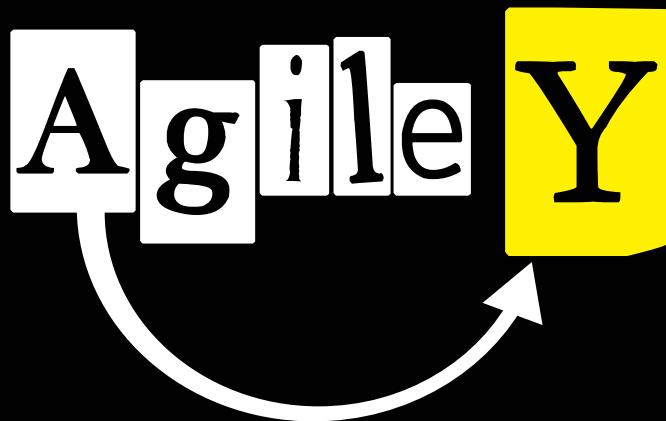
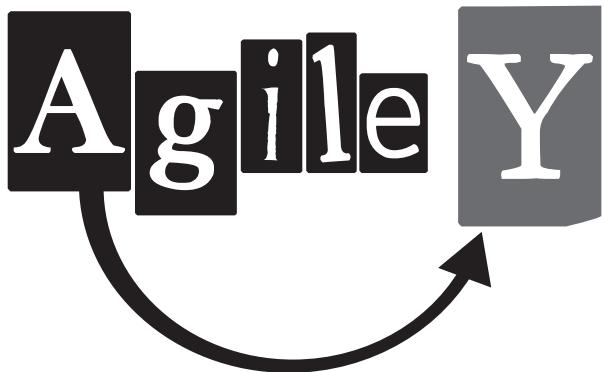
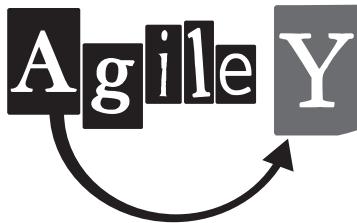


NGUYỄN HIỀN



NHÀ XUẤT BẢN TRI THỨC





- [15](#) PHẦN 01: AGILE
[16](#) Chương 01: Phát triển phần mềm cần linh hoạt
[25](#) Chương 02: Agile là giải pháp
[43](#) Chương 03: Scrum
[89](#) Chương 04: Kanban & Scrumban
[102](#) Chương 05: Kỹ thuật và công cụ
- [151](#) PHẦN 02: TỔ CHỨC LINH HOẠT
[152](#) Chương 06: Nhóm tự tổ chức liên chức năng
[169](#) Chương 07: Mở rộng Agile
[183](#) Chương 08: Tổ chức Agile
- [200](#) PHẦN 03: CÁ NHÂN LINH HOẠT
[201](#) Chương 09: Quản lý cá nhân
[209](#) Chương 10: Trợ giúp của Kanban
[224](#) Chương 11: Tối ưu hiệu quả
- [239](#) Phụ lục: Tham khảo
[245](#) Phụ lục: Thuật ngữ

Bạn đang cầm trên tay *Agile Y*, một trong những cuốn sách về Agile đầu tiên bằng tiếng Việt. Tôi coi đây là một trong những dự án lớn trong cuộc đời mình. Sáu năm trước, chúng tôi đã thành lập Hanoi Scrum và Agile Vietnam với mong muốn giới thiệu tới cộng đồng một phương pháp phát triển phần mềm hiện đại. Sau sáu năm, bằng những nỗ lực tuyệt vời, những người bạn của tôi đã gây dựng một cộng đồng lớn mạnh qua hàng chục hội thảo quy mô lớn, hàng chục học liệu được dịch và biên soạn. Nhưng cộng đồng vẫn thiếu những cuốn sách.

Một cuốn sách với những nội dung cơ bản, cung cấp góc nhìn rộng về Agile; nhằm trả lời những câu hỏi căn bản giúp cộng đồng Agile tiến thêm một bước. Và đó là lý do *Agile Y* ra đời.

Tôi mong rằng cuốn sách này có thể giúp những lập trình viên Việt Nam hiểu được giá trị mà Agile mang lại, không chỉ trong lĩnh vực phát triển phần mềm mà còn trong mỗi tổ chức và cuộc sống cá nhân. Từ đó trang bị cho mình một phong cách làm việc mới, sẵn sàng phù hợp với sự thay đổi của thế giới.

Cuốn sách này là lời cảm ơn đến những người bạn của tôi tại Hanoi Scrum và Agile Vietnam, những người đã đóng góp không vì lợi nhuận, miệt mài trong một thời gian dài. Cuốn sách này là lời cảm ơn tới Mikkel Lomholt, Lasse Rey-Andersen, và những cộng sự của tôi tại văn phòng Hà Nội – những người đã cho tôi cơ hội được thực hành và áp dụng những kiến thức về Agile trong những dự án phần mềm và xây dựng tổ chức tại Planday.

Và trên hết, cuốn sách này là lời cảm ơn đến các bạn, những người đã ủng hộ Hanoi Scrum, Agile Vietnam, và đặc biệt là sự mến mộ dành cho Agile, đã cùng chung tay với chúng tôi xây dựng một cộng đồng Agile lớn mạnh tại Việt Nam.

Ghi chú dành cho phiên bản ebook 2020.

Agile Y được xuất bản và giới thiệu vào tháng 12/2016. Tôi rất mừng vì đến thời điểm này vẫn nhận được những phản hồi tích cực từ các bạn độc giả. Nhiều bạn đọc sử dụng Agile Y trong bước đầu thực hành Agile; nhiều bạn đọc sử dụng Agile Y như một cuốn cẩm nang để chuẩn bị cho việc lấy chứng chỉ Agile/Scrum.

Điều này dẫn tôi tới quyết định xuất bản miễn phí Agile Y dưới định dạng ebook vào năm 2020.

Phiên bản 2020 đã chỉnh sửa nhiều lỗi chính tả, lỗi diễn đạt và bổ sung một số thông tin nhỏ, cũng như định dạng để phù hợp với phiên bản ebook. Tôi hy vọng bạn sẽ có trải nghiệm tốt hơn, cũng như tìm được những điều thú vị và giá trị bổ ích như những độc giả khác từ *Agile Y phiên bản ebook*.

Bạn được phép sử dụng và phân phối *Agile Y phiên bản ebook* với mục đích cá nhân và phi thương mại. Bạn chỉ được phép phân phối *Agile Y phiên bản ebook* ở nguyên bản hình thức, nội dung và định dạng. Bạn không được phép chỉnh sửa *Agile Y phiên bản ebook*, in ấn và tái phân phối *Agile Y phiên bản ebook* dưới các định dạng khác khi chưa có sự đồng ý của tác giả. Bạn không được phép sử dụng toàn phần hoặc một phần nội dung của *Agile Y phiên bản ebook* nếu không trực tiếp công bố *Agile Y* là nguồn tham khảo.

Khi bạn tiếp tục lưu trữ và tìm hiểu các nội dung từ trang 6 đến hết trang 255 của cuốn sách này, bạn mặc nhiên đồng ý với điều khoản trên.

Nếu muốn ủng hộ tác giả, bạn có thể ủng hộ thông qua địa chỉ:

<https://www.buymeacoffee.com/hiennguyen>

<https://leanpub.com/agiley>

Xin cảm ơn và chúc các bạn có được trải nghiệm tốt cùng *Agile Y phiên bản ebook*.

"Thế giới của chúng ta đang thay đổi quá nhanh". Tôi muốn chào đón bạn tới cuốn sách này bằng một khẳng định không thể bị phản bác.

Tôi viết những dòng này ngay sau vụ khủng bố gây chấn động vào ngày 13/11/2015 tại Paris khiến 129 người bị thiệt mạng, và ngay lập tức ở bên kia bờ Đại Tây Dương cuộc chạy đua vào Nhà Trắng bỗng nhiên ủng hộ ông Donald Trump dù ông bị chỉ trích chỉ vài ngày trước đó vì chính sách hạn chế người nhập cư. Bà Hillary Clinton cũng nhận được nhiều sự ủng hộ hơn bởi những chính sách cứng rắn của mình trong vấn đề an ninh dù trước đó những chiến dịch tranh cử của bà không thật sự ấn tượng với người Mỹ. Facebook cũng ngay lập tức kích hoạt chức năng *I'm safe*, và chỉ một ngày sau khi "giành điểm" về sự phản ứng nhanh với sự kiện, đích thân Mark Zuckerberg phải dẹp công việc hàng ngày của mình sang một bên và đăng đàn giải thích về việc *tại sao Paris chứ không phải Beirut* được Facebook kích hoạt chức năng này.

Còn chúng ta? Ngay sau khi bạn hào hứng với một chiếc iPhone 6S mới cứng vừa được phân phối, bạn hiểu rằng mình cũng cần phải "gom lúa" cho chiếc iPhone 7 sẽ ra mắt vào đầu năm sau. Sáu tháng trước, Apple quảng cáo cho Apple watch cứ như thể đã đến thời của những thiết bị di động nhỏ bé, khiến giới lập trình viên xôn xao, nghĩ xem sẽ thay đổi ứng dụng của mình ra sao cho phù hợp. Giờ đây họ giới thiệu chiếc iPad Pro, to hơn cả một chiếc laptop và gây ra cơn đau đầu tiếp theo cho những công ty phát triển ứng dụng di động.

Nói như vậy để thấy rằng thế giới của chúng ta đang thay đổi từng ngày và không còn chỗ cho việc bám theo những kế hoạch dài hạn. Giờ đây khả năng thích nghi và phản ứng kịp thời với những thay đổi thường xuyên chính là *năng lực sống còn* của cá nhân và doanh nghiệp. Apple, công ty quyền lực nhất thế giới cũng không thể bám theo *chiếc điện thoại có kích thước vàng* của iPhone 4 như kế hoạch, họ buộc phải chấp nhận và phản ứng với thói quen ưa thích những chiếc điện thoại màn hình lớn của người dùng khi giới thiệu tới 2 chiếc iPhone mới. iPhone 6 và iPhone 6+ đã chạm đúng thị phần màu mỡ nhất của Samsung và buộc công ty này phải thay đổi kế hoạch một lần nữa với sản phẩm Galaxy S Edge.

Hơn 200 năm trước, Adam Smith đã sáng tạo ra một phương pháp sản xuất gọi là *quy trình*, giúp tăng năng suất sản phẩm đầu ra tới 200 lần và thay đổi cả nền công nghiệp. Ngày nay, quy trình đó vẫn đang phát huy những điểm mạnh của mình trong dây chuyền sản xuất bằng việc chia nhỏ những phân đoạn công việc và mỗi người chỉ cần quan tâm tới phần việc mình đang đảm nhiệm và bỏ mặc vấn đề tổng thể. Mặc dù vẫn đang phát huy tính ưu việt trong các ngành công nghiệp sản xuất hàng loạt, mô hình này xuất hiện rất nhiều điểm yếu, đặc biệt trong nền công nghiệp tri thức, nơi mà sự thích ứng với thay đổi cần được

ưu tiên hàng đầu. Khi đó mô hình làm việc theo nhóm ưu tiên sự cộng tác trở nên thành công hơn rất nhiều. Đây là đặc điểm chung của những kỳ lân công nghệ – nơi những startup thay đổi cả thế giới chỉ với một nhóm người cộng tác với nhau với những công việc không hẳn được phân định rạch ròi thành những giai đoạn trong một quy trình đòi hỏi sự chính xác và tốn nhiều thời gian. Ngay cả những công ty lớn như Microsoft hay Google cũng không muốn duy trì quá nhiều những bộ phận cồng kềnh với hàng loạt những phòng ban có trách nhiệm riêng biệt cùng sự cộng tác yếu. Và lựa chọn của họ là chuyển dịch sang một mô hình *những startup trong lòng tập đoàn khổng lồ*.

Đan Mạch, nơi khai sinh ra công ty tôi đã làm việc, luôn là một quốc gia giàu có và hạnh phúc nhất thế giới. Chính phủ nơi đây đặc biệt xuất sắc trong việc phản ứng nhanh với những thay đổi, tôi tin đó chính là *chìa khoá vàng* trong quản lý.

Và từng cá nhân cũng vậy, chúng ta buộc phải thích nghi với mô hình làm việc kiểu mới, nơi việc cộng tác trong nhóm thực sự quan trọng để tối ưu khả năng của mỗi cá nhân. Cuộc sống của chúng ta biến đổi từng ngày và chúng ta không thể phớt lờ hiện thực đó bởi việc bám sát một kế hoạch dài hạn đã được đặt ra chí từ một vài tháng trước. Chỉ một năm trước đây, tôi hoàn toàn không có khái niệm về cuốn sách này và đặt một kế hoạch khác cho mình; nhưng giờ tôi nhận thấy đây là thời điểm không thể bỏ qua để viết cuốn sách đầu tiên về Agile bằng tiếng Việt trong lúc cộng đồng Agile tại Việt Nam lớn mạnh hơn bao giờ hết về số lượng nhưng lại thiếu những kiến thức cơ bản. Điều này cũng giống như tôi đã háo hức lên kế hoạch mua một chiếc điện thoại iPhone 6S và rồi lại huỷ kế hoạch đó ngay khi Apple công bố chính thức, vì chiếc iPhone 6S vốn không có gì đặc biệt; và tôi dùng số tiền đó để mua một chiếc máy ảnh mới. Chúng ta hoàn toàn không biết điều gì sẽ chờ đợi mình trong một cuộc sống đầy biến động. Vì vậy, lập và bám sát một kế hoạch chi tiết cho cả một năm dường như là điều không khả thi; thay vào đó là đặt mục tiêu, lên kế hoạch sơ bộ và chỉ có kế hoạch chi tiết cho một tuần (hay thậm chí một ngày) kế tiếp. Việc thích nghi và phản ứng với những thay đổi hàng ngày quan trọng hơn rất nhiều. Nhưng tất nhiên, không có một phương pháp nào phù hợp với mọi cá nhân hay mọi dự án, mọi tổ chức. Cái chúng ta cần tìm kiếm là *phương pháp giúp chúng ta tìm ra cách làm việc tốt nhất cho từng cá nhân, cho từng dự án và cho từng tổ chức cụ thể trong hiện thực khách quan luôn thay đổi*.

Dù đề cao sự ứng biến linh hoạt cho từng hoàn cảnh, biến động trong thời gian ngắn như ngày hay tuần; tôi không phủ nhận việc mỗi cá nhân hay tổ chức vẫn cần một tầm nhìn cho những mục tiêu dài hạn. Tôi chỉ nhấn mạnh rằng con

đường tới mục tiêu cần được xây đắp bởi từng viên gạch được quyết định bởi sự phù hợp với từng thời điểm hơn là được quyết định bởi việc bám sát một bản kế hoạch chi tiết. Nhiều năm gần đây, tôi vẫn giữ thói quen hạn chế làm việc vào hai ngày trong năm (phương pháp được tôi sử dụng hàng năm để lên kế hoạch năm cho mình), và thêm vào một khoảng thời gian 1 giờ mỗi tuần. 1/2 giờ vào ngày thứ 7 để nhìn lại những gì đã làm trong tuần, và 1/2 giờ vào sáng Chủ nhật để lên một kế hoạch sơ bộ cho tuần tới. Phương pháp đó đảm bảo cho tôi có cách xử lý với rất nhiều công việc đột xuất nhưng vẫn giữ được mục tiêu dài hạn.

