề quá ăn sâu mà việc thêm vào sau trở nên khó khăn? Mặc dù cũng sẽ tốn sức lực đáng kể để tái cấu trúc giải pháp đơn giản hiện thời thành giải pháp bạn thực sự cần, nhưng có vẻ như việc tái cấu trúc ít tốn kém hơn là xây dựng một tính năng còn nghi ngờ. Và một điểm rất quan trọng là chuyển giao sớm cho khách hàng, nó có sức mạnh ghê gớm trong việc tập trung sự chú ý của khách hàng, tăng uy tín, và là nguồn lớn để học tập.

Thiết kế đang xảy ra?

Một trong những khó khăn của thiết kế tiến hóa là rất khó để nói liệu thiết kế có đang xảy ra. Sự nguy hiểm của thiết kế trộn lẫn lập trình là việc lập trình xảy ra mà có thể không có thiết kế – đó là tình huống Thiết kế Tiến hóa phân kỳ và thất bại.

Không có câu trả lời dễ dàng cho câu hỏi này, nhưng đây là một số gợi ý.

- Lắng nghe người làm kỹ thuật.
- Quan tâm tới lượng mã bị bỏ đi. Một dự án thực hiện tái cấu trúc tốt sẽ dần loại bỏ mã xấu. Nếu không có gì được xóa đi thì đó gần như chắc chắn là một dấu hiệu của việc đang không thực hiện đủ tái cấu trúc.

"Dẫu sao mọi thứ trên Trái đất đều đơn giản"

Vậy thiết kế đã chết?

Bản chất của thiết kế đã thay đổi. Thiết kế trong XP cần những kỹ năng sau:

- Mong muốn giữ mã rõ ràng và đơn giản nhất có thể.
- Kỹ năng tái cấu trúc để bạn có thể tự tin thực hiện bất cứ cải tiến nào khi thấy cần thiết.
- Kiến thức tốt về mẫu thiết kế: không chỉ là giải pháp mà còn đánh giá cao thời điểm sử dụng và cách tiến hóa thành mẫu thiết kế.
- Thiết kế có quan tâm tới những thay đổi trong tương lai, biết rằng quyết định hiện thời sẽ bị thay đổi trong tương lai.
- Biết cách giao tiếp của thiết kế và người cần hiểu thiết kế bằng cách sử dụng mã, sơ đồ và trên tất cả: sự đàm thoại.

Đó là sự lựa chọn một cách kiêm tốn các kỹ năng, nhưng để trở thành một nhà thiết kế tốt luôn rất khó. Thực tế XP không làm nó đơn giản hơn, ít nhất là với tôi. Nhưng tôi nghĩ XP cho chúng ta một cách suy nghĩ mới về thiết kế hiệu quả bởi nó tạo ra thiết kế tiến hóa, một chiến lược hợp lý. Tôi là một người rất hâm mộ tiến hóa – nếu không phải thế thì ai biết tôi có thể là gì?

Kanban cho mọi người

Dương Trọng Tấn

Hằng ngày chúng ta có rất nhiều việc phải lo: từ công việc cá nhân, học tập, tới công việc ở nơi công sở, các dự án lớn lao cho tới những việc mua quà sinh nhật cho cô bạn cấp Một. Tất cả việc nào cũng muốn làm, việc gì cũng có vẻ gấp. Vậy mà mỗi ngày chỉ có 24 giờ.

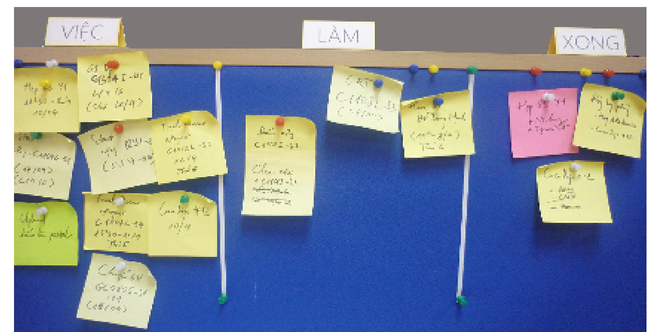
Nhiều người ức chế vì không thể hoàn thành được công việc được giao, lúc nào cũng phải làm quá nhiều việc, đang làm việc này lại có yêu cầu việc khác. Stress dồn lên stress, càng khiến hiệu suất công việc giảm đi trông thấy. Làm sao đây?

Jim Benson đã học tập các nguyên lí cốt lõi của Tinh gọn (Lean) để sáng tạo phương pháp được gọi là Personal Kanban (Bảng công việc cá nhân), áp dụng cho mọi người với chỉ hai quy tắc đơn giản: giới hạn khối lượng công việc đang làm (Limit Work-In-Progress) về khả năng của cá nhân (capacity), và trực quan hóa đầu việc. Với Personal Kanban, ai cũng có thể giảm đáng kể stress và gia tăng năng suất lao động mà không phải thuê thêm Ô-sin nhắc việc.

Cách làm một Bảng công việc cá nhân rất đơn giản, chỉ bằng vài miếng giấy dán, hay những phần mềm ghi chú có sẵn trên PC, hoặc những công cụ số hóa chuyên nghiệp mạnh mẽ.

Tạo Personal Kanban với giấy dán

Mời bạn xem hai cái Personal Kanban của hai người sau đây:



Kanban của một Giám đốc đào tạo

Personal Kanban của một Lập Trình Viên – Sinh viên

Kanban trên mây và di động

Trông đơn giản nhỉ? Tôi có phải mô tả thêm điều gì để bạn thực hiện ngay không?

Trên một cái bảng (hay chỗ nào dán\dính giấy được) bạn tạo ra ba cột có tên "Cần làm" (ToDo), "Đang làm" (Doing) và "Xong" (Done).

Khi có việc cần làm (việc tự nghĩ ra, việc được giao…), bạn viết vào tờ giấy dán và đặt vào ToDo trước, phân tích kĩ lưỡng nên làm ngay hay để làm sau. Việc này có bản chất là "lập kế hoạch", sẽ giúp bạn có được trình tự và cách làm công việc có bài bản hơn. Nhiều người bắt tay vào làm ngay việc được giao mà không suy nghĩ, tính toán. Đó không phải là chiến lược tốt. Nếu bạn có thể xếp độ ưu tiên theo giá trị (cái nào có giá trị thì làm trước), thì ta có thể mất ít công sức hơn mà làm được nhiều giá trị hơn (sử dụng quy tắc Pareto, 80-20).

Khi quyết định làm việc gì, ta sẽ chuyển công việc sang cột Doing. Có thể ghi ngày giờ bắt đầu làm lên giấy. Giới hạn số lượng thẻ ở cột này (ví dụ 3). Đừng để nhiều, vì nó sẽ khiến bạn phải nhảy từ công việc nọ sang công việc kia (task-switching), là nguồn cơn của thiếu hiệu quả và stress. Con số 3 hay 5 tùy thuộc giới hạn khả năng (capacity) của từng người, chỉ bạn mới biết được. Khi bạn đặt con số 5 và thấy bắt đầu rối tung lên thì chắc là phải giới hạn con số đó xuống thấp hơn. Thực hiện trong một hai tuần rồi đánh giá lại con số đó. Qua một hai tuần ta sẽ có con số hợp lí. Nhưng khi khởi đầu, tôi gợi ý là nên để con số 3.

Khi làm xong việc gì thì đặt nó sang cột Done, có thể ghi ngày giờ kết thúc lên giấy để đánh giá về sau.

Việc đặt một công việc sang cột Done chứ không vứt đi sẽ giúp bạn nhìn thấy được tiến độ công việc, tạo giá trị thúc đẩy bản thân. Đó là giá trị của trực quan hóa (visualization).

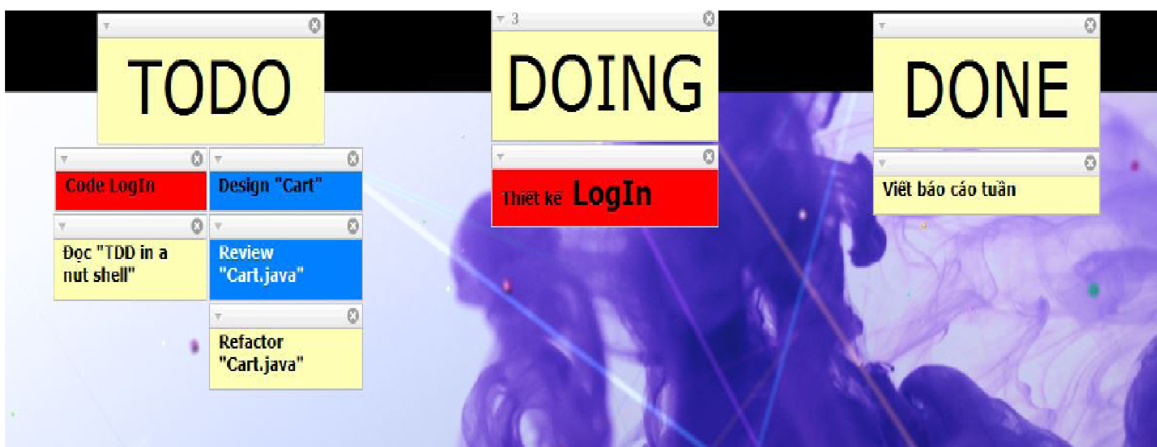
Về tờ giấy dán, bạn có thể chọn nhiều màu, dùng nó tùy theo chủ ý. Ví dụ: các việc học tập để giấy xanh, các việc giấy tờ để giấy vàng, các việc liên quan đến khách hàng dùng giấy đỏ v.v.. Tùy bạn. Nhưng hãy dùng có chủ ý. Việc này sẽ giúp cho bảng trực quan hơn, có sức sống hơn.

Vào mỗi cuối tuần (hay cuối tháng – tùy bạn, tôi thì thích cuối tuần), bạn có thể ngồi lại một chút tự đánh giá lại mình đã làm được bao nhiêu việc, trong đó có bao nhiêu việc ưu tiên, bao nhiêu việc trễ hẹn, bao nhiêu việc đạt kết quả mĩ mãn. Rồi so sánh với tuần trước đó. Qua đó, bạn có thể đánh giá được cách thức làm việc của chính mình. Rồi đưa ra một vài ý tưởng để cải thiện cách làm việc của chính mình. Điều này sẽ giúp bạn liên tục cải tiến. Ngày một tốt hơn.