Dự định ban đầu của tôi chỉ là một cuốn sách về Scrum hoặc Agile trong những dự án phát triển phần mềm như tự thân phương pháp này đã có. Nhưng tôi nhận thấy Agile còn có rất nhiều tác dụng trong cả cuộc sống cá nhân đến những tổ chức nhỏ và lớn, thậm chí rất lớn. Bởi vậy, nếu chỉ viết riêng về lĩnh vực phát triển phần mềm, e rằng còn quá nhiều điều thiếu sót. Đặc biệt đây là cuốn sách đầu tiên về Agile bằng tiếng Việt, cuốn sách này nên cung cấp một cái nhìn tổng quát hơn về sức mạnh của một phương pháp tích cực trong những mô hình làm việc cần sự ứng biến cao với những thay đổi. Và quan trọng hơn hết, là phương pháp này cho phép tìm ra mô hình làm việc phù hợp nhất cho từng cá nhân hay tổ chức cụ thể.

CUỐN SÁCH NÀY DÀNH CHO AI?

Đây không phải là cuốn sách về công nghệ, kỹ thuật nhưng mọi lập trình viên đều nên đọc. Theo tôi, điểm yếu của lập trình viên Việt Nam không chỉ là công nghệ hay kỹ thuật. Điều chúng ta còn thiếu là phương pháp làm việc khoa học và cách thức thực hành có chủ đích. Công nghệ và kỹ thuật luôn phát triển rất nhanh và cần cập nhật thường xuyên, đặc biệt trong ngành công nghệ thông tin (CNTT); nhưng nhờ Internet khoảng cách này ngày càng được thu hẹp giữa những nước phát triển và Việt Nam. Chúng ta dễ dàng tiếp cận những thay đổi từ Java, Swift... tới những công nghệ phức tạp hơn; nhưng lại bỏ qua cách thức làm việc để tiếp cận và thực hành chúng một cách đúng đắn để tạo ra những sản phẩm tốt. Và khi một phần của thế giới đã bước sang giai đoạn *làm việc tự chủ với phản hồi nhanh* thì phần lớn kỹ sư CNTT Việt Nam vẫn đang loay hoay với việc áp dụng kỹ thuật cụ thể trên một yêu cầu và giải pháp rõ ràng. Điều này làm giảm lợi thế cạnh tranh của kỹ sư CNTT Việt Nam khi chúng ta thiếu những kỹ năng cộng tác, làm việc nhóm, tự quản... – những kỹ năng giờ đây đã được *cứng hoá*, là bắt buộc, không còn là kỹ năng mềm bổ trợ.

Những nhà quản lý, những người chịu trách nhiệm về sản phẩm, dự án phần mềm. Bởi trong hơn 10 năm gần đây, cách thức tổ chức dự án và phát triển phần mềm đã có nhiều thay đổi. Tất cả những gì chúng ta đang làm, với một cách thức

tổ chức dự án cồng kềnh, phân hoạch rõ ràng và bám sát kế hoạch giờ đây đã không còn phù hợp; thay vào đó là sự linh hoạt, gọn nhẹ dựa trên cộng tác giữa những cá nhân nhằm tối ưu giá trị mang lại cho khách hàng.

Những nhà quản lý cao cấp, những người mong muốn tổ chức hướng tới lợi ích của từng nhân viên bằng một môi trường làm việc thân thiện và vui vẻ. Thị trường lao động về cơ bản đã thay đổi trong những năm gần đây và đặc biệt thay đổi rất mạnh tại thị trường CNTT non trẻ như Việt Nam; quan niệm về *chất lượng cuộc sống* ngày càng được nâng cao trong từng cá nhân; và lương, thưởng đôi khi không còn là điều kiện tiên quyết. Ngày càng nhiều nhân viên bước dần lên những bậc cao hơn của tháp Maslow nơi họ muốn đứng ra nhận trách nhiệm bằng việc thể hiện khả năng trong một môi trường khoáng đạt hơn thay vì chỉ chăm chỉ làm theo mệnh lệnh của cấp trên.

Những ai không nên đọc cuốn sách này? Những người có hiểu biết sâu sắc về Agile. Đây không phải là một cuốn sách sâu sắc về Agile; những kiến thức này đã xuất hiện rất nhiều trên thế giới. Thay vào đó, cuốn sách này vẽ ra một bức tranh tổng thể, điểm ra những vấn đề cơ bản của Agile trong thế giới phát triển phần mềm và cá nhân. Điều tôi nghĩ tới khi viết cuốn sách này đó là, mặc dù cộng đồng Agile đã phát triển rất nhanh tại Việt Nam, song các tổ chức vẫn dè dặt ở mức tìm hiểu; bởi cộng đồng thiếu một góc nhìn toàn cảnh, những phương pháp, những thay đổi tích cực cũng như rủi ro nhận được khi áp dụng Agile trong tổ chức. Cuốn sách này được coi như điểm khởi đầu cho những người mới tìm hiểu về Agile có được góc nhìn rộng về những điều cần quan tâm; và khi thực sự quan tâm tới một vấn đề, phương pháp cụ thể, bạn có thể biết cách tìm tới những kiến thức chuyên sâu ở một nơi khác.

NỘI DUNG

Agile Y không thực sự là một cuốn sách về Agile đúng nghĩa mà bao gồm nhiều nội dung trong những bài viết của tôi trên blog cá nhân, những buổi nói chuyện, những nguồn tư liệu tham khảo về Agile được cấu trúc lại theo một cách hợp lý hơn. Tại đây bạn có thể tìm thấy những nội dung về Agile được cấu trúc theo 3 phần gồm 11 chương như sau:

Phần I: Agile trong phát triển phần mềm

Phần I là những nội dung quan trọng nhất trong cuốn sách này, dành cho lập trình viên và những nhà quản lý dự án phát triển phần mềm, gồm 5 chương.

Chương 1 đặt ra những vấn đề cơ bản về những hạn chế trong việc phát triển phần

mềm theo phương pháp truyền thống, những lý do Agile cần được áp dụng trong việc phát triển phần mềm hiện đại. Những nội dung, phân tích này không thực sự mới bởi Agile đã được thừa nhận là phương pháp phát triển phần mềm hiện đại; bạn có thể bỏ qua chương này nếu đã biết về Agile.

Chương 2 giới thiệu về Agile, tuyên ngôn Agile và điểm qua một số phương pháp nằm trong *chiếc ô Agile* giúp bạn hiểu cách Agile tiếp cận để giải quyết những vấn đề hạn chế của phương pháp phát triển truyền thống đã được đặt ra trong *Chương 1*.

Chương 3 giới thiệu về Scrum, phương pháp phát triển phần mềm phổ biến nhất trong *chiếc ô Agile*, giúp bạn hiểu về cách một nhóm phát triển phần mềm vận hành và những nguyên lý, giá trị tạo nên nền tảng sức mạnh của Scrum. Tôi khuyến nghị mọi tổ chức mới thực hành Agile tìm hiểu kỹ và sử dụng Scrum.

Chương 4 giới thiệu về Kanban, Scrumban – những phương pháp có sự tăng trưởng nhanh nhất trong *chiếc ô Agile*. Đây thường là phương pháp được những nhóm phát triển phần mềm lựa chọn sau khi đã thực hành thành thực Scrum.

Chương 5 bàn về những vấn đề kỹ thuật thường được áp dụng trong những nhóm Agile nhằm đạt hiệu quả cao trong thực hành Agile. Hầu hết những kỹ thuật được giới thiệu trong chương này thuộc về XP (eXtreme Programming), những kỹ thuật không thể thiếu với những nhóm thực hành Agile, dù theo bất cứ phương pháp nào.

Phần II: Agile trong tổ chức

Khác với Phần I giới thiệu về Agile “nguyên thuỷ”, nơi việc phát triển phần mềm tập trung vào nhóm. *Phần II* cho thấy bức tranh về việc những tổ chức lớn hơn thực hành Agile. *Phần II* dành cho những nhà quản lý cấp cao thấy được bức tranh toàn cảnh trong việc mang Agile vào tổ chức.

Chương 6 phân tích về cách những tổ chức ngày nay vận hành bằng cách sử dụng những nhóm tự tổ chức liên chức năng nhằm tăng sự linh hoạt qua việc tối đa hoá việc cộng tác và chịu trách nhiệm. Những vấn đề về trao quyền cũng như cộng tác cũng được đặt ra trong chương này.

Chương 7 giới thiệu về việc mở rộng Agile, cho thấy cách Agile áp dụng trong những tổ chức lớn, những dự án lớn. Trong khoảng hơn 10 năm gần đây, Agile đã chứng minh được sự phù hợp và hiệu quả trong thời đại ngày nay với quy mô nhóm. Và giờ đây, Agile đang từng bước đập tan hoài nghi rằng Agile chỉ phù hợp với những nhóm nhỏ, dự án nhỏ.

Chương 8 bàn về việc chuyển đổi tổ chức từ cấu trúc truyền thống sang môi trường thực sự Agile nhằm nâng cao tính cạnh tranh. Chuyển đổi trong tổ chức chưa bao giờ là việc là dễ dàng, và không có chiếc đua thần nào giúp mang tới hiệu quả tức thì. Ở đây tôi chỉ điểm qua một số ưu thế và khó khăn cũng như cạm bẫy có thể gặp phải trong quá trình thay đổi.

Phần III: Agile cho cá nhân

Phần III không nằm trong kế hoạch về nội dung ban đầu, song tôi muốn dành phần này cho lập trình viên – những người đang bị cuốn trôi bởi quán tính công việc và rất ít quan tâm tới hiệu quả công việc cũng như chất lượng cuộc sống cá nhân. Phần này giúp bạn áp dụng Agile vào cuộc sống, thực hành có chủ ý nhằm nâng cao lợi thế cạnh tranh cá nhân.

Chương 9 đặt ra những vấn đề cơ bản về việc quản lý cá nhân trong cuộc sống ngày nay; những vấn đề cần quản lý để nâng cao hiệu quả công việc và chất lượng cuộc sống.

Chương 10 giới thiệu về cách áp dụng Kanban cá nhân nhằm quản lý công việc tốt hơn; trực quan và hiệu quả. Về cơ bản bạn sẽ gặp lại những khái niệm được đề cập đến trong Chương 4; nên sẽ thật thiếu sót nếu những lập trình viên dù đã thấy giá trị Kanban trong nhóm phát triển phần mềm lại bỏ lỡ cơ hội áp dụng nó vào cuộc sống cá nhân.

Chương 11 đặt ra những nguyên lý cơ bản đứng sau Kaban cá nhân cũng như những kỹ thuật được áp dụng trong cuộc sống hàng ngày nhằm nâng cao hiệu quả công việc. Tất nhiên, không có một phương pháp nào phù hợp với mọi cá nhân, song những nguyên lý cơ bản thì không thay đổi.

Phụ lục

Phần kết của cuốn sách bao gồm 2 phụ lục.

Phụ lục tham khảo điểm qua những nguồn tài liệu và đánh giá cơ bản giúp bạn tìm hiểu tiếp để có hiểu biết sâu sắc về Agile trong quá trình áp dụng cho cá nhân, nhóm hay tổ chức.

Phụ lục thuật ngữ Agile điểm qua những thuật ngữ được sử dụng trong Agile cùng giải thích ngắn gọn giúp bạn dễ tiếp cận với những tài liệu chuyên sâu, nơi những thuật ngữ này không được giải thích.

ĐỊNH DẠNG NỘI DUNG

Xuyên suốt những chương sách của Agile Y, bạn có thể tìm được những nội dung phong phú, về nhiều khía cạnh khác nhau của việc áp dụng những tư tưởng Agile trong việc phát triển phần mềm, trong xây dựng tổ chức và cho cá nhân. Do đó, bất cứ ai cũng có thể nhận được một nội dung hữu ích, dù ít hay nhiều, sau khi đọc xong Agile Y.

Thiết kế của Agile Y đảm bảo các phần nội dung độc lập một cách tương đối qua 3 phần và từng chương nhằm giúp những ai đã có hiểu biết về Agile và quan tâm tới những nội dung cụ thể nhanh chóng tiếp nhận được ý tưởng. Nhằm tăng tính tương tác, tôi sử dụng hai chỉ dẫn xen ngang giữa các phần nội dung là *Gợi ý* và *Thử thách*.

Gợi ý là những chỉ dẫn ngắn gọn để bạn có thể thực hành được phần nội dung tương ứng.

Thử thách dưới dạng câu hỏi hoặc bài tập giúp bạn hiểu hơn về nội dung tương ứng.

Kết thúc mỗi chương là phần *Tổng kết* những nội dung chính, nơi tôi chọn ra những điểm quan trọng nhất được mô tả chỉ trong một hoặc hai câu, giúp bạn dễ dàng ghi nhớ phần nội dung.

Ngoài ra, qua kinh nghiệm cá nhân, tôi nhận thấy có nhiều cách hiểu chưa đúng về Agile trong những nội dung cụ thể; bạn có thể tìm thấy trong phần *Hiểu đúng* kèm những giải thích ngắn gọn. Những nội dung này hầu hết được tôi chỉ ra trong phần nội dung chính; song theo kinh nghiệm cá nhân, tôi nghĩ nên có một phần riêng đề cập tới những hiểu nhầm hay gấp để giúp bạn dễ dàng ghi nhớ hơn.

NGÔN NGỮ SỬ DỤNG

Agile Y chứa đựng cách truyền tải nội dung gần với những bài viết, trò chuyện, thường được tôi sử dụng trong những bài viết trên blog. Rất nhiều từ ngữ, thuật ngữ được sử dụng trong cuốn sách này được giữ nguyên (hoặc lai tạp) bằng tiếng Anh (có thể kèm theo chú thích); nên nhiều chỗ bạn có thể cảm thấy khó chịu vì một câu tiếng Việt chứa những từ tiếng Anh. Song tôi không tham vọng sẽ dịch những thuật ngữ này sang tiếng Việt trong cuốn sách; ít nhất, những

thuật ngữ tiếng Anh cũng giúp bạn dễ Google hơn để tìm được những tài liệu tham khảo liên quan khi nguồn tài liệu về Agile bằng tiếng Việt chưa thực sự phong phú trong thời điểm này.



AGILE

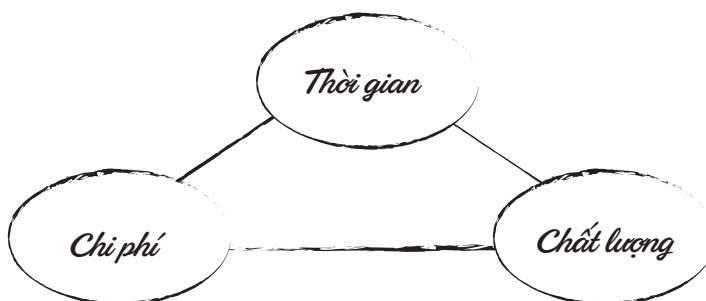


h

PHÁT TRIỂN
PHẦN MỀM
CÂN LINH HOẠT

Tôi nghĩ rằng không cần phải chứng minh tầm quan trọng của Công nghệ thông tin (CNTT) trong cuộc sống của mỗi cá nhân và doanh nghiệp ngày nay, bạn chắc chắn dễ dàng đồng ý với tôi về nhận định này. Và ngành công nghiệp phần mềm giờ đây đang nắm giữ *xương sống* của ngành CNTT, len lỏi vào từng ngõ ngách của cuộc sống, từ bàn tay nhỏ bé của mỗi chúng ta tới những nhà kho khổng lồ của Amazon. Năm 2016, thế giới sẽ chào đón thiết bị di động thứ 2,5 tỷ, gần như toàn bộ số lượng đó được sản xuất trong 10 năm gần đây; 1 tỷ dân số thế giới đang sở hữu ít nhất một chiếc điện thoại thông minh với khoảng 40 phần mềm đơn giản được cài đặt sẵn khi xuất xưởng từ khoảng 5000 nhà sản xuất điện thoại thông minh khác nhau. Đó là ví dụ đơn giản nhất dù nó chỉ cho thấy chưa tới 5% khối lượng những dự án phần mềm đang được triển khai trên thế giới.