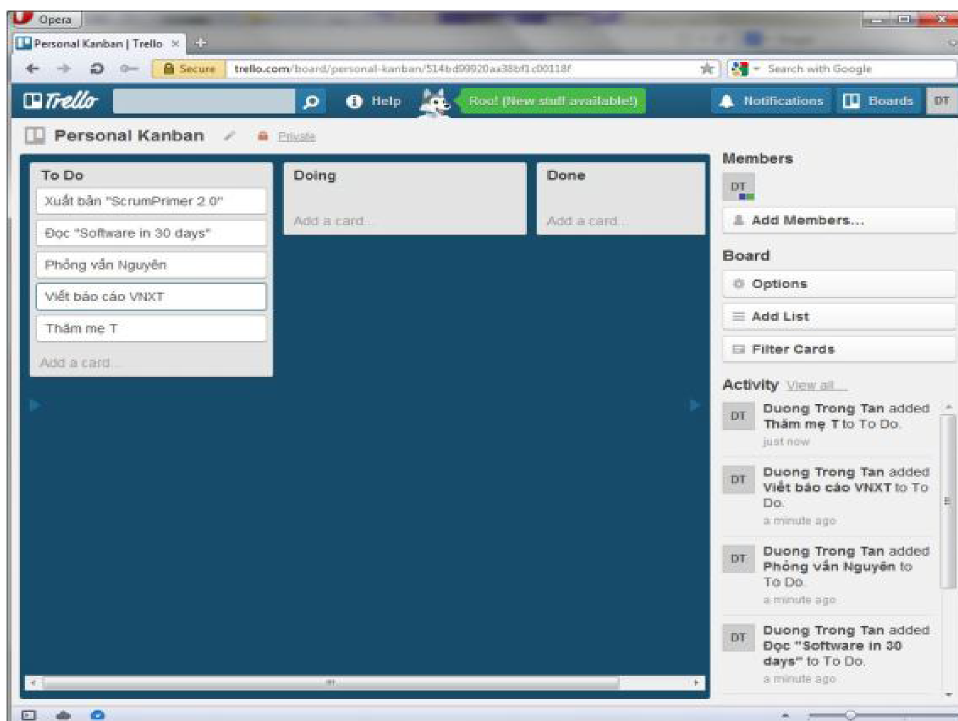
Biến Desktop thành kanban

Bạn có thể vận dụng quy tắc trên để biến desktop thành kanban như hình dưới:

Chỉ cần một phần mềm giả lập giấy dán (sticky notes), dọn sạch desktop và làm vài cái cột là xong. Cách làm hoàn toàn tương tự như mô tả ở phần trên.



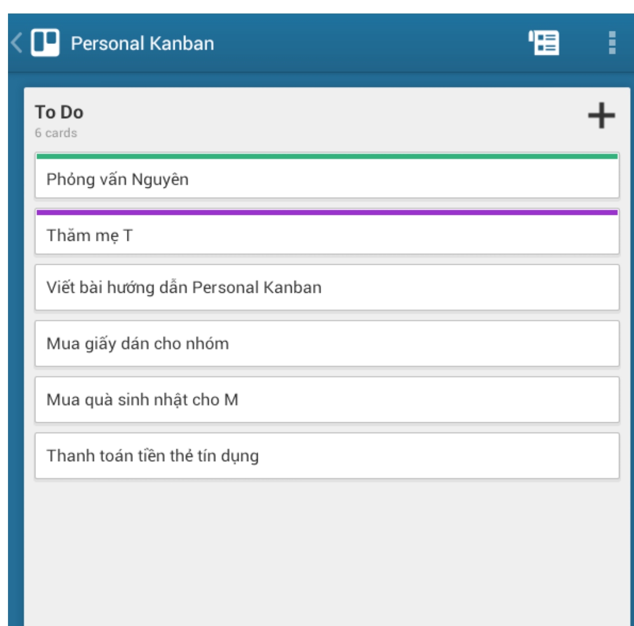
Kanban trên Desktop



Ảnh chụp màn hình Galaxy Tab 7 Plus

Lựa chọn công cụ "chuyên nghiệp"

Nếu bạn hay di chuyển, làm việc nhiều trên máy tính chứ không thích giấy dán thì có thể lựa chọn một công cụ số hóa ưa thích để làm Personal Kanban. Có rất nhiều công cụ chạy trên nhiều nền tảng khác nhau, từ Android, iOS đến Windows hay Ubuntu. Công cụ loại này có thể kể đến: Trello, KanbanFlow, Pomodoro Daisuki, LeanKit, Kanbanery, JIRA, KanbanTool, PearlTrees,… Với những công cụ hỗ trợ đa nền tảng, (như Trello chẳng hạn) bạn có thể tạo lập một bảng công việc lưu trữ "trên mây", truy xuất ở mọi nơi, mọi lúc.



Những công việc cần làm

Việc tạo lập Personal Kanban trên các dịch vụ này khá đơn giản. Hãy vào trang web của nhà cung cấp và đăng kí với vài thao tác nhỏ, sẽ có ngay một kanban tiện dụng và đẹp mắt.

Personal Kanban là công cụ hết sức đơn giản nhưng hữu hiệu. Rất nhiều người thấy ngạc nhiên vì độ hiệu của công cụ đơn giản này. Bạn hãy thử và sẽ thấy ngay điều đó sau một tuần. Tuy thế Personal kanban chỉ là công cụ, nó không quyết định được hiệu quả công việc hằng ngày của bạn. Điều đó phụ thuộc vào khả năng xử lí từng việc cụ thể đó. Personal kanban chỉ giúp bạn trực quan hóa mọi thứ để dễ bề trù liệu và lập kế hoạch, giới hạn công việc để không bị phân tán và mất tập trung. Nếu biết dùng công cụ một cách hữu hiệu, nó sẽ giúp ích rất nhiều; bằng không, nó chỉ làm phiền bạn thêm thôi. Có người bảo công cụ nối dài cánh tay của con người; nhưng có người lại bảo càng nhiều công cụ con người càng dễ bị lệ thuộc. Cũng có ý đúng cả. Tôi tin là bạn có thể dễ dàng làm chủ Personal Kanban. Chắc chắn thế.

Kiểm thử đơn vị với PHPUnit trên Netbeans

Nguyễn Việt Khoa

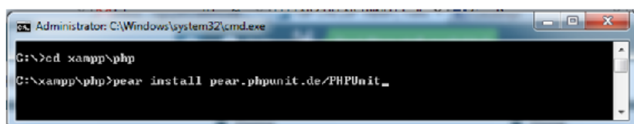
Khái niệm Kiểm thử đơn vị (Unit Testing) không còn xa lạ với những lập trình viên luôn hướng tới chất lượng của từng dòng code. Với bài viết này tôi mong muốn sẽ giúp các lập trình viên PHP biết cách triển khai Unit Testing với framework PHPUnit. Đây là một Framework nổi tiếng thế giới, nó cho phép bạn triển khai Unit Testing với nhiều IDE khác nhau, ở bài viết này tôi sẽ hướng dẫn bạn triển khai với Netbeans 7.2.

Netbeans là IDE được nhiều lập trình viên Java sử dụng, tuy vậy đây cũng là một trong những IDE hỗ trợ lập trình PHP và đặc biệt là hỗ trợ đắc lực cho việc triển khai kiểm thử tự động với PHPUnit.

Trước hết để có thể triển khai ứng dụng với PHP bạn cần có môi trường cho ngôn ngữ này đã. Bạn dễ dàng làm quen với PHP và chuẩn bị môi trường để phát triển ứng dụng (web) với ngôn ngữ này. Về mặt căn bản bạn chỉ cần cài XAMPP là đủ để phát triển ứng dụng web với PHP, nếu không cần đi quá sâu về môi trường phát triển PHP.

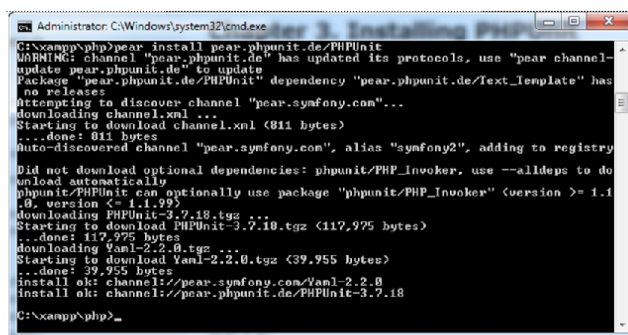
Có thể bạn đã có XAMPP, tuy nhiên PHPUnit cần phải có thêm sự hỗ trợ của PEAR. Hướng dẫn cài PEAR có khá nhiều trên Internet, ví dụ như trang web http://www.phpunit.de. Hoặc bạn có thể thực hiện các lệnh như tôi đã làm trên màn hình lệnh của Windows (cũng có thể bạn có cách khác đơn giản hơn? cho chúng tôi cùng biết với nhé). Lưu ý, nếu bạn dùng Windows với tài khoản thường thì cần phải chạy CMD (chương trình để chạy các lệnh) với quyền của Administrator (Run as Administrator), đặc biệt là trên Windows 8.

Lưu ý lệnh cài đặt PHPUnit với PEAR (dòng lệnh thứ 2) chỉ thực hiện được thành công với những điều kiện sau:



- Bạn đã sẵn có XAMPP và đang vào đúng thư mục php nằm trong thư mục cài đặt của nó (tôi đang có XAMPP ở tại C:\xampp).
- Máy tính của bạn phải được kết nối Internet vì lệnh cài đặt trên sẽ tải bộ cài từ repo trên mạng về trước khi cài vào máy tính của bạn.

Bạn sẽ nhận được các thông báo dạng như sau:



Nếu toàn thấy "OK" như trên thì chắc bạn đã thành công và PHPUnit đã sẵn sàng để bạn xài nó.

Nếu báo lỗi bạn cần chạy lệnh sau:

pear config-set auto_discover 1

Sau đó chạy lại lệnh:

pear install pear.phpunit.de/PHPUnit

Nếu vẫn có lỗi chắc bạn cần tìm sự hỗ trợ từ Internet hoặc từ một chuyên gia nào đó.

Bây giờ là lúc bạn cần kiểm tra xem Netbeans bạn đang dùng đã hỗ trợ PHP chưa? Nếu chưa bạn có hai lựa chọn sau:

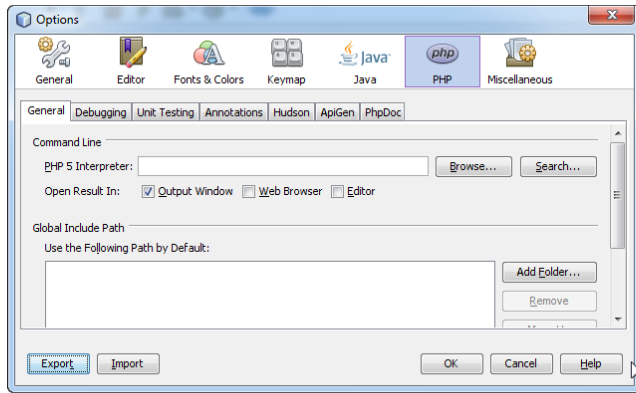
- Tải bản Netbeans hỗ trợ PHP và cài đặt nó: http://netbeans.org/downloads/index.html
- Sử dụng công cụ cài đặt plug-in của Netbeans để tìm các plug-in về PHP:

Bạn cần cài đủ những plug-in sau:

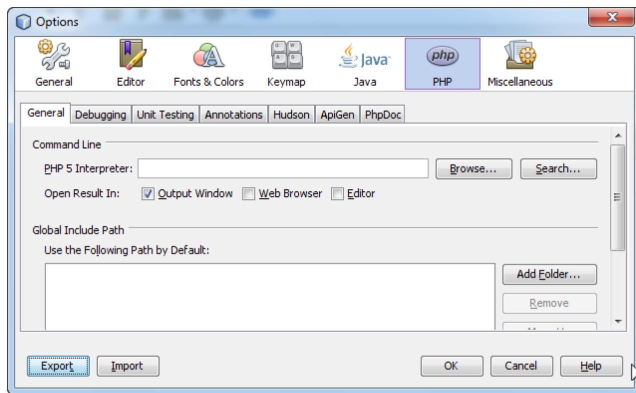
- PHP

- Selenium Module for PHP
- PHP Documentor Tag Help
- PHP Documentor

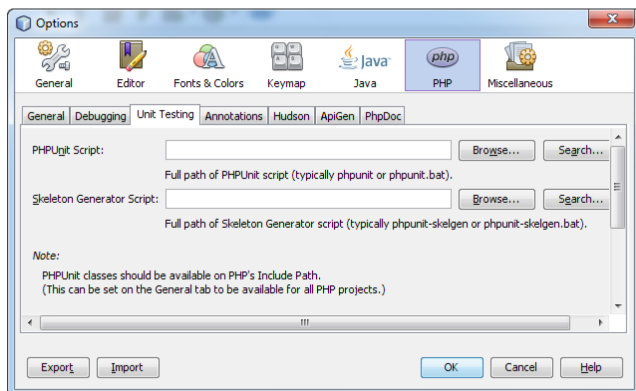
Những plug-in trên ngoài việc hỗ trợ viết PHP nó còn giúp bạn triển khai Unit Testing ngay trên Netbeans với PHPUnit.