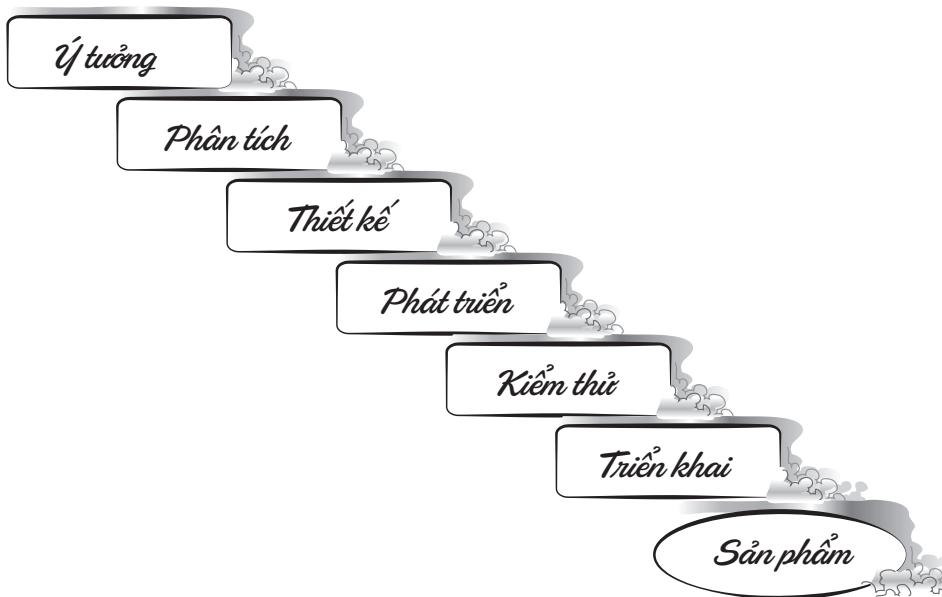
Nhưng việc phát triển trong các dự án phần mềm có gì khó khăn? Nghiên cứu vào năm 2010 cho thấy ngành công nghiệp phần mềm là nơi đáng thất vọng nhất với những nhà quản trị dự án tài ba, bởi chỉ 40% số dự án thành công, 25% số dự án thất bại; số dự án còn lại cần nhiều hơn những nỗ lực dự tính để hoàn thành. Một dự án phần mềm được coi là thành công khi đảm bảo 3 yếu tố chính: *đúng thời hạn (on time)*, *trong chi phí dự toán (on budget)* và *đảm bảo chất lượng (quality)*. Trong đó yếu tố đảm bảo chất lượng là khó đo lường nhất. Dù rất dễ tính, xem xét yếu tố *đảm bảo chất lượng* chỉ là phần mềm đáp ứng được những yêu cầu đã đề ra không có lỗi, cũng chỉ có 40% các dự án thành công. Thật khó tưởng tượng được sự lãng phí khi hàng năm thế giới chi ra hàng trăm tỉ đô la cho những dự án phần mềm, thì chỉ 40% số tiền đó thực sự mang lại hiệu quả. Tất nhiên, ngày nay tỉ lệ những dự án thành công đã tăng lên rất nhanh trên toàn thế giới; song đáng tiếc, số liệu của năm 2010 có vẻ vẫn đang đúng với thực tế tại Việt Nam.



Hình 1.1: Các yếu tố của dự án phát triển phần mềm

Nhưng vì sao những dự án phát triển phần mềm lại rủi ro và tốn kém như vậy? **Bởi sự cứng nhắc của những phương pháp tiếp cận.**

Hãy cùng nhìn lại một trong những phương pháp truyền thống đã từng được coi là “kinh thánh” trong phát triển phần mềm: *phương pháp thác nước (waterfall)*. Phương pháp này chia giai đoạn phát triển phần mềm thành những công đoạn như sau:



Hình 1.2: Phương pháp phát triển phần mềm thác nước

Việc phát triển phần mềm được trải qua những giai đoạn tách biệt và rất cụ thể được thực hiện bởi từng phòng ban chuyên trách; từ phân tích ý tưởng, thiết kế tổng thể, thiết kế chi tiết, lập trình, kiểm thử đến triển khai và bảo trì.

Nếu bạn là một lập trình viên, hãy thử lên kế hoạch cho việc phát triển một hệ thống phần mềm đáp ứng yêu cầu tính thuế thu nhập cá nhân.

Đó thực sự là một phần mềm không phức tạp. Vậy chúng ta cần bao nhiêu thời gian để hoàn thành? 1 tuần? 1 tháng? Ngay sau khi chúng ta hoàn thiện phần mềm, chúng ta biết rằng thuế thu nhập cá nhân đã được điều chỉnh với khoản 9.000.000 không bị tính thuế, và những cá nhân có người phụ thuộc đều tiếp tục được miễn trừ thuế thu nhập cá nhân ở các mức khác nhau. Và hệ thống bỗng

dựng hoàn toàn sụp đổ, chúng sẽ phải bắt đầu lại từ đinh thac nước để phát triển một hệ thống khác hoặc bàn giao một sản phẩm không thể sử dụng bởi nó không đáp ứng nhu cầu thực tế. Ngay sau đó, đề xuất giảm 50% thuế thu nhập cá nhân với nhân sự làm việc trong những doanh nghiệp CNTT cũng sắp được phê chuẩn.

Mô hình phát triển phần mềm *thác nước* đã từng là kinh thánh trong việc phát triển phần mềm, nhưng rất ít được sử dụng trực tiếp bởi vòng đời phát triển sản phẩm quá dài, cần nhiều nguồn lực cũng như không hiệu quả bởi quá nhiều những đoán định cùng khối lượng công việc rất lớn chỉ vào từng thời điểm xác định. Một số phương pháp cải tiến từ mô hình thác nước như RUP (Rational Unified Process) không thực sự hiệu quả, thậm chí khiến việc phát triển phần mềm cồng kềnh và phức tạp hơn bao giờ hết. Giờ đây, những phương pháp đó đã không còn phù hợp nữa, ít nhất bởi 3 lý do chính sau:

Phản ứng kém với những thay đổi yêu cầu

Bởi chúng ta đã có một *kế hoạch hoàn hảo* cho những yêu cầu đã biết và tin rằng việc bám sát kế hoạch đó là con đường dẫn tới thành công, trong khi khách hàng chỉ thực sự nhìn thấy những gì là phần mềm ở điểm cuối của một quy trình tổn quá nhiều thời gian. Trong khi người dùng đang đối mặt với hàng loạt những biến đổi xảy ra hàng ngày ngoài thế giới, những nhà phát triển phần mềm hoàn toàn bị cách ly bởi những yêu cầu từ đinh thac nước và không được cập nhật kịp thời. Phần mềm khi được triển khai đã là quá muộn để đáp ứng những nghiệp vụ mới. Phát triển phần mềm theo phương pháp truyền thống không giống như xây một ngôi nhà, khi chúng ta có thể nhìn thấy từng tầng mới được dựng lên hàng ngày. Việc phát triển phần mềm truyền thống giống như chúng ta dựng một bức tường cách ly người dùng cho tới khi ngôi nhà hoàn thành và *chìa khoá trao tay*; khi đó nếu tầng 1 quá thấp, chúng ta chỉ còn cách phá bỏ hoàn toàn ngôi nhà và xây lại. Một cách ví von không thực sự chính xác nhưng nó cho thấy một phần thực trạng những dự án phần mềm thất bại hiện nay; vậy tại sao chúng ta không để cho người dùng phản hồi và chỉnh sửa "tầng một" ngay khi có thể vì dù sao sửa chữa phần mềm thường đơn giản và đỡ tốn kém hơn việc sửa đổi một ngôi nhà.

Những nhà quản trị dự án phần mềm tài ba theo phương pháp truyền thống thường có hai cách giải quyết việc thay đổi yêu cầu của người dùng. Và theo tôi, cả hai cách đều thực sự không tốt.

Cách thứ nhất: dứt khoát loại bỏ những thay đổi bằng cách khéo léo chối những yêu cầu một cách tỉ mỉ bằng hợp đồng, bám sát kế hoạch để hoàn thành sản phẩm. Đây đã từng là công thức vàng để có một dự án thành công nhưng nó thực sự cứng nhắc và vô trách nhiệm. Bởi dự án chỉ thực sự thành công về mặt giấy tờ

khi đảm bảo 3 yếu tố trên nhưng sản phẩm của dự án, phần mềm – công sức của nhiều con người trong một thời gian dài, lại có thể không bao giờ được sử dụng. Để đáp ứng được nhu cầu hiện tại, một hợp đồng mới lại được ký kết với những thay đổi; và khách hàng sẽ lại chờ đợi một khoảng thời gian tương tự để nhận được sản phẩm của một *nhu cầu quá khứ của tương lai*. Vòng lặp này chỉ đến hồi kết khi khách hàng thực sự hết tiền đầu tư hoặc hoàn toàn mất kiên nhẫn vào một hệ thống thông tin mà họ từng kỳ vọng sẽ trợ giúp cho công việc của mình. Sở dĩ đây đã từng là công thức thành công vì những nhà phát triển phần mềm theo kiểu *chốt hợp đồng* giờ đây không còn được ưa chuộng bởi chính họ đã đánh mất lợi thế cạnh tranh khi cung cấp giá trị cho khách hàng bằng cách làm cho mình an toàn. Và khi có nhiều tổ chức cung cấp dịch vụ phát triển phần mềm theo kiểu linh hoạt hơn, họ không còn là lựa chọn của bên A.

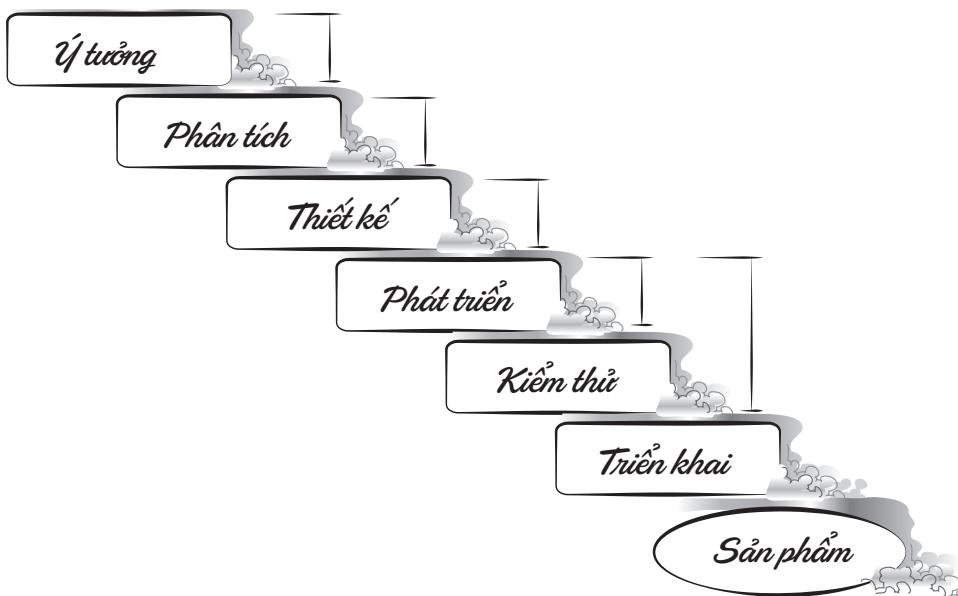
Cách thứ hai: chào đón mọi thay đổi và bắt đầu tại đỉnh thác nước mỗi khi có yêu cầu mới. Đây là cách làm phổ biến hiện nay và nó gây ra những cơn đau đầu không ngừng cho đội phát triển phần mềm đến độ nhóm phát triển phần mềm công khai chỉ trích khách hàng. Tin tôi đi, những câu nói trong nhóm kiểu như *thằng điên này, vừa mới hôm qua bảo khác mà là rất phổ biến bởi tôi đã từng làm trong nhiều nhóm phát triển phần mềm khác nhau, và tình trạng chung đều là bằng mặt và không bằng lòng*. Những nhà quản trị dự án luôn biết cách mềm dẻo trong việc chào đón những yêu cầu bổ sung của khách hàng và cứng rắn trong việc yêu cầu đội phát triển lao vào một guồng quay chẳng mấy khi có lối thoát. Và khách hàng, rồi cũng chỉ nhận được *phần mềm trên giấy tờ* hoặc ở một thời điểm rất trễ. Tôi sử dụng khái niệm *phần mềm trên giấy tờ* để chỉ những phần mềm vốn chỉ tồn tại trong những bản thiết kế, “đúng rồi, trông nó sẽ như thế này” nhưng đó chỉ đơn thuần là một đống giấy tờ, đó không phải là một hệ thống phần mềm được cài đặt và thực sự giúp ích cho người dùng. Hoặc nếu có, thì đó là sản phẩm của sự cố gắng hơn mức tưởng tượng của đội phát triển khi họ liên tục phải làm việc với cường độ cao trong suốt thời gian dài.

Thời điểm nhận phản hồi quá trễ; khối lượng phản hồi lớn

Ngay khi khách hàng không có sự thay đổi trong yêu cầu, phương pháp thác nước vẫn chưa đựng rủi ro rất lớn bởi khách hàng chỉ thực sự phản hồi về sản phẩm ở điểm cuối của một quy trình quá dài. Và dù mỗi chức năng chỉ phải chỉnh sửa một vài điểm, nhưng việc chuyển giao cùng lúc quá nhiều chức năng cũng tạo ra khối lượng công việc khổng lồ, thậm chí thời gian để sửa đổi cũng không kém hơn thời gian tạo ra một sản phẩm mới vì quy trình lại được bắt đầu từ *đỉnh thác*.

Có rất nhiều lý do cho việc phần mềm không phù hợp ngay sau khi chuyển giao dù khách hàng hoàn toàn không thay đổi yêu cầu ban đầu. Lý do chính là việc

không hiểu đúng yêu cầu đã xuất phát ngay từ đinh thác. Chúng ta cần nhớ rằng những lập trình viên không phải là những siêu nhân, không thể đòi hỏi họ có được hiểu biết về nghiệp vụ ngân hàng như một chuyên viên ngân hàng (và rồi hết dự án đó họ lại làm sản phẩm về quản lý kho). Một chuyên viên ngân hàng là người dùng thực sự lại không biết cách mô tả yêu cầu của mình tới lập trình viên. Và đội phát triển sinh ra một vị trí cho một công đoạn trong việc phát triển phần mềm là BA (*business analysis, phân tích nghiệp vụ*). Khi phần mềm cuối cùng không giống như kỳ vọng, người ta thường không hay nhìn lại cả quá trình, mà thường phân tích xem bộ phận BA đã làm việc như thế nào và sản sinh ra hàng tá những phương pháp nhằm đảm bảo yêu cầu từ người dùng được hiểu đúng. Nhưng nếu nhìn lại các bước trong mô hình thác nước, chúng ta sẽ thấy có những điểm chuyển giao như:



Hình 1.3: Mô hình phát triển phần mềm thác nước

Những điểm chuyển giao này thường gây ra những hiểu nhầm:

- Yêu cầu của khách hàng có thể không được hiểu đúng.
- Nghiệp vụ được mô tả có thể không được cụ thể hoá thành thiết kế đúng.
- Thiết kế có thể không được chuyển thành mã nguồn đúng.
- Mã nguồn có thể không được kiểm thử đúng.

Nếu tỉ lệ sai sót chỉ là 5% ở mỗi giai đoạn này, nó sẽ nhanh chóng tăng theo mức lũy thừa trong tổng thể cả quá trình và phát triển thành một con số khủng khiếp.