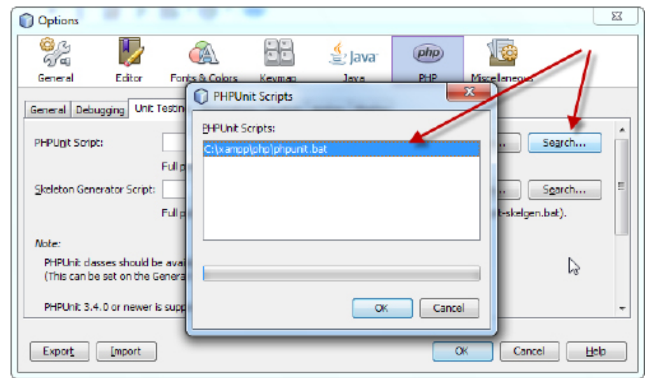
Việc tiếp theo là kiểm tra xem Netbeans đã nhận PHPUnit chưa theo các bước sau: Từ thanh trình đơn của Netbeans > Tools > Options > chọn tab PHP



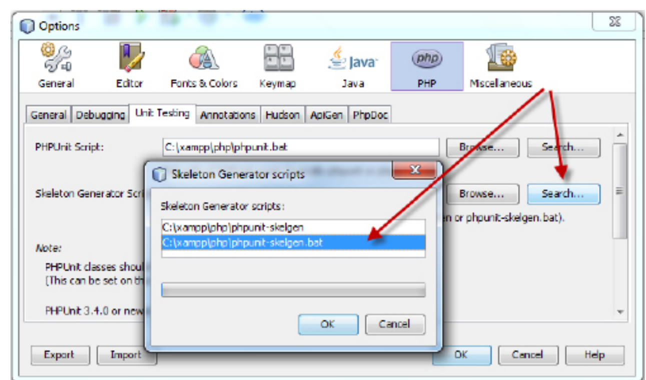
Mở tiếp sang tab Unit Testing



Bấm nút Search trong phần PHPUnit Script để Netbeans tự động xác định PHPUnit.



Bấm nút Search trong phần Skeleton Generator Script để Netbeans tự động xác định công cụ sinh ra các mã dành cho Unit Testing



Bạn đã sẵn sàng cho những Kiểm thử đơn vị đầu tiên.

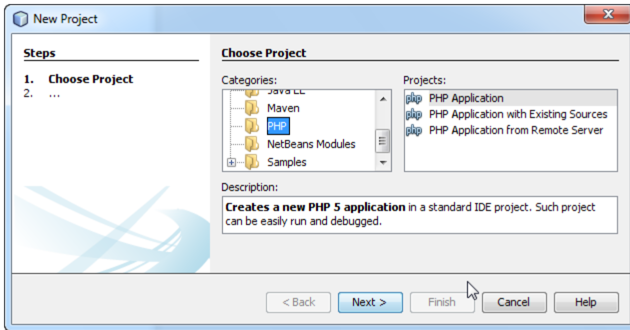
Vấn đề lúc này là: bạn đã biết căn bản về PHP chưa? Nếu câu trả lời là chưa! Xin mời bạn tìm hiểu trước về PHP ở loạt bài sau trên Tạp chí Lập trình:

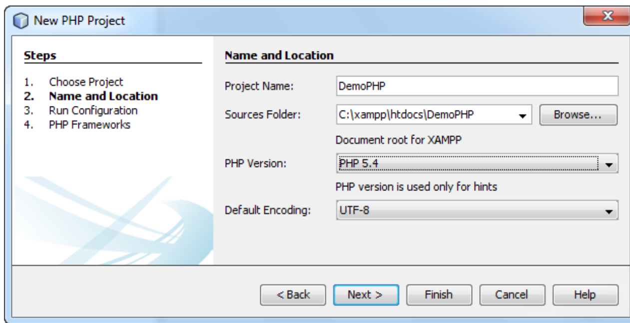
- Bắt đầu với PHP và CMS
- [PHP] Những dòng code đầu tiên
- [PHP] Các cấu trúc điều kiện
- [PHP]Các cấu trúc lặp
- [PHP]Quy ước đặt tên

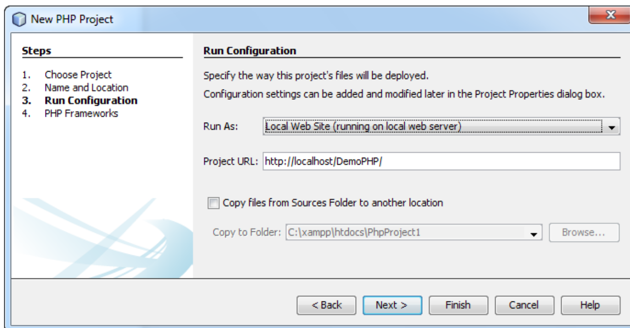
Hoặc tự tìm hiểu ở những nguồn phong phú hơn như www.php.net hay w3schools.com. Tuy nhiên với những dòng code mà tôi sử dụng dưới đây, bạn không nên lo lắng về khả năng code PHP của mình (chỉ cần bạn đã biết về OOP với Java\C++\C#\v.v..)

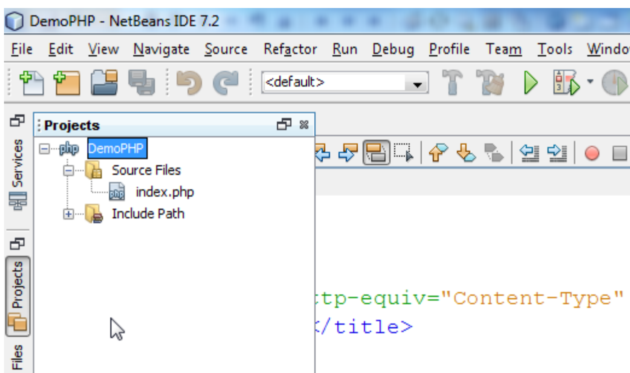
Nếu câu trả lời là OK, chúng ta cùng nhau đi tiếp từ đây.

Chúng ta sẽ bắt đầu bằng việc tạo một project PHP trên Netbeans, ví dụ project của tôi là DemoPHP

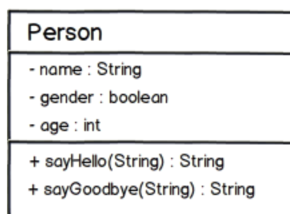




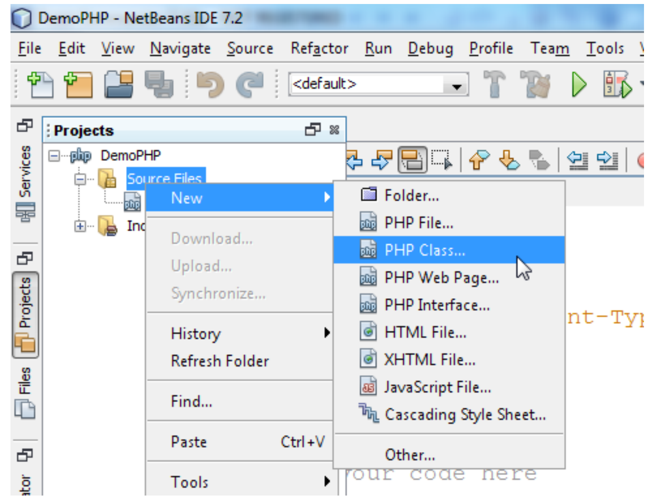




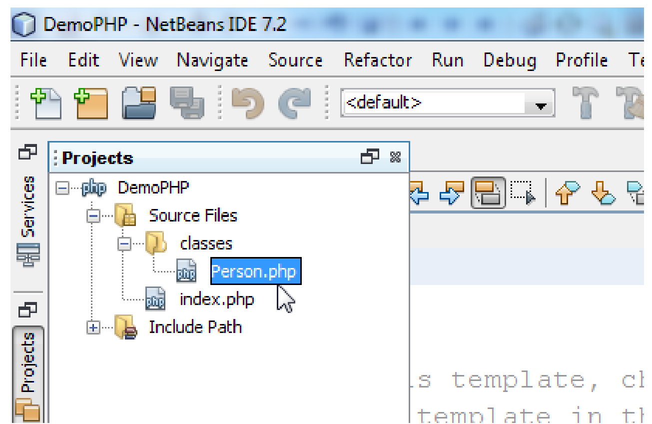
Giờ ta cần xây dựng một lớp (class) như sau cho project của mình



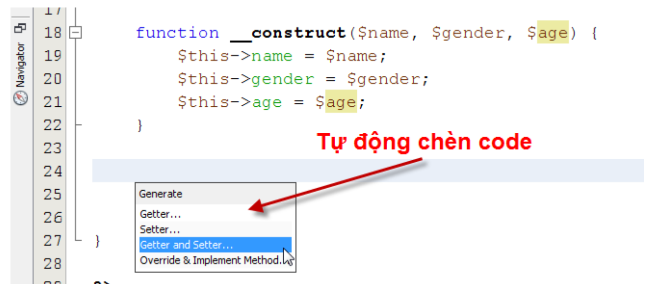
Cụ thể trên Netbeans như sau



Bạn nên đưa các class vào một thư mục để tiện quản lý, VD tôi đưa chúng vào thư mục classes



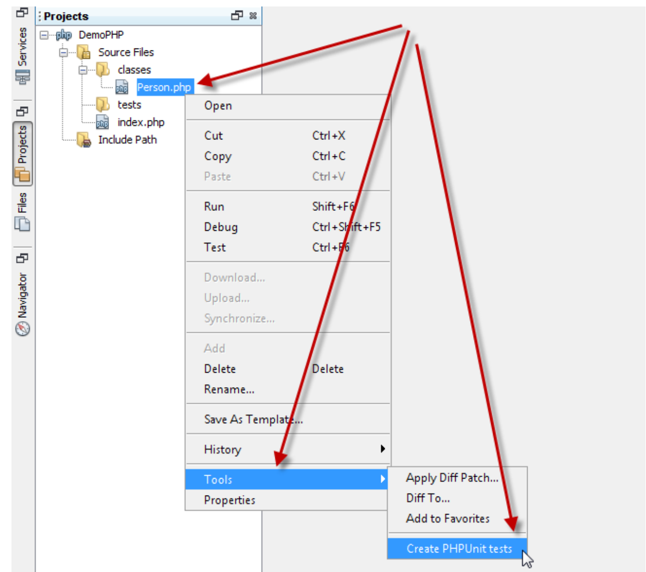
Bạn lưu ý là Netbeans hỗ trợ tự động sinh các code căn bản cho một class trong PHP (sử dụng tổ hợp phím Alt+Insert)



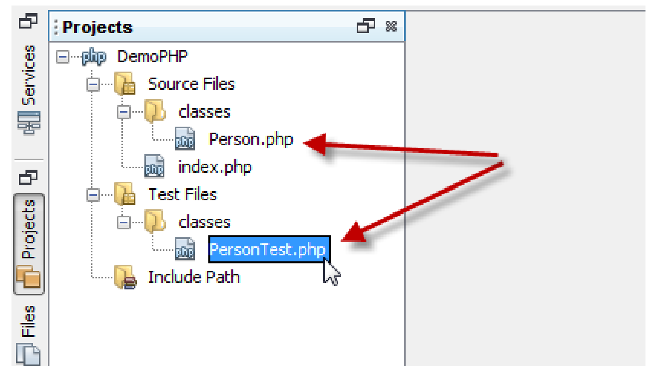
Đây là code tôi viết cho class này (thực tế phần lớn là do Netbeasn tự sinh ra)

```
<?php
/*
** Description of Person
** @author KhoaNV
*/
class Person {
    private $name;
    private $gender;
    private $age;

    function __construct($name, $gender, $age) {
        $this->name = $name;
        $this->gender = $gender;
        $this->age = $age;
    }

    public function getName() {
        return $this->name;
    }

    public function setName($name) {
        $this->name = $name;
    }

    public function getGender() {
        return $this->gender;
    }

    public function setGender($gender) {
        $this->gender = $gender;
    }

    public function getAge() {
        return $this->age;
    }

    public function setAge($age) {
        $this->age = $age;
    }

    public function sayHello($otherName){
        throw new Exception("Error!");
    }

    public function sayGoodbye($otherName){
        throw new Exception("Error!");
    }
}
?>
```