Nhưng cách chúng ta giảm thiểu sai sót mới là vấn đề cần xem xét. Thông thường, hiểu nhầm cần được giải quyết bằng cách *trao đổi*, nhưng trong mô hình thác nước, hiểu nhầm được giải quyết dựa trên *tài liệu*. Và đó thực sự là sai lầm lớn nhất. Đặc trưng của *quy trình* và mô hình thác nước là gì? Đó là *trách nhiệm* được *phân tách rạch rồi giữa các bộ phận* với *đầu ra là những tài liệu không hướng tới giá trị cho người dùng*. Thay vì viết ra những đoạn mã chương trình hữu ích cho người dùng, những lập trình viên chỉ *gõ* lại những dòng lệnh đã được đặc tả theo thiết kế chi tiết – đó là trách nhiệm của họ. Và thật nực cười là những người làm ra phần mềm lại không hiểu công việc mình làm vì điều gì, và cho ai. Và khi họ làm tới phần mềm thứ 100 để quản lý nhà hàng, hiểu biết của họ về nghiệp vụ quản lý nhà hàng vẫn như ngày đầu.

Lãng phí

Có hai vấn đề về gây ra lãng phí lớn: *tài liệu* và *chức năng không được sử dụng*.

Mô hình phát triển phần mềm truyền thống có quá nhiều tài liệu, trong khi tài liệu lại không phải là phần mềm. Người ta làm ra tài liệu chỉ để khẳng định rằng *tôi đã hoàn thành trách nhiệm của mình và chuyển giao nó cho giai đoạn tiếp theo*. Thiết kế chi tiết được vẽ trên giấy về cơ bản không mang lại giá trị cho khách hàng, nhưng họ lại phải trả tiền cho việc đó. Và như tôi đã đề cập ở trên, để giảm thiểu hiểu nhầm, phương pháp truyền thống yêu cầu nhiều tài liệu hơn nữa, chi tiết hơn nữa. Và đó là lý do chi phí phát triển phần mềm dường như tăng theo cấp số nhân theo thời gian cũng như độ phức tạp.

Theo nghiên cứu từ Pragmatic Bookshelf, chỉ khoảng 36% số chức năng của phần mềm được sử dụng thường xuyên. Thật ra thống kê này không quá bất ngờ, bởi theo nguyên lý 20/80 thì chỉ khoảng 20% số chức năng được sử dụng bởi khoảng 80% số lượng người dùng (và mang lại 80% giá trị cho từng người dùng cụ thể). Tôi hy vọng bạn không ngạc nhiên khi biết rằng 45% số chức năng hiện có của Microsoft Word gần như không được sử dụng. Vậy tại sao chúng ta nên chờ đợi sản phẩm Microsoft Word được hoàn thiện 100% số chức năng trước khi đưa chúng đến tay người dùng? Tại sao không chuyển giao sản phẩm Microsoft Word với 20% chức năng trong 2 tháng đầu tiên, và phát triển 80% chức năng còn lại sau đó?

Đây chỉ là những lý do cơ bản cho thấy sự lỗi thời và hạn chế của phương pháp phát triển phần mềm truyền thống. Tuy vậy trong suốt một thời gian dài, ngành

công nghiệp phần mềm vẫn duy trì sự bảo thủ của mình chỉ với một lý do duy nhất *sản xuất phần mềm là một công việc phức tạp*. Không ai phủ nhận điều này, *sản xuất phần mềm là một công việc phức tạp*, và chúng ta cần một cách tiếp cận khác giúp việc này đơn giản hơn, chứ không phải làm nó phức tạp thêm nữa. Bởi thế giới giờ đây đã thay đổi, bất kỳ sự phản ứng chậm chạp nào cũng sẽ đều không được chấp nhận.

TỔNG KẾT

Phương pháp phát triển phần mềm truyền thống phân chia quá trình phát triển phần mềm thành những giai đoạn độc lập, do đó phát sinh những vấn đề *lãng phí, phản ứng kém với những thay đổi trong yêu cầu bởi khởi lượng phản hồi lớn tại thời điểm trễ*. Việc bắt đầu lại từ đỉnh thác luôn chứa đựng nhiều bất ổn khiến tỉ lệ dự án thành công không cao.

Về cơ bản, thế giới cần một *phương pháp phát triển phần mềm mới*, giải quyết bài toán theo cách khác, thay vì chỉ *tối ưu cục bộ* trong một giai đoạn nhất định.

02

AGILE
LÀ
GIẢI PHÁP

Trong nỗ lực cải thiện việc phát triển phần mềm để đáp ứng những nhu cầu thay đổi của thực tế, một loạt những phương pháp đã được nghiên cứu và có những hiệu quả nhất định trong thực tế như: theo dõi người dùng, đặt câu hỏi để hiểu đúng yêu cầu, thiết kế mở để dễ dàng thay đổi... Tôi không phản đối những cải tiến này vì chúng đều mang lại hiệu quả thực sự và giúp chất lượng của từng công đoạn được cải thiện rất nhiều. Tuy vậy, việc nâng cấp cục bộ trong phương pháp phát triển phần mềm truyền thống phải trả giá bằng sự phức tạp trong quy trình, đòi hỏi rất nhiều kỹ năng và tài liệu trong vòng đời phát triển sản phẩm. Cách giải quyết bế tắc này khiến lượng công việc bị nhân đôi, và cái giá phải trả không hề tương xứng. RUP, là một ví dụ rõ ràng cho những gì tôi đang nói tới, RUP chứa hàng loạt những cải tiến cục bộ khiến việc làm chủ một phần RUP cũng trở nên phức tạp. Cái chúng ta thiếu là cách giải quyết bài toán từ gốc, từ chính vòng đời của việc phát triển phần mềm.

Vì thế, dù có những cải thiện đáng kể, phương pháp phát triển phần mềm truyền thống vẫn bị chỉ trích là quá quan liêu, cứng nhắc và không phản ứng đủ nhanh với những thay đổi thực tế. Vào cuối những năm 1990, một nhóm chuyên gia trong việc phát triển phần mềm với tư tưởng cấp tiến đã nghĩ tới việc tìm ra những phương pháp mới để *giải quyết bài toán từ gốc*, thay đổi tổng thể cách thức phát triển phần mềm thay vì chỉ tối ưu hoá một giai đoạn của vòng đời phát triển phần mềm truyền thống. Hàng loạt những nghiên cứu độc lập đã đưa ra những phương pháp cụ thể như Scrum, Extreme programming, Crystal... và tất cả những phương pháp này đều cho thấy sự gọn nhẹ, tập trung vào giá trị chính là chìa khoá để giải quyết vấn đề.

Thuật ngữ *phát triển phần mềm linh hoạt* (*agile software development*) lần đầu được đưa ra trong cuộc họp (thảo luận kết hợp giải trí, trượt tuyết) giữa 17 chuyên gia hàng đầu, những người đã nghiên cứu và áp dụng những phương pháp cụ thể, tại Wasatch Utah, theo lời kêu gọi của Bob Martin. Và sau nhiều tranh cãi, thảo luận, một bản *tuyên ngôn Agile* (*The Manifesto for Agile Software Development*) đã được đưa ra.

Bản dịch đầu tiên được công bố bởi Hanoi Scrum (đã hợp nhất thành Agile Vietnam) vào năm 2011, được biết đến rộng rãi tại Việt Nam như sau:

Chúng tôi đã phát hiện ra cách phát triển phần mềm tốt hơn bằng cách thực hiện nó và giúp đỡ người khác thực hiện. Qua công việc này, chúng tôi đã đi đến việc đánh giá cao:

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

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

Dịch từ: <http://agilemanifesto.org>



Hình 2.1: Tuyên ngôn Agile

Nếu bạn thấy những bản tuyên ngôn này quá ngắn và chung chung, thì có thể tham khảo *12 nguyên lý đứng sau tuyên ngôn phát triển phần mềm linh hoạt* dưới đây. Tôi sẽ phân tích các nguyên lý này kỹ hơn ở phần sau của cuốn sách.

1. Ưu tiên cao nhất của chúng tôi là thỏa mãn khách hàng thông qua việc chuyển giao sớm và liên tục các phần mềm có giá trị.
2. Chào đón việc thay đổi yêu cầu, ngay cả khi sự thay đổi đó đến rất muộn trong quá trình phát triển. Các quy trình linh hoạt tận dụng sự thay đổi cho các lợi thế cạnh tranh của khách hàng.
3. Thường xuyên chuyển giao phần mềm chạy tốt tới khách hàng, từ vài tuần đến vài tháng, ưu tiên cho các khoảng thời gian ngắn hơn.
4. Nhà kinh doanh và nhà phát triển phải làm việc cùng nhau hàng ngày trong suốt dự án.
5. Xây dựng các dự án xung quanh những cá nhân có động lực. Cung cấp cho họ môi trường và sự hỗ trợ cần thiết, và tin tưởng họ để hoàn thành công việc.
6. Phương pháp hiệu quả nhất để truyền đạt thông tin tới nhóm phát triển và trong nội bộ nhóm phát triển là hội thoại trực tiếp.
7. Phần mềm chạy tốt là thước đo chính của tiến độ.
8. Các quy trình linh hoạt thúc đẩy phát triển bền vững. Các nhà tài trợ, nhà phát triển, và người dùng có thể duy trì một nhịp độ liên tục không giới hạn.
9. Liên tục quan tâm đến các kỹ thuật và thiết kế tốt để gia tăng sự linh hoạt.
10. Sự đơn giản – nghệ thuật tối đa hóa lượng công việc chưa xong – là cẩn bản.
11. Các kiến trúc tốt nhất, yêu cầu tốt nhất, và thiết kế tốt nhất sẽ được làm ra bởi các nhóm tự tổ chức.
12. Nhóm phát triển sẽ thường xuyên suy nghĩ về việc làm sao để trở nên hiệu quả hơn, sau đó họ sẽ điều chỉnh và thay đổi các hành vi của mình cho phù hợp.

Nhưng thực sự những điều gì đứng sau Tuyên ngôn Agile?

Tuyên ngôn phát triển phần mềm linh hoạt ghi nhận một tập hợp chung tổng

thể các giá trị và nguyên tắc cho tất cả các phương pháp linh hoạt như Scrum, XP... vào thời điểm đó. Bản tuyên ngôn chi tiết về bốn giá trị cốt lõi cho phép các nhóm phát triển phần mềm có hiệu suất cao.

- cá nhân và sự tương tác;
- phần mềm chạy tốt;
- cộng tác với khách hàng;
- phản hồi với các thay đổi.

Đây là những giá trị làm việc thực sự. Mỗi phương pháp linh hoạt tiếp cận các giá trị theo cách khác nhau, nhưng tất cả những phương pháp này có các quy trình và cách thực hành cụ thể thúc đẩy một hoặc nhiều giá trị.

Các cá nhân và sự tương tác hơn là quy trình và công cụ

Các cá nhân và sự tương tác rất cần thiết để các nhóm phát triển phần mềm có hiệu suất cao. Các nghiên cứu về *bão hòa thông tin liên lạc* trong một dự án cho thấy, khi không tồn tại vấn đề truyền thông, các nhóm có thể thực hiện tốt hơn bình thường khoảng 50 lần. Để tạo điều kiện thông tin liên lạc, phương pháp linh hoạt dựa trên một chu kỳ thường xuyên kiểm tra và thích ứng. Những chu kỳ có thể dao động từ vài phút với cấp lập trình tới vài giờ với một nhóm trong cuộc họp đứng hàng ngày, tới vài tuần với một *Iteration*.

Tuy vậy, việc chỉ tăng tần suất liên lạc và phản hồi các thông tin là không đủ để loại bỏ vấn đề truyền thông. Những chu kỳ kiểm tra và thích ứng chỉ thực sự tốt khi các thành viên đội thể hiện một số hành vi quan trọng:

- tôn trọng giá trị cá nhân;
- sự thật trong mọi giao tiếp;
- minh bạch hoá mọi thông tin, hành động và quyết định;
- tin tưởng rằng mỗi thành viên sẽ luôn ủng hộ nhóm;
- kiến tạo cam kết cho nhóm cũng như các mục tiêu của nhóm.

Để nuôi dưỡng những tư tưởng và hành vi này, công tác quản lý cũng cần thể

hiện sự linh hoạt thông qua việc cung cấp một môi trường có sự hỗ trợ tốt, tạo điều kiện hòa nhập cho tất cả thành viên trong nhóm. Chỉ khi đó, các nhóm mới có thể đạt được đầy đủ tiềm năng để làm việc với hiệu quả cao nhất.

Đối với các nhóm phát triển phần mềm, việc biến những tư tưởng này thành những hành vi thuần thực là không đơn giản. Hầu hết các nhóm thường lảng tránh những yếu tố về sự thật, minh bạch do bị giới hạn bởi các chuẩn mực văn hóa và những kinh nghiệm từ xung đột trong quá khứ. Nhưng nếu không có sự thật, không minh bạch hóa thông tin và quyết định thì chắc chắn sẽ không có sự tin tưởng và cam kết cũng chỉ là giả tạo. Bởi vậy, lãnh đạo và các thành viên trong nhóm cần được khuyến khích tạo ra những xung đột tích cực. Khi những thành viên trong nhóm tham gia vào các hoạt động xung đột tích cực, nhóm sẽ đạt được nhiều mục đích khác bên cạnh việc hoạt động hiệu quả hơn:

- *Cải tiến quy trình.* Chính những xung đột trong nhóm cho thấy một danh sách các trở ngại hoặc các vấn đề trong nhóm hoặc trong cả tổ chức. Việc giải quyết những vấn đề giúp nhóm tạo ra quy trình của chính mình và đạt hiệu suất làm việc cao nhất. Những xung đột xảy ra trong nhóm này cũng gợi ý đến những xung đột có thể xảy ra với những nhóm khác và cho thấy tiềm năng cho việc cải tiến quy trình bằng việc loại bỏ chúng theo thứ tự ưu tiên.
- *Đổi mới.* Những xung đột xảy ra do những tư tưởng trái ngược nhau chính là nguồn gốc cho sự đổi mới.
- *Cam kết.* Cam kết chỉ xảy ra khi mọi người đồng thuận về mục tiêu chung và sau đó đấu tranh để cải thiện giữa những cá nhân trong nhóm. Và chỉ những xung đột mới tạo ra sự đồng thuận về mục tiêu chung thực sự chứ không phải ở bề nổi.

Trong những giá trị này, *cam kết* là giá trị đặc biệt quan trọng. Bởi sau khi đã loại bỏ những yếu tố nặng nề trong quy trình phát triển phần mềm truyền thống, cam kết là thứ còn lại duy nhất đảm bảo mọi thứ hoạt động tốt. Nhưng nếu tổ chức đưa cho mỗi cá nhân quyền lựa chọn giữa một quy trình phức tạp và *tự do trong cam kết*, chúng ta không khó để có được câu trả lời.

Để tạo ra các nhóm có hiệu suất cao, phương pháp Agile hướng giá trị vào cá nhân và sự tương tác hơn là quá trình và công cụ. Thực tế là, tất cả các phương pháp Agile tìm cách tăng giao tiếp và cộng tác thông qua những chu kỳ thường xuyên kiểm tra và thích ứng. Tuy nhiên, chu kỳ này chỉ hoạt động khi những nhà

lãnh đạo khuyến khích các xung đột tích cực, bởi chỉ có xung đột tích cực mới có thể xây dựng một nền tảng vững chắc cho *sự thật, minh bạch, lòng tin, tôn trọng và cam kết* của nhóm.

Phần mềm chạy tốt hơn là tài liệu đầy đủ

Phần mềm chạy tốt là sự khác biệt lớn của Agile so với phương pháp phát triển phần mềm truyền thống. Tất cả những phương pháp được liệt kê trong tuyên ngôn Agile đều nhấn mạnh việc chuyển giao những phần nhỏ của một phần mềm chạy tốt trong những khoảng thời gian xác định.