Tôi chưa vội hoàn chỉnh hàm sayHello và sayGoodbye, vì cái chúng ta đang cần là Unit Testing với PHPUnit và tôi thì thích triển khai nó theo TDD. Do đó, tôi sẽ tạo một test (kiểm thử) cho lớp Person, cụ thể là cho phương thức sayHello. Netbeans với những plug-in mà bạn cài đặt ở trên sẽ dễ dàng giúp bạn tạo ra cái test này. Bạn nên tạo một thư mục để chứa các test, VD tôi đặt thư mục này là tests (cùng thư mục cha với thư mục classes). Mời bạn quan sát hình ảnh sau:



Nếu không gặp lỗi gì, bạn sẽ có một lớp mới dùng để test cho lớp Person vừa tạo ở trên



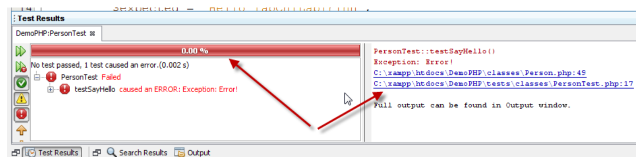
Để đỡ mất tập trung vào các hàm test khác (tự sinh ra bởi công cụ) bạn có thể xóa hết các hàm có trong lớp PersonTest này và chỉ để lại hàm test cho sayHello

```
<?php
require_once '../../classes/Person.php';
class PersonTest extends PHPUnit_Framework_TestCase {

    /**
     * @covers Person::sayHello
     * @todo   Implement testSayHello().
     */
    public function testSayHello() {
        $this->markTestIncomplete(
            'This test has not been implemented yet.'
        );
    }
}
```

Giờ ta cần hoàn chỉnh một test-case để kiểm tra tính đúng đắn của sayHello. Mời bạn xem đoạn code sau

```
<?php
require_once '../../classes/Person.php';

class PersonTest extends PHPUnit_Framework_TestCase {
  /**
   * @covers Person::sayHello
   * @todo   Implement testSayHello().
   */
  public function testSayHello() {
    $expected = "Hello TapChiLapTrinh. I'm Khoa";
    $person = new Person("Khoa", true, 18);

    $this->assertEquals($expected, $person->sayHello('TapChiLapTrinh'));
  }
}
```

Lưu ý dòng code "require_once '../../classes/Person.php';" dùng để xác định nơi chứa code của lớp mà bạn muốn test. Tôi không hiểu sao nó không được tự động chèn vào? Hiện tôi tự code dòng đó, nếu bạn biết cách nào đó mà không phải làm vậy, xin hãy khai sáng cho tôi.Bạn hãy chạy thử test-case này để kiểm tra kết quả (nhấp chuột phải vào và chọn Run File hoặc nhấp chuột phải vào lớp Person và chọn Test). Chắc chắn nếu bạn chưa hoàn chỉnh code của hàm sayHello thì bạn sẽ nhận được thông báo như sau

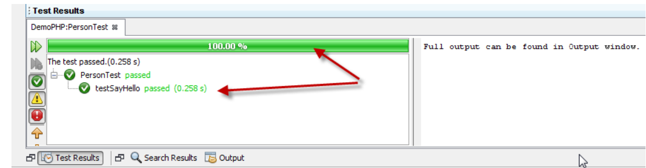


Nếu bạn nhận được thông báo như trên thì bạn đã có tín hiệu vui mừng đầu tiên trong TDD, "Red: Create a test and make it fail".

Nào, bây giờ thì chúng ta quay lại để hoàn tất code của hàm sayHello

```
public function sayHello($otherName){
  $strTemp = trim($otherName);
  if($strTemp!=''){
    return "Hello " . $otherName . ". I'm " . $this->name;
  }
  throw new Exception("Error!");
}
```

Vòng lại bước chạy test-case xem sao nào? Nếu bạn vẫn nhận được Red, chắc là bạn cần xem lại code của hàm sayHello. Còn với tôi kết quả là như sau



Tôi có thể vui được rồi, vì mình đã có tín hiệu vui thứ hai trong TDD đó là "Green: Make the test pass by any means necessary".

Đến đây tôi xin tạm dừng bài viết của mình, tôi nghĩ rằng lúc này bạn đã có thể tự mình triển khai Unit Testing với PHPUnit được rồi.

Chúc bạn thành công!

Kiểm thử đơn vị (Unit testing)

Mục tiêu chính của kiểm thử đơn vị là lấy một đoạn mã nhỏ ở trong ứng dụng, cách ly nó khỏi những đoạn mã còn lại, sau đó xác định xem nó có hoạt động chính xác như bạn mong đợi hay không. Thông thường, một kiểm thử đơn vị sẽ gọi một phương thức nào đó của bạn với các tham số đầu vào, sau đó nó sẽ so sánh kết quả trả về với kết quả mong đợi để đánh giá hành vi của phương thức đó. Mỗi đơn vị được kiểm thử riêng biệt trước khi tích hợp chúng vào trong các module và tiến hành kiểm thử giao diện.

Lợi ích của kiểm thử đơn vị

Giá trị của kiểm thử đơn vị đã được khẳng định thông qua tỷ lệ lỗi được tìm thấy trong quá trình áp dụng nó. Kiểm thử đơn vị cho phép chúng ta tin tưởng hơn vào dòng mã của mình. Sử dụng kiểm thử đơn vị còn giúp ta tiết kiệm được thời gian; Khi phát hiện ra một lỗi, chúng ta sẽ viết mã kiểm thử cho nó, sau đó sẽ sửa lỗi, và như vậy lỗi này sẽ không thể xảy khi chúng ta phát hành sản phẩm bởi vì đoạn mã kiểm thử sẽ phát hiện ra nó ngay.
Một lợi điểm nữa của kiểm thử đơn vị đó là nó cung cấp cho chúng ta một tài liệu vô cùng chi tiết, bởi vì bản thân nó cho phép chúng ta biết được những đoạn mã của mình sẽ được thiết kế ra sao.