Nhóm Agile cần thống nhất cách hiểu chính xác *phần mềm chạy tốt* (*working software*) trong nhóm (thường được gọi là *DoD – Definition of Done*), hay *phần tăng trưởng (increment)*. Thông thường, đây là một chức năng hoàn chỉnh, đã vượt qua toàn bộ công đoạn kiểm thử và người dùng có thể vận hành được.

Tất cả các nhóm phát triển phần mềm linh hoạt phải thiết lập định nghĩa về *phần mềm chạy tốt* một cách thống nhất. Ở mức người dùng cuối, một *phần tăng trưởng* hay một chức năng được coi là hoàn thành khi và chỉ khi chức năng đó vượt qua tất cả các *test case* bởi một người dùng thông thường. Ở mức tối thiểu, chức năng đó phải vượt qua *unit* và *intergration test case*. Những nhóm phát triển phần mềm linh hoạt bao giờ cũng thực hiện việc tích hợp và kiểm thử tích hợp cho mỗi phần tăng trưởng hoặc chức năng được thêm mới.

Dữ liệu tổng hợp từ nhiều dự án của một công ty nằm trong số những công ty có tỷ lệ lỗi thấp nhất thế giới, được chứng nhận CMMI ở mức 5, cho thấy lợi ích của việc thực hành Agile. Cụ thể, họ đã có thể tăng gấp đôi tốc độ sản xuất và giảm 40% số bug bằng cách sau:

- xác định trước các *acceptance test case* ngay khi xác định được chức năng cần phát triển;
- xây dựng những chức năng *tuần tự* và *theo độ ưu tiên*;
- thực hiện *acceptance tests* trên từng tính năng *ngay khi chúng được thực hiện*;
- fix bug là công việc có *độ ưu tiên cao nhất* và phải được thực hiện *sớm nhất* có thể.

Việc xác định trước những *acceptance test case* thường được gọi là *test-driven development* (*phát triển hướng kiểm thử*) và đòi hỏi nhóm phát triển phải luôn chú ý để vượt qua mọi kiểm thử được xác định trước. Phương pháp này rất tuyệt vời và gần như đi ngược với phương pháp phát triển phần mềm truyền thống (việc kiểm thử được thực hiện sau công đoạn lập trình) vì nó giải quyết triệt để vấn đề *hiểu nhầm* giữa các công đoạn như đã được đề cập ở Chương 1.



Hình 2.2: Phương pháp phát triển phần mềm truyền thống

Theo phương pháp truyền thống, từ 1 yêu cầu sẽ phát triển theo 2 hướng khác nhau là lập trình và kiểm thử, do đó gây ra những *hiểu nhầm* và định nghĩa khác nhau về *chức năng đúng*. Bằng cách định nghĩa trước việc kiểm thử, nhóm phát triển chỉ có 1 cách hiểu thống nhất.



Hình 2.3: Phương pháp Agile

Cũng bởi việc fix bug luôn được ưu tiên nên tại bất cứ thời điểm nào nhóm phát triển cũng biết chính xác *hiện trạng* của việc phát triển, có bao nhiêu bug, chất lượng phần mềm ra sao... thay vì chỉ biết vào phút cuối (trong giai đoạn kiểm thử riêng biệt) như phương pháp phát triển phần mềm truyền thống. Do đó, tại bất kỳ thời điểm nào, nếu cảm thấy tự tin về chất lượng, nhóm hoàn toàn có thể triển khai phần mềm. Vì vậy, phương pháp Agile được mô tả là *built-in quality*; tức là tự bản thân phương pháp đã giúp sản phẩm có được chất lượng đảm bảo.

Công tác với khách hàng hơn là đàm phán hợp đồng

Trong hai thập kỷ qua, tỷ lệ thành công của dự án đã tăng gấp đôi trên toàn

thế giới. Những cải tiến này chính là kết quả của các dự án nhỏ hơn và *chuyển giao thường xuyên hơn, cho phép khách hàng cung cấp thông tin phản hồi về phần mềm đều đặn hơn*. Các tác giả của tuyên ngôn Agile đã rất sớm nhấn mạnh rằng *khách hàng tham gia vào suốt quá trình phát triển phần mềm là điều cần thiết để thành công*.

Các phương pháp Agile đã sớm có giải pháp thúc đẩy giá trị này bằng cách định nghĩa một công việc ủng hộ khách hàng làm việc cùng với nhóm phát triển. Công việc này được định nghĩa rõ bởi vai trò của Product Owner trong Scrum, sẽ được chúng ta tìm hiểu rõ hơn trong chương tiếp theo. Đôi khi *khách hàng* là một tập bao gồm quá nhiều người dùng cá nhân và việc lấy tất cả những ý kiến của họ là việc bất khả thi, Product Owner là người đại diện cho tập người dùng, giúp nhóm phát triển làm rõ những yêu cầu về chức năng dựa trên việc cộng tác và lấy phản hồi từ những bên liên quan, giúp sản phẩm có tính thực tế cao trong thời điểm hiện tại.

Các số liệu thống kê từ ngành công nghiệp phần mềm cho thấy, tỉ lệ thành công được tăng lên gấp đôi với sự tham gia của khách hàng xuyên suốt quá trình phát triển sản phẩm. Ý thức được tầm quan trọng của việc gắn kết khách hàng vào quá trình phát triển, những phương pháp Agile đã nhanh chóng đưa ra việc *cộng tác với khách hàng* là một phần không thể thiếu của tuyên ngôn Agile.

Phản hồi với các thay đổi hơn là bám sát kế hoạch

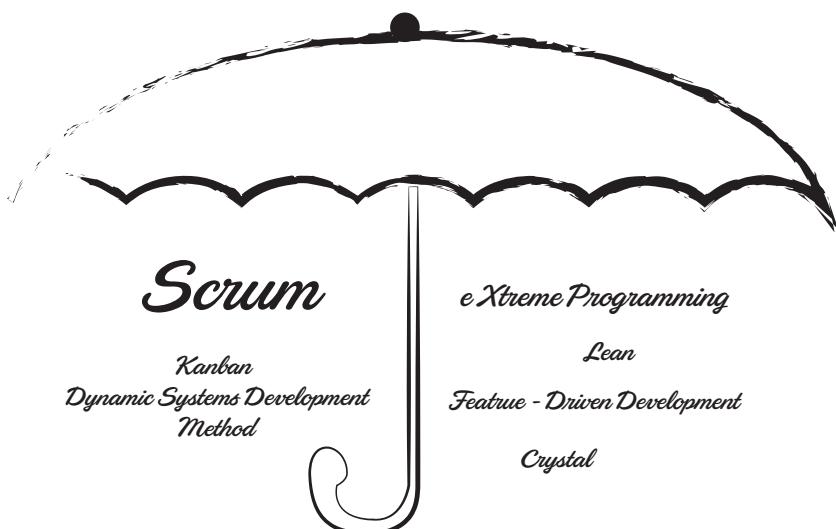
Phản hồi với những thay đổi là chìa khoá vàng để đưa dự án đến thành công. Theo thống kê từ ngành công nghiệp phần mềm, hơn 60% yêu cầu về sản phẩm được thay đổi trong quá trình phát triển. Ngay cả khi dự án phát triển phần mềm theo phương pháp truyền thống kết thúc thành công theo đúng kế hoạch về thời gian, ngân sách và tính năng yêu cầu, khách hàng vẫn thường không hài lòng vì những gì họ nhận được không thực sự là những gì họ cần. Định luật Humphrey nói rằng, *khách hàng không bao giờ biết chính xác những gì họ muốn cho tới khi họ thực sự thao tác với phần mềm*. Nếu khách hàng chỉ nhìn thấy phần mềm vào giai đoạn chuyển giao sau một quá trình dài, thời điểm đó là quá muộn để đưa ra những thay đổi phù hợp trên những phản hồi được gửi tới nhóm phát triển.

Tất cả những phương pháp Agile đều dựa trên việc liên tục điều chỉnh kế hoạch phù hợp với những thay đổi về yêu cầu từ khách hàng. Nhóm phát triển sẽ luôn cung cấp những phần tăng trưởng để mang lại giá trị kinh doanh cao nhất. Với

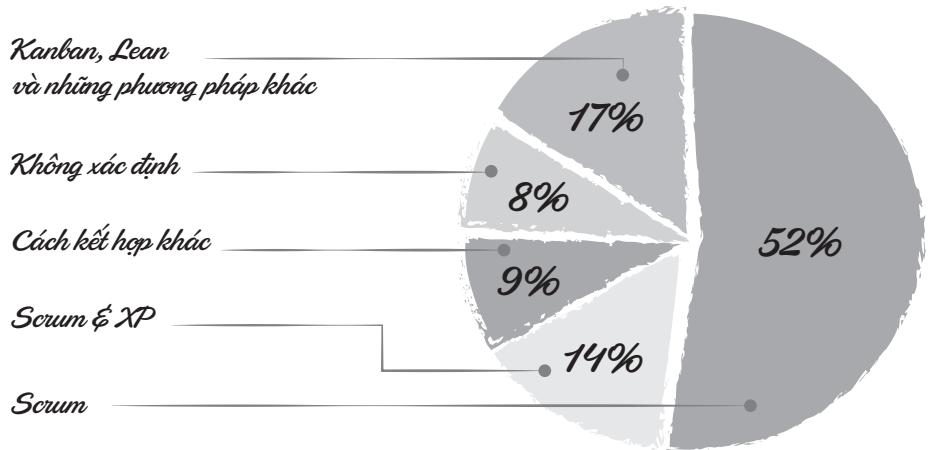
nguyên lý 80/20 được áp dụng triệt để, khách hàng có thể dừng việc phát triển ngay khi phần mềm đã thoả mãn nhu cầu sử dụng; và nhóm phát triển, đương nhiên, cũng cảm thấy hào hứng với những công việc mang lại giá trị và hiệu quả cao. Điều này lý giải tại sao những nhóm phát triển phần mềm theo những phương pháp Agile thường cảm thấy *hạnh phúc* hơn.

Trong chiếc ô Agile

Trong 17 người ký tên vào Tuyên ngôn Agile, mỗi người đều là cha đẻ của một hoặc một vài phương pháp. Tất cả những phương pháp này đều chia sẻ tầm nhìn chung trong tuyên ngôn Agile; về việc cố gắng cộng tác với khách hàng, trong nhóm phát triển, nhằm liên tục tích hợp và chuyển giao những phần tăng trưởng phù hợp với thay đổi yêu cầu. Dù chia sẻ chung tầm nhìn và phương pháp luận, mỗi phương pháp có cách tiếp cận khác nhau và phù hợp với những hoàn cảnh khác nhau. Chương này không có tham vọng đi sâu vào tất cả những phương pháp nằm trong *chiếc ô Agile*, tôi chỉ muốn điểm qua một số phương pháp được sử dụng phổ biến trên thế giới cũng như đặc trưng của từng phương pháp, từ đó bạn có thể tìm hiểu thêm để có phương pháp phù hợp với nhóm của mình.



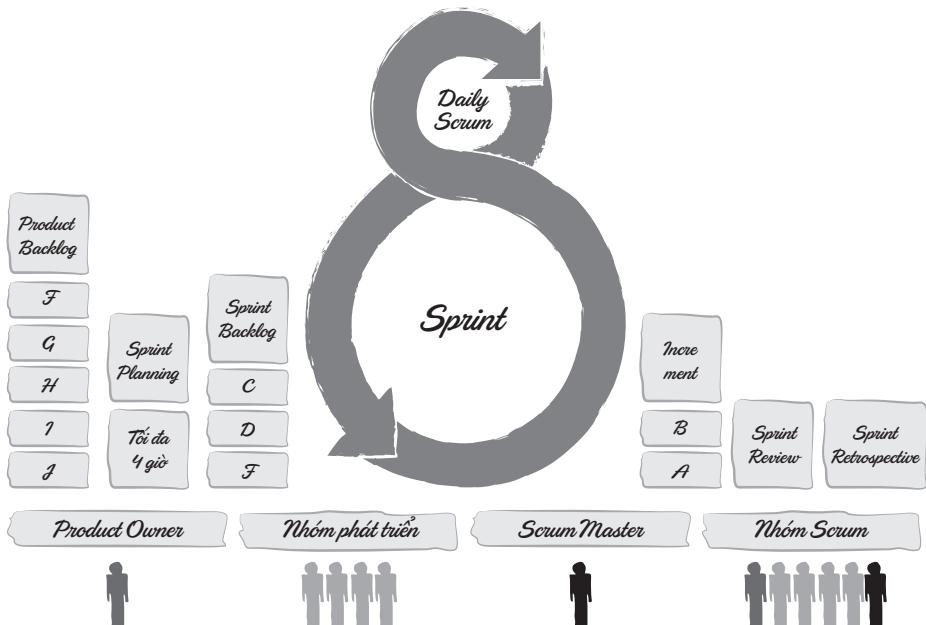
Hình 2.4: Chiếc ô Agile



Hình 2.5: Tỉ lệ thực hành những phương pháp Agile

Scrum: Tập trung, hợp tác

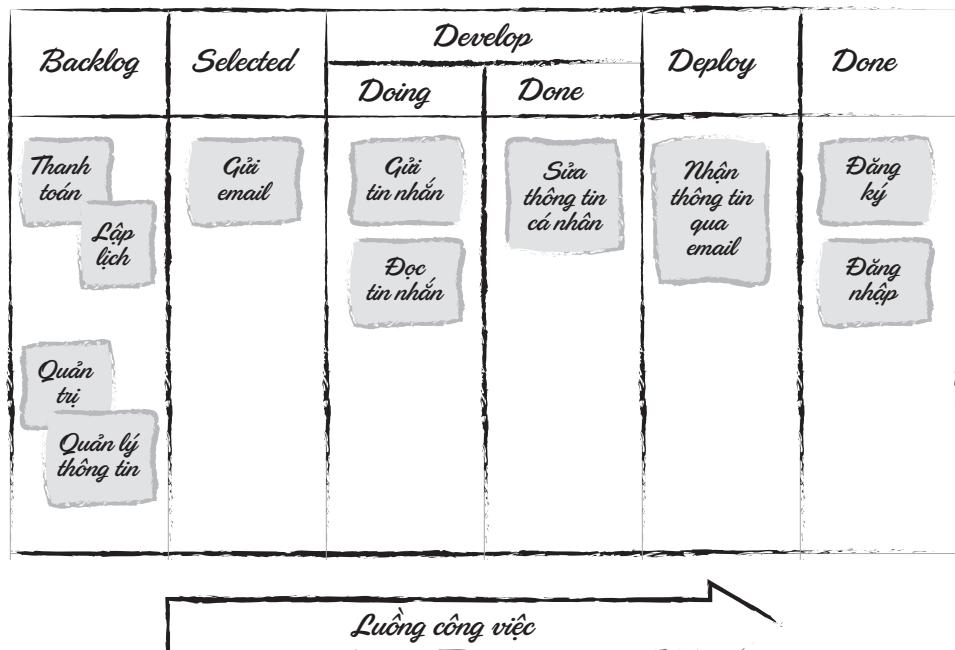
Scrum là phương pháp được sử dụng nhiều nhất, và áp đảo, trong những phương pháp nằm trong *chiếc ô Agile*; một phần vì Scrum đơn giản nhưng đủ chi tiết và đặc biệt phù hợp với việc phát triển sản phẩm. Mặc dù mức độ phổ biến của Scrum hiện đang có xu hướng giảm dần bởi nhiều nhóm phát triển cho rằng Scrum không phải là phương pháp hiệu quả nhất với quá nhiều sự kiện; song chúng ta đều phải thừa nhận rằng, Scrum rất hiệu quả, đặc biệt với những nhóm mới thực hành Agile, bởi tính đầy đủ và toàn vẹn của hướng dẫn thực hành. Scrum vận hành dựa trên những *khung thời gian cố định, lặp đi lặp lại, gọi là Sprint* giúp nhóm phát triển luôn có được *nhiều đập đều đặn và sự tập trung cao độ trong giai đoạn ngắn*.



Hình 2.6: Scrum

Kanban: Tối ưu hiệu quả

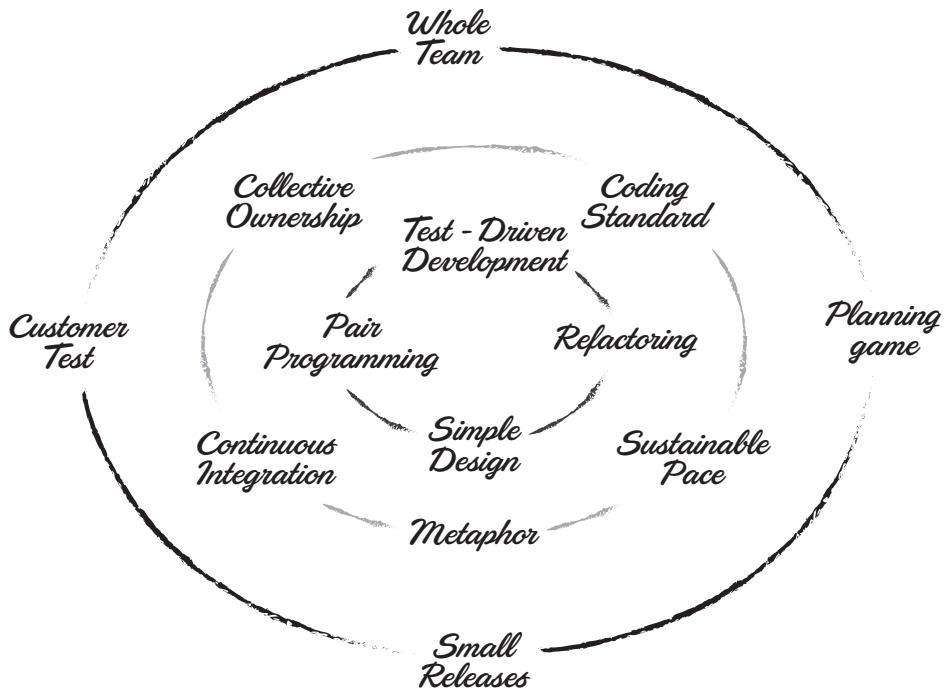
Kanban tập trung vào luồng công việc (workflow) thay vì thời gian. Mỗi công việc được trực quan hóa trên một bảng trạng thái tương ứng với luồng thực hiện công việc, đi từ TO-DO tới DONE. Để tối ưu sự tập trung, Kanban sử dụng khái niệm giới hạn công việc đang thực hiện (WIP: Work-In-Progress). Nhờ tính trực quan và tập trung vào luồng công việc, Kanban giúp nhóm phát triển nhanh chóng nhận ra những nơi có thể cải tiến để có một luồng công việc (quy trình) hiệu quả hơn.



Hình 2.7: Kanban

XP (Extreme Programming): Kỹ thuật và chất lượng

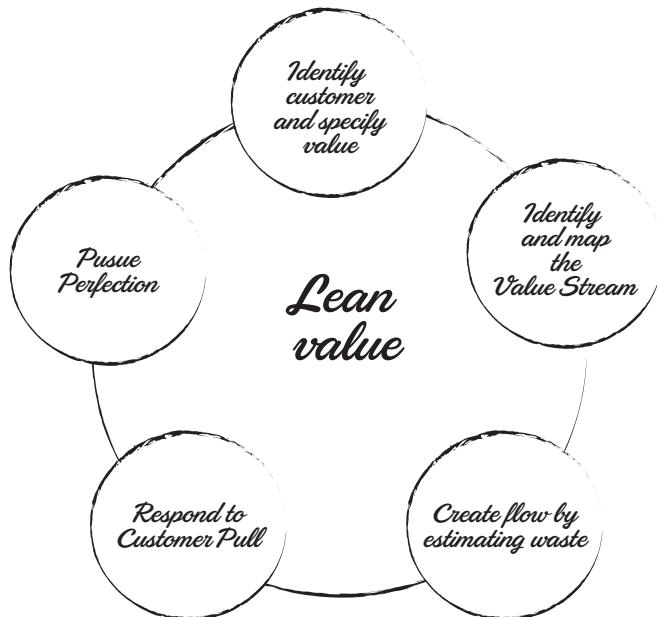
XP là một phương pháp đặc biệt, bởi dù nhóm phát triển phần mềm sử dụng phương pháp nào trong *chiếc ô Agile*, nhóm đều thực hành XP hoặc một số kỹ thuật XP. Tư tưởng của XP là phân chia các tác vụ thành những công việc cụ thể lặp lại trong thời gian ngắn và tập trung vào kỹ thuật để đưa ra sản phẩm với chất lượng cao bằng những thực hành như *lập trình cặp*, *TDD (Test Driven Development)*, *code refactoring*...



Hình 2.8: Những kỹ thuật trong XP

Lean Software Development: Tinh gọn và cổ vũ học tập

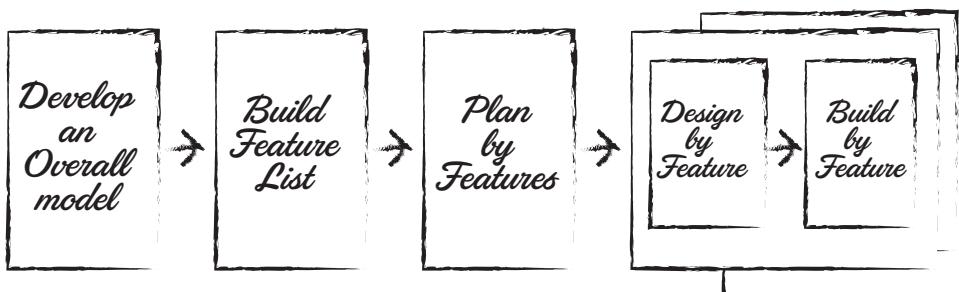
Lean Software Development được Mary và Tom Poppendieck cụ thể hóa tư tưởng của Lean vào việc phát triển phần mềm: *loại bỏ tất cả những công việc không cần thiết* nhằm thúc đẩy sự *chuyển giao nhanh* và *tiết kiệm* hơn. Nếu có bất cứ trở ngại nào được tìm thấy, những thành viên trong nhóm phát triển phần mềm được trao quyền để giải quyết một cách nhanh nhất có thể. Những tư tưởng của Lean rất đáng lưu tâm trong bất cứ phương pháp nào: *loại bỏ lãng phí, built-in quality, trì hoãn cam kết, tối ưu hệ thống, khuyến khích việc học tập...*



Hình 2.9: Những giá trị của Lean Software Development

FDD (Feature-Driven Development): Đa nhiệm

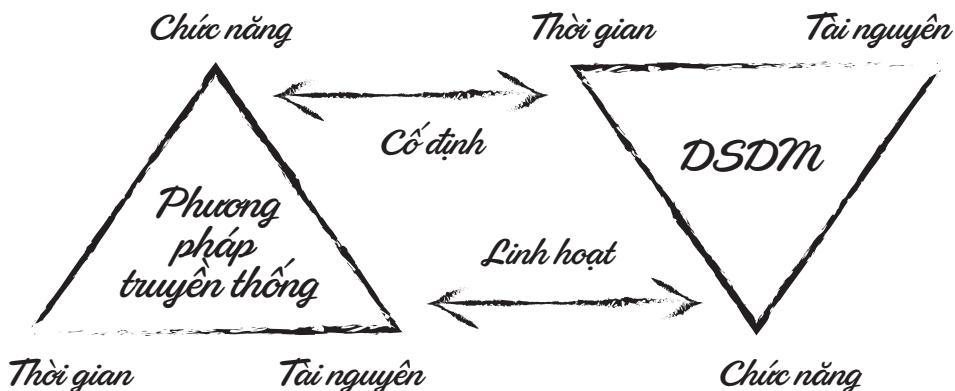
Dự án phần mềm bắt đầu với một khoảng thời gian ngắn nhằm thiết lập những công việc sẽ xảy ra với những mô hình cụ thể cùng danh sách những chức năng và kế hoạch phát triển được trao quyền. Sau đó, mỗi Iteration thực hiện một vài chức năng đảm bảo đủ cho việc chuyển giao với đầy đủ chức năng và tài liệu liên quan. FDD nói chung khá phức tạp và có thể thực hiện **đa nhiệm** trên danh sách **chức năng đã được lên kế hoạch**; nên FDD thường không phù hợp với những dự án nhỏ và vừa.



Hình 2.10: FDD

DSDM (Dynamic Systems Development Method): Thời hạn và ngân sách cố định

DSDM tập trung vào *thời hạn và ngân sách cố định* nhằm tối ưu giá trị. Dựa trên MoSCoW (Must, Should, Could, Won't Have) nhằm đưa ra danh sách ưu tiên cho mỗi chức năng, nhóm phát triển cố gắng thực hiện từng chức năng trong *phạm vi thời gian và ngân sách định trước*. Do đó, DSDM phù hợp với những dự án có nguồn lực hạn chế.



Hình 2.10: DSDM

Trên đây chỉ là một số phương pháp đặc trưng trong *chiếc ô Agile*, vẫn còn hàng chục những phương pháp khác; và tất nhiên, mỗi phương pháp phù hợp với những hoàn cảnh cụ thể khác nhau. Nếu nhóm của bạn mới thực hành Agile, tôi vẫn khuyến nghị bạn thực hành Scrum. Dù rằng Scrum không phải là phương pháp hiệu quả nhất, Scrum vẫn rất hiệu quả, và sau khi làm chủ Scrum, bạn sẽ dễ dàng hơn trong việc tiếp cận và thực hành những phương pháp khác.

HIỂU ĐÚNG

Agile là Scrum; Scrum là Agile.

Đây là hiểu nhầm rất phổ biến, đặc biệt với những người mới tìm hiểu về Agile bởi bạn có thể thấy trong rất nhiều bài báo (cả Agile Vietnam và Hanoi Scrum) viết là: Agile\Scrum. Hãy xem lại ghi chú về *chiếc ô Agile* để biết rằng Agile là phương pháp luận, và Scrum là một phương pháp cụ thể. Nhưng tác giả những bài báo viết Agile\Scrum cũng không sai, thường là do ý đồ cụ thể để những độc giả mới làm quen với Agile thấy “quen thuộc” hơn; nhất là với hơn 50% thị phần sử dụng, Scrum thường là *con đường bước vào thế giới Agile* của những nhóm mới thực hành Agile.

Agile không coi trọng tài liệu.

Đây là hiểu nhầm cũng rất phổ biến, có thể gặp tại bất cứ buổi thảo luận nào về Agile tại Việt Nam với tần suất 1 câu hỏi / 1 buổi. Tuyên ngôn Agile và tất cả những phương pháp trong *chiếc ô Agile* chưa bao giờ đề cập đến vấn đề tài liệu. Nơi duy nhất bạn tìm thấy chính là câu thứ 2 trong tuyên ngôn Agile *phần mềm chạy tốt hơn là tài liệu đầy đủ*. Phương pháp Agile nhấn mạnh việc chuyển giao *phần mềm chạy tốt* có độ ưu tiên cao hơn so với tài liệu; bởi tài liệu đầy đủ chưa bao giờ là vấn đề quyết định phần mềm chạy tốt. Trong phương pháp truyền thống, rất nhiều công đoạn *hướng đến đầu ra là tài liệu*: đặc tả yêu cầu, thiết kế, thiết kế chi tiết... điều này khiến những nhân viên thực hiện công đoạn này *xao nhãng mục tiêu chung là tạo ra phần mềm chạy tốt* bởi tài liệu là thước đo đầu ra của quá trình công việc của họ. Tuyên ngôn Agile muốn loại bỏ sự xao nhãng này nhưng *không phủ định* tài liệu. Hãy tạo ra bất cứ tài liệu nào bạn cảm thấy cần thiết, song đừng coi đó là mục tiêu, một tần tài liệu không làm nên phần mềm chạy tốt.

Agile không coi trọng quy trình, công cụ.

Tương tự như trên, hãy để ý đến từ hơn là trong mỗi phát biểu của Tuyên ngôn Agile và phần cuối của bản Tuyên ngôn (nơi rất ít người chú ý): *Mặc dù các điều bên phải vẫn còn giá trị, nhưng chúng tôi đánh giá cao hơn các mục ở bên trái.*

TỔNG KẾT

Agile là phương pháp phát triển phần mềm hiện đại, nhằm giải quyết tận gốc những vấn đề tồn tại của phương pháp phát triển phần mềm truyền thống. Agile tập trung vào *cá nhân và sự tương tác* hơn là quy trình và công cụ; phần mềm chạy tốt hơn là tài liệu đầy đủ; cộng tác với khách hàng hơn là đàm phán hợp đồng; phản hồi với các thay đổi hơn là bám sát kế hoạch. Những vấn đề bên phải vẫn có giá trị nhưng Agile tập trung vào những điều bên trái hơn. Có hàng chục phương pháp nằm trong *chiếc ô Agile* đều chia sẻ tầm nhìn chung trong Tuyên ngôn Agile với 12 nguyên lý.

Scrum là phương pháp được sử dụng nhiều nhất, và thường là điểm bắt đầu của những nhóm mới thực hành Agile bởi sự đơn giản nhưng hoàn thiện và dễ hiểu.

Kanban là phương pháp có sự tăng trưởng nhanh nhất hiện nay.

eXtreme Programming (XP) chứa đựng những practice được sử dụng nhiều nhất trong những nhóm thực hành Agile.

DR

SCRUM

Scrum là phương pháp nổi tiếng và phổ biến nhất trong tập hợp những phương pháp phát triển phần mềm theo tư tưởng Agile. Điều khiến Scrum trở nên phổ biến là phương pháp này được đóng gói rất ngắn gọn nhưng lại tương đối đầy đủ để thực hành với những vai trò và công việc xác định. Scrum thường là phương pháp phát triển phần mềm đầu tiên được các doanh nghiệp lựa chọn khi quyết định chuyển đổi theo Agile. Một lý do khác là, vì sự phổ biến của mình, Scrum khiến các công ty lần đầu áp dụng Agile tin rằng đây là phương pháp tốt nhất và đảm bảo cho sự thành công. Do vậy, đã có một thời gian người ta luôn đánh tráo hoặc đồng nhất Agile và Scrum, đặc biệt ở những cộng đồng còn hiểu biết thấp về Agile như Việt Nam. Ngay cả Hanoi Scrum, lần đầu nghĩ đến việc thành lập một cộng đồng thực hành Agile tại Việt Nam vào năm 2009, cũng bắt đầu với việc tìm hiểu và phát triển theo phương pháp Scrum.

Vào năm 1986, Takeuchi và Nonaka là hai người đầu tiên đưa ra tên Scrum nhằm chỉ tới một phương pháp phát triển có hiệu suất cao trong một nhóm làm việc liên chức năng, được đăng tải trên tạp chí Harvard Business Review. Thời điểm đó, ý niệm về một phương pháp đột phá mới chỉ dừng lại ở phạm vi một nghiên cứu và báo cáo. Năm 1993, Jeff Sutherland đã cụ thể hoá thành một quy trình với những chỉ dẫn đầy đủ và mượn lại tên Scrum đã được sử dụng trước đó của Takeuchi và Nonaka.