Những ngôn ngữ nào có thể triển khai kiểm thử đơn vị?

Hiện nay trên thị trường tồn tại rất nhiều nền tảng kiểm thử đơn vị cho nhiều ngôn ngữ khác nhau. Hãy tìm lấy một nền tảng và bắt đầu áp dụng cho ngôn ngữ lập trình mà bạn ưa thích nhé.

TCLT **Search** Images Mail Drive Calendar Sites Groups News More ▾

Thơ tình bên Computer

Có nhiều khi gục đầu bên KeyBoard,
Anh vô tình nhấn Shift viết tên em,
Anh yêu em mà em chẳng Open,
Mở cửa trái tim và Save anh vào đó.

Cửa nhà em, mẹ đã gài Password,
Anh suýt rách quần vì cố vượt FireWall,
Nhớ lần đầu khi đưa em về Home,
Anh hôn trộm liền xơi ngay một Tab,
Anh bàng hoàng quay xe BackSpace,
Ngoái nhìn em mà chẳng thể Ctrl,
Anh tức giận khi thấy một thằng Alt,
Cứ Insert mỗi khi mình nói chuyện.
Có nhiều khi muốn thẳng tay Delete,
Nhưng vì em, anh nuốt giận Cancel.

Anh biết anh chỉ là Hacker nghèo,
Còn hắn có @ và Esc.
Em thích hắn làm lòng anh Space,
Bước thẫn thờ chìm xuống vực PageDown
Em vội bước ra đi quên Logoff
Chẳng một lời dù chỉ tiếng Standby
Em quên hết kỷ niệm xưa đã Add
Quẳng tình anh vào khoảng trống Recyclebin

Anh vẫn đợi trên nền xanh Desktop
Bóng em vừa Refresh hồn anh
Từng cú Click em đi vào nỗi nhớ
Trong tim anh... Hardisk dần đầy
Anh ghét quá, muốn Clean đi tất cả
Nhưng phải làm sao khi chẳng biết Username

Hay mình sẽ một lần Full Format...
Em đã change Password cũ còn đâu!
Anh sẽ cố một lần, anh sẽ cố...
Sẽ Retry cho đến lúc Error
Nhưng em hỡi làm sao anh có thể
Khi Soft anh dùng đã hết Free Trial !
Hình bóng em vẫn mãi Default..

(*sưu* tầm)

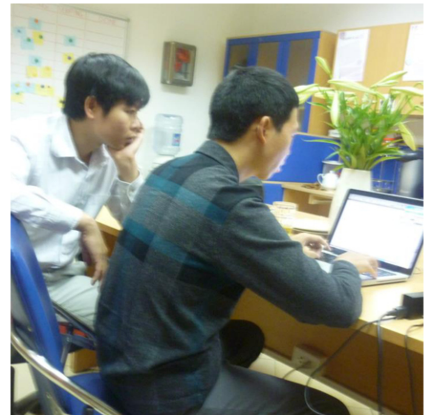
Bạn đọc với Tạp chí Lập trình

trantuanson1991: Cảm ơn Ban biên tập đã có những bài viết rất hay và bổ ích. Chúc Tạp chí Lập trình ngày càng thành công hơn với hướng đi mới của mình!

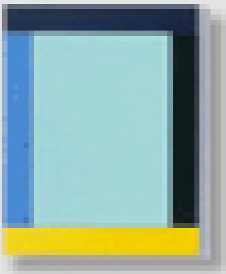
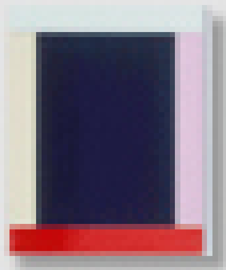
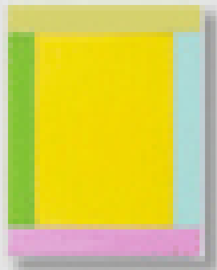
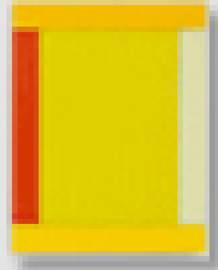
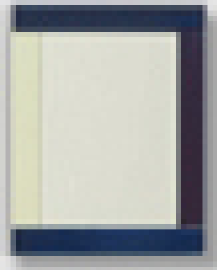
Bạch Hổ: Chúc các thành viên Ban biên tập vững tay code, chắc tay bút, luôn tìm và đem đến cho độc giả những thông tin hữu ích của mọi mặt trong ngành IT

Hải Nguyên Trần: Chào các bạn biên tập của Tạp chí Lập Trình, mình cũng mới biết đến TCLT, nhưng thấy rất hay và hữu ích. Chúc TCLT đạt được những điều mong muốn!

MHGS: Trước đây hồi còn là Sinh viên mình hay đọc tạp chí Echip nhưng bây giờ đã ra trường và đi làm thì Echip ko còn phù hợp nữa nên mình rất muốn tìm đọc những tạp chí nâng cao hơn về lập trình. Mình toàn phải săn các tạp chí và blog của nước ngoài để học. Vì vậy khi biết ở VN mới có Tạp chí lập trình mình rất háo hức



Bàn về sự
ĐƠN GIẢN

ĐỘI PHÁT TRIỂN

Phạm Anh Đới
Nguyễn Việt Khoa
Nguyễn Khắc Nhật
Nguyễn Minh Quân
Dương Trọng Tấn
Đặng Kim Thi
Nguyễn Ngọc Tú

THIẾT KẾ BÌA

Hà Dũng Hiệp

Vol.2

04/2013