Tất nhiên, Scrum hoàn toàn tuân theo những giá trị của *tuyên ngôn Agile*:

Các cá nhân và sự tương tác hơn là quy trình và công cụ

Scrum là một phương pháp phát triển theo nhóm nhằm cung cấp giá trị cho việc phát triển phần mềm. Các thành viên trong nhóm làm việc cùng nhau với quy trình và các công cụ do chính nhóm đồng thuận thiết lập nhằm đạt được mục đích chung và tạo ra giá trị cao nhất.

Scrum khuyến khích các thành viên trong nhóm tương tác cùng nhau với một mục tiêu xác định, nhằm:

- xác định được các công việc cần thực hiện để hoàn thành mục tiêu;
- tìm ra cách tốt nhất để hoàn thành một công việc;
- thực hiện công việc;
- nhận diện những vấn đề tồn tại;
- có trách nhiệm giải quyết tất cả những khó khăn trong phạm vi nhóm;

- làm việc với các bộ phận khác của tổ chức để giải quyết các mối quan tâm bên ngoài vùng kiểm soát của nhóm.

Phần mềm chạy tốt hơn là tài liệu đầy đủ

Scrum cụ thể hoá khái niệm *Iteration* của Agile bằng *Sprint*. Mỗi Sprint là một khoảng thời gian làm việc xác định, đủ dài để phát triển ít nhất một chức năng và đủ ngắn để chuyển giao và nhận được phản hồi của khách hàng sớm nhất có thể. Vì thế mỗi Sprint thường có thời gian từ 1 đến 4 tuần, và được khuyến nghị là 2 tuần.

Kết quả của mỗi Sprint sẽ là một hoặc nhiều chức năng có thể sử dụng được thông qua thao tác của người dùng, thường được gọi là *phản tăng trưởng (incremental)*, có *thể chuyển giao (shippable, deliverable, releasable)*. Tất nhiên, phản tăng trưởng này có thể chưa hình thành được sản phẩm với đầy đủ những nghiệp vụ cần thiết, nhưng chức năng đó phải hoàn chỉnh với giới hạn đã đặt ra, và đặc biệt là có thể được thao tác bởi người dùng – đây là điều rất quan trọng giúp người dùng có phản hồi nhanh chóng.

Cộng tác với khách hàng hơn là đàm phán hợp đồng

Scrum được thiết kế để thúc đẩy sự cộng tác giữa các thành viên trong nhóm với nhau để tìm ra cách tốt nhất nhằm hoàn thiện sản phẩm. Scrum chỉ định một vai trò đại diện cho khách hàng, gọi là *Product Owner (PO – chủ sản phẩm)*, với nhiệm vụ phối hợp với các bên liên quan để kiểm tra, điều chỉnh tầm nhìn của sản phẩm nhằm đưa sản phẩm có giá trị nhất trong các ràng buộc hiện tại.

Phản hồi với các thay đổi hơn là bám sát kế hoạch

Nhóm Scrum thường xuyên cập nhật kế hoạch. Mỗi Sprint đều có *một mục tiêu cụ thể*, gọi là *Sprint Goal*, và kế hoạch để đạt được mục tiêu vào cuối Sprint. Nhiều nhóm Scrum cũng có những kế hoạch dài hạn cho cả dự án cũng như lộ trình release sản phẩm theo từng giai đoạn; nhưng thông thường, khoảng thời gian cho cả dự án là quá dài với thông tin ban đầu không rõ ràng nên đó thường là những kế hoạch sơ bộ. Những kế hoạch cụ thể chỉ được tạo ra và cập nhật theo *từng Sprint*, và *từng ngày* để đảm bảo kế hoạch là đúng đắn và khả thi với những điều kiện hiện tại. Tuy vậy, mục tiêu của nhóm không phải là bám theo kế hoạch đã có. Về bản chất, việc suy nghĩ để đưa ra những ý tưởng nhằm cập nhật kế hoạch và thực hiện đúng mục tiêu để ra quan trọng hơn nhiều so với kế hoạch.

Trong phần lớn trường hợp, một kế hoạch dài hạn được xây dựng dựa trên lượng

thông tin hạn chế sẽ không hiệu quả, bởi đó thường không phải là phương án tốt nhất. Những kế hoạch ngắn hạn với những thông tin rõ ràng tại thời điểm hiện tại và giả định ít bị thay đổi trong tương lai ngắn có nhiều giá trị hơn, và tạo ra cơ hội cải thiện sự thành công của dự án vì những kiến thức mới có cơ hội được tích hợp nhanh vào những kinh nghiệm đã có.

Nhóm Scrum liên tục phản hồi với những thay đổi để cho kết quả tốt nhất. Scrum tạo ra một vòng phản hồi thông qua Sprint hay cuộc họp hàng ngày, cho phép các nhóm liên tục kiểm tra và thích nghi nhằm mang lại sản phẩm có giá trị tối đa.

SCRUM LÀ GÌ?

Scrum là một *Agile framework* trong đó nhóm có thể giải quyết vấn đề thích nghi phức tạp, nhằm cung cấp giá trị cao nhất về kinh doanh và sáng tạo sản phẩm.

Scrum thường được mô tả là:

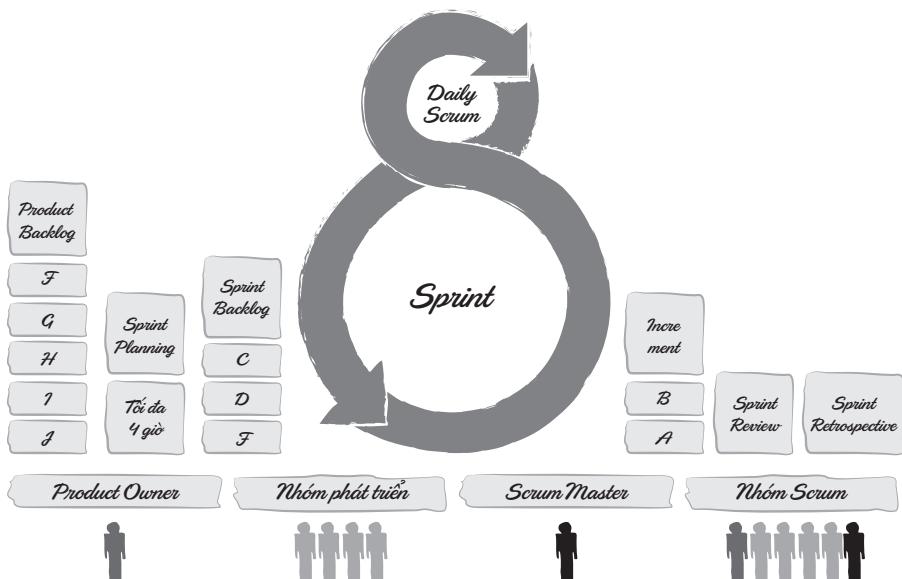
- *Lightweight framework*: Một khung làm việc gọn nhẹ, cung cấp cho các nhóm phát triển phần mềm cách tổ chức và làm việc cơ bản thông qua những *vai trò, sự kiện, artifact* (*tạo tác*) xác định; tuy nhiên, Scrum khuyến khích các nhóm tự tạo ra quy trình và cách thực hiện phù hợp cho riêng mình.
- *Đơn giản để hiểu*: Thông thường, Scrum có thể được mô tả ngắn gọn dưới 30 phút và dưới 1 trang A4.
- *Khó khăn để làm chủ*: Tuy vậy, chính vì Scrum đơn giản và khuyến khích các nhóm tự tạo ra quy trình của riêng mình, rất nhiều nhóm thực hành Scrum hiểu sai và không làm chủ được Scrum, khiến việc thực hiện Scrum BUT xảy ra phổ biến.

Scrum không phải là một quy trình hay kỹ thuật để phát triển sản phẩm phần mềm, Scrum là một management framework cho phép chúng ta sử dụng những quy trình hay kỹ thuật khác nhau nhằm đạt hiệu quả cao nhất về sự thích ứng và sáng tạo.

Scrum framework bao gồm *Scrum Team, các vai trò và nhiệm vụ liên quan, các sự kiện, artifact, và các quy tắc*. Mỗi thành phần trong Scrum framework phục vụ một mục đích cụ thể và cần thiết để thực hành Scrum hiệu quả.

Các quy tắc của Scrum ràng buộc với nhau bao gồm: *các sự kiện, vai trò, tạo tác, và mối quan hệ, tương tác giữa chúng*.

Cách làm việc theo Scrum có thể được minh họa như hình dưới đây:



Hình 3.1: Tổng quan về Scrum

Nhóm phát triển phần mềm bắt đầu với một danh sách những chức năng cần có trong sản phẩm, gọi là *Product Backlog*, với cơ cấu nhân sự gồm 3 thành phần: *Product Owner*, *ScrumMaster*, *nhóm phát triển*. *Product Owner* tạo ra *Product Backlog* chứa các yêu cầu của dự án với những hạng mục được sắp xếp theo độ ưu tiên. *Nhóm Scrum* sẽ thực hiện việc phát triển sản phẩm thông qua nhiều *Sprint*. Mỗi *Sprint* kéo dài từ 1 đến 4 tuần và có *Sprint Goal* cụ thể, hoàn thành ít nhất 1 chức năng có thể chuyển giao cho người dùng (phần tăng trưởng, shippable...). Tại mỗi *Sprint*, *nhóm Scrum* sẽ làm việc như sau:

- Bắt đầu *Sprint* với một cuộc họp để ước lượng công việc, lập kế hoạch và xác định *Sprint Goal*. Buổi họp này được gọi là *Sprint Planning*.
- Các ngày trong *Sprint*:
 - Nhóm phát triển sẽ làm việc cùng *Product Owner* để đảm bảo yêu cầu được hiểu đúng.
 - Thực hiện *Daily Scrum*, cuộc họp có thời lượng dưới 15 phút để đồng bộ các công việc của nhóm.
 - *ScrumMaster* đảm bảo loại bỏ những trở ngại, trợ giúp cho nhóm phát triển làm tốt nhất công việc của mình.

- Kết thúc Sprint:

- Nhóm thực hiện *Sprint Review* để review sản phẩm.
- Thực hiện *Sprint Retrospective* để cải tiến cách thức thực hiện của nhóm.

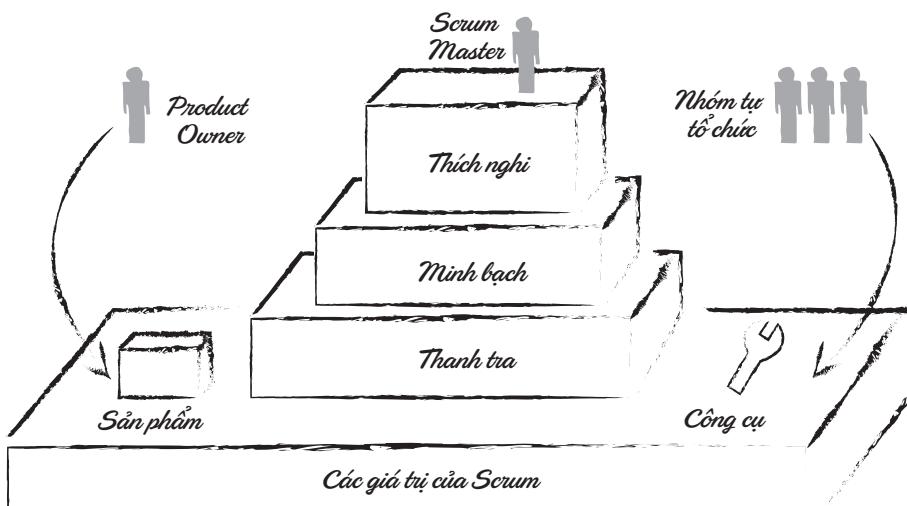
Cách làm việc theo Sprint sẽ được lặp đi lặp lại cho tới khi các hạng mục trong Product Backlog đều được hoàn tất hoặc khi Product Owner quyết định dừng dự án căn cứ theo tình hình thực tế.

Scrum sử dụng chiến thuật *Có giá trị hơn làm trước* nên các hạng mục có nhiều giá trị luôn được hoàn thành trước. Do đó Scrum luôn mang lại giá trị cao nhất về mặt kinh doanh, nghiệp vụ.

Scrum tập trung vào việc chuyển giao chức năng có giá trị hoàn chỉnh nên sản phẩm sẽ được hình thành dần nhưng vẫn có thể đưa vào sử dụng tại bất cứ thời điểm nào. Do đó Scrum thường tối ưu quá trình phát triển – phản hồi.

Scrum sử dụng việc cải tiến quy trình sau mỗi Sprint nên nhóm Scrum sẽ dần tìm ra cách làm việc phù hợp của riêng mình. Do đó Scrum thường cho năng suất lao động cao.

Một cách làm dễ hiểu? Đúng. Bởi vậy một tính chất của Scrum là *rất dễ hiểu*. Tuy vậy, để *thực hành thành thực Scrum lại không hề đơn giản*. Chúng ta hãy phân tích cụ thể Scrum.



Hình 3.2: Scrum

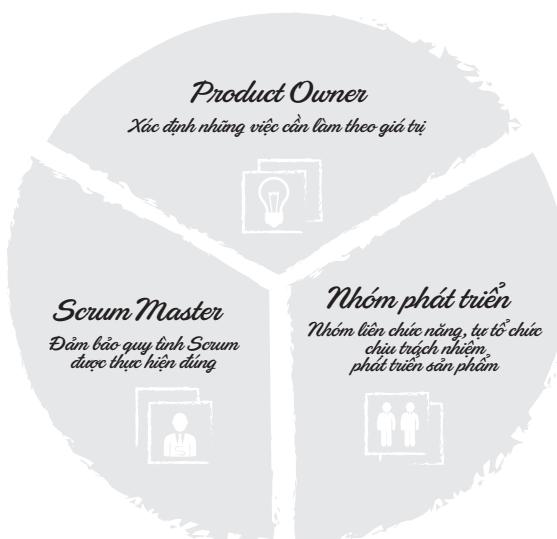
CÁC VAI TRÒ TRONG SCRUM

Các vai trò trong nhóm Scrum được thiết kế để tối ưu hoá sự linh hoạt, sáng tạo và năng suất, bao gồm ba vai trò: *Product Owner*, *ScrumMaster*, *Nhóm phát triển* – đây là *nhóm tự tổ chức và liên chức năng*.

Nhóm tự tổ chức có nghĩa là nhóm tự chọn quy trình, cách thức, kỹ thuật tốt nhất để thực hiện công việc nhằm đạt được mục tiêu đề ra, không phụ thuộc hay chịu bất cứ ảnh hưởng nào của một người ngoài nhóm.

Nhóm liên chức năng có nghĩa là nhóm có đầy đủ những kiến thức và kỹ năng cần thiết để thực hiện công việc và đạt được mục tiêu đề ra.

Hai yếu tố này khiến nhóm Scrum khác nhiều với những nhóm phát triển phần mềm truyền thống bởi họ không phải nhận những *mệnh lệnh* cụ thể nhưng đôi khi không phù hợp đến từ những người quản lý, cũng như không bị chia tách bởi việc phân nhóm theo chức năng. Với cách làm thông thường, Project Manager quản lý việc phát triển sản phẩm dựa trên những nhóm có chức năng riêng biệt như nhóm requirement engineer, nhóm developer, nhóm tester... Điều này dẫn đến hiệu quả kém trong *tương tác để hoàn thiện một chức năng* (bởi sự tương tác nhóm-nhóm, không phải cá nhân-cá nhân); đồng thời Project Manager cũng có thể đưa ra nhiều quyết định không sáng suốt vì rất khó tìm ra một cách làm việc hiệu quả giữa các nhóm có chức năng, ngữ cảnh và văn hoá khác biệt.



Hình 3.3: Các vai trò trong Scrum

Product Owner

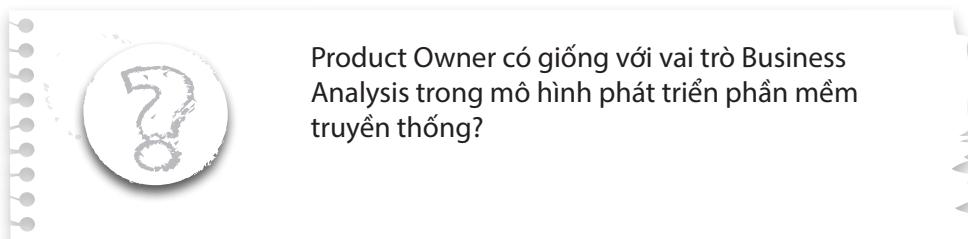
Product Owner là người chịu trách nhiệm tối đa hóa giá trị của sản phẩm và công việc của Nhóm phát triển. Product Owner là người duy nhất chịu trách nhiệm về việc quản lý sản phẩm, quản lý Product Backlog bao gồm:

- Đảm bảo tất cả hạng mục trong Product Backlog được thể hiện rõ ràng và không gây hiểu nhầm;
- Sắp xếp các hạng mục trong Product Backlog nhằm đạt được mục tiêu và nhiệm vụ theo cách tốt nhất;
- Tối ưu hóa giá trị kinh doanh, nghiệp vụ từ công việc của nhóm phát triển;
- Đảm bảo rằng Product Backlog được trình bày một cách trực quan, minh bạch và rõ ràng cho tất cả thành viên, và cho thấy những gì nhóm Scrum sẽ làm việc trong các Sprint tiếp theo;
- Bảo đảm Nhóm phát triển hiểu được mục trong Product Backlog ở mức độ cần thiết.

Công việc trên có thể được thực hiện bởi Product Owner hoặc nhóm phát triển, song Product Owner là người duy nhất chịu trách nhiệm.

Cần lưu ý là, Product Owner là một người, không phải là một nhóm. Product Owner có thể đại diện cho nguyện vọng của một nhóm trong Product Backlog, nhưng những người muốn thay đổi mức độ ưu tiên của một hạng mục trong Product Backlog phải thông qua Product Owner. Điều này giúp thông tin được đưa tới Nhóm phát triển là duy nhất, nhất quán và không gây hiểu nhầm.

Để Product Owner hoàn thành tốt vai trò của mình, toàn bộ tổ chức phải tôn trọng quyết định của họ. Các quyết định về sản phẩm của Product Owner nằm ở việc sắp xếp những hạng mục và quản lý Product Backlog; và Product Owner là người duy nhất được phép yêu cầu nhóm phát triển phải làm những chức năng gì, với yêu cầu cụ thể ra sao.



Nhóm phát triển

Nhóm phát triển bao gồm các chuyên gia, những người có đầy đủ năng lực chuyên môn cùng cộng tác với nhau để đưa ra một phần tăng trưởng vào cuối mỗi Sprint. Chỉ có các thành viên của nhóm phát triển tạo ra các phần tăng trưởng.

Nhóm phát triển có các đặc điểm sau:

- *Là nhóm tự tổ chức.* Không ai được phép yêu cầu nhóm phát triển thực hiện theo cách thức nào để biến những hạng mục trong Product Backlog thành phần tăng trưởng có thể chuyển giao được;
- *Là nhóm liên chức năng,* với tất cả các kỹ năng cần thiết để hoàn thành những công việc nhằm đạt được Sprint Goal;
- Những thành viên trong nhóm phát triển *không có những chức danh khác nhau*, bất kể công việc họ đang thực hiện là gì; không có trường hợp ngoại lệ cho quy tắc này;
- *Nhóm phát triển là đơn vị nhỏ nhất.* Có nghĩa là trong nhóm phát triển không thể bao gồm những nhóm nhỏ hơn; không có trường hợp ngoại lệ cho quy tắc này;
- Các thành viên trong nhóm có những kiến thức, kỹ năng chuyên ngành khác nhau nhưng đều có *trách nhiệm để hoàn thành mục tiêu đề ra*, hoàn toàn thuộc về nhóm như một đơn vị nhỏ nhất.

Tôi đặc biệt nhấn mạnh 3 đặc điểm cuối cùng, bởi chúng rất quan trọng và tạo ra sự khác biệt của nhóm Scrum.

Việc không có nhiều chức danh khác nhau trong nhóm cũng như không chia nhỏ nhóm hơn nữa khiến các thành viên trong nhóm cộng tác với nhau hiệu quả hơn và đồng thuận trong trách nhiệm hoàn thành mục tiêu. Điều này không thực sự dễ hiểu, nhưng nếu trong nhóm có chức danh *technical leader* đồng nghĩa với những người khác không có trách nhiệm trong việc lựa chọn công nghệ hay kỹ thuật phù hợp cho mục tiêu hay kỹ năng của họ. Tương tự, nếu chúng ta tiếp tục phân tách Nhóm phát triển thành *nhóm lập trình, nhóm kiểm thử...* sẽ khiến mọi thành viên trong nhóm cố gắng hoàn thành mục tiêu của nhóm nhỏ thay vì mục tiêu chung của cả nhóm phát triển phần mềm. Các nghiên cứu để tăng năng suất và tạo động lực làm việc cho cá nhân đều tạo ra một cách hữu hiệu là *cho họ một chức danh*: một nhân viên sẽ cố gắng làm việc tốt để được đề bạt lên một vị trí cao hơn như *team lead*; hoặc khi nhân viên đã cố gắng làm việc tốt, tổ chức có xu hướng “tặng” cho

họ một chức danh cao hơn nhằm cho họ cảm giác quan trọng hơn. Nhưng chúng ta cần tỉnh táo, việc tạo ra một hệ thống phân cấp rất dễ tạo ra động lực cho nhân viên hoàn thành công việc với mục tiêu cá nhân: đưa họ lên vị trí mới. Điều này không có gì là sai trái, nhưng thông thường, bởi quá tập trung vào mục tiêu cá nhân, nhân viên thường không hướng tới mục tiêu chung và bỏ quên việc trợ giúp những thành viên khác của nhóm, thậm chí là cạnh tranh bằng cách kìm hãm năng lực của những thành viên khác. Ví dụ, thay vì hướng tới mục tiêu làm ra sản phẩm với chất lượng tốt nhất, những developer bỏ mặc việc xây dựng kiến trúc cho technical leader, và họ cố gắng làm theo kiến trúc đã được quyết định, dù rằng họ không có kiến thức hay khả năng gì về kiến trúc đó. Tương tự, thay vì trợ giúp những lập trình viên hạn chế bug, những thành viên với chức danh tester sẽ dành thời gian để tìm ra thật nhiều bug, bởi một tester giỏi chắc chắn phải tìm ra được nhiều bug.

Scrum không quy trách nhiệm theo cá nhân, cả nhóm phải chịu trách nhiệm cho kết quả đầu ra. Điều này nghe có vẻ hơi hướng Xã hội Chủ nghĩa và điều nghi ngờ về việc *cha chung không ai khóc* hoàn toàn có lý. Tuy nhiên, tôi tin rằng một Nhóm Xã hội Chủ nghĩa hoàn toàn có khả năng thành công. Đặc biệt khi hội tụ được 5 giá trị Scrum sẽ được đề cập ở dưới, nhóm Scrum sẽ cho hiệu suất rất cao.

Quy mô của nhóm Scrum thường phải đủ lớn để nhóm có đủ những kỹ năng cần thiết để trở thành một nhóm liên chức năng, nhưng phải đủ nhỏ để việc cộng tác và minh bạch thông tin trở nên dễ dàng. Vì vậy, Nhóm phát triển thường được khuyến nghị có số lượng từ 7 đến 9 thành viên. Những dự án cần nhiều nguồn lực, như hàng chục hay hàng trăm nhân viên, cần được chia nhỏ thành những nhóm liên chức năng và thực hiện *Scrum of Scrums*; chúng ta sẽ thảo luận về vấn đề này trong các chương sau. Tức là, thay vì phân chia 100 người tham gia dự án thành 3 bộ phận: phân tích nghiệp vụ, 10 người; lập trình, 60 người; kiểm thử, 30 người; chúng ta sẽ chia thành 10 nhóm phát triển thực hiện 10 module, mỗi nhóm có 1 thành viên làm công việc phân tích nghiệp vụ, 6 lập trình viên và 3 thành viên làm nhiệm vụ kiểm thử. Việc phân chia trong thực tế có thể không như vậy, tuỳ thuộc vào ngữ cảnh của dự án, nhưng các nhóm phải là liên chức năng.



Nhóm phát triển không có chức danh quản lý (manager) và lãnh đạo (leader), vậy ai sẽ đưa ra và chịu trách nhiệm cho những quyết định của nhóm?

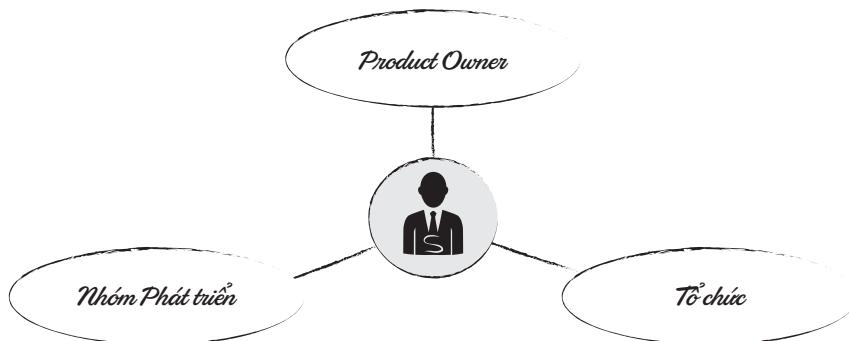
ScrumMaster

ScrumMaster là người chịu trách nhiệm đảm bảo Scrum được hiểu và được thực hành đúng trong nhóm Scrum. ScrumMaster thực hiện việc này bằng cách đảm bảo rằng nhóm Scrum luôn tôn trọng lý thuyết Scrum và quy tắc đã được thống nhất trong nhóm.

ScrumMaster thường được gọi với một cái tên mỹ miều là *lãnh đạo-đầy tớ (servant-leader)* đúng như hai công việc chính họ cần thực hiện:

- Dẫn dắt nhóm Scrum theo đúng định hướng hoạt động của Scrum, đảm bảo nhóm Scrum tìm ra cách thức hoạt động tốt nhất của riêng mình;
- Hỗ trợ nhóm Scrum loại bỏ những trở ngại để hoàn thành mục tiêu đề ra theo cách tốt nhất.

Đây là vị trí rất “khó nhằn” bởi ScrumMaster là người có quyền lực nhưng họ cần sử dụng quyền lực của mình một cách mềm dẻo và đúng đắn dựa trên hiểu biết sâu sắc về Scrum cũng như nhạy cảm để nhận biết những vấn đề đang tồn tại và khuyến khích nhóm phát triển tìm ra giải pháp phù hợp nhất của riêng mình. Sự hiệu quả của nhóm Scrum phụ thuộc rất nhiều vào hoạt động của ScrumMaster khi họ thực hiện đúng nghĩa là một *servant* với *tâm nhìn của một leader*.



Hình 3.4: Vai trò ScrumMaster

ScrumMaster định hướng và hỗ trợ Product Owner:

- tìm kiếm các kỹ thuật để quản lý hiệu quả Product Backlog;
- giúp nhóm Scrum hiểu sự cần thiết của việc giữ cho các hạng mục trong Product Backlog rõ ràng và súc tích;
- hiểu kế hoạch phát triển sản phẩm trong môi trường thực nghiệm;

- bảo đảm Product Owner biết cách sắp xếp Product Backlog để tối đa hóa giá trị;
- hiểu và thực hành về sự linh hoạt;
- tổ chức những sự kiện của Scrum khi cần thiết.

ScrumMaster định hướng và hỗ trợ Nhóm phát triển:

- huấn luyện Nhóm phát triển có khả năng tự tổ chức và làm việc liên chức năng;
- giúp nhóm phát triển tạo ra các sản phẩm có giá trị cao;
- loại bỏ rào cản ảnh hưởng tới tiến độ của Nhóm phát triển;
- tổ chức những sự kiện của Scrum khi cần thiết;
- huấn luyện các Nhóm phát triển trong tổ chức trong đó Scrum chưa hoàn toàn được hiểu rõ và chấp nhận.

ScrumMaster định hướng và hỗ trợ tổ chức:

- dẫn dắt và huấn luyện tổ chức trong việc áp dụng Scrum;
- lập kế hoạch triển khai Scrum trong tổ chức;
- giúp nhân viên và các bên liên quan hiểu và thực hành Scrum;
- giúp thay đổi đó làm tăng năng suất của nhóm Scrum;
- làm việc với những ScrumMaster khác để tăng hiệu quả của việc áp dụng Scrum trong tổ chức.

Đây là một vị trí đặc biệt quan trọng trong những tổ chức mới thực hành Agile. Các tổ chức mới thực hành Scrum thường chia sẻ rằng họ hay gặp phải vấn đề với *bước hai*. Tôi thích gọi như vậy, với *bước một* là việc trao quyền tự chủ cho nhóm phát triển cùng Product Owner, nhóm phát triển thường rất hào hứng và cho kết quả cao trong thời gian đầu, song họ có xu hướng quán tính dẫn tới giai đoạn trì trệ - đó chính là *bước hai*, nơi tổ chức cần một ScrumMaster thật tốt để đảm bảo tổ chức vượt qua giai đoạn này, và tiến xa hơn từ bước đà được tạo ra bởi *bước một*, thay vì coi đó là điểm kết thúc và hài lòng với những gì đã có.



Vai trò của ScrumMaster giống với Project Manager trong phương pháp phát triển phần mềm truyền thống? ScrumMaster chịu trách nhiệm cho sự thành công và thất bại của dự án?

CÁC SỰ KIỆN SCRUM

Các sự kiện được quy định trong Scrum được tạo ra nhằm duy trì sự tương tác đều đặn với mục tiêu duy trì 3 trụ cột của Scrum là *sự minh bạch, thanh tra và khuyến nghị* cũng như giảm thiểu những cuộc họp không cần thiết trong các dự án. Lưu ý, tất cả những sự kiện này đều có timebox (giới hạn thời gian), có nghĩa là không sự kiện nào được phép kéo dài hơn khoảng thời gian cho phép.

Sprint

Sprint là sự kiện dài nhất, phức tạp nhất và là *trái tim* của Scrum.

Thành phần tham dự:

- ScrumMaster
- Product Owner
- Nhóm phát triển

Thời gian:

- Từ 1 đến 4 tuần.

Kết quả:

- Ít nhất một chức năng được chuyển giao

Một vòng đời phát triển sản phẩm phần mềm theo phương pháp Scrum bao gồm nhiều Sprint, và chỉ bao gồm Sprint. Một Sprint thường được kéo dài từ 1 đến 4 tuần và không được phép dài hơn, thường được khuyến nghị là 2 tuần. Sprint được bắt đầu ngay khi dự án bắt đầu hoặc ngay khi Sprint trước đó kết thúc.

Một Sprint thường phải *đủ ngắn để phần tăng trưởng nhận được phản hồi nhanh nhất có thể*, nhưng phải đủ dài để hoàn thành được ít nhất một phần tăng trưởng. Scrum không quy định thời gian dành cho mỗi Sprint nhưng thường khuyến nghị là 2 tuần. Lý do chính là, đây là khoảng thời gian vừa đủ để cả nhóm Scrum *chạy nước rút*, cũng không quá cảm thấy bị áp lực với phần tăng trưởng và đủ thời gian kiểm soát được những công việc của cá nhân cũng như cả nhóm. Khoảng thời gian ngắn cũng giúp Sprint Goal rõ ràng hơn. Việc Sprint có độ dài nhiều hơn 4 tuần là quá nguy hiểm với những rủi ro gấp phải đến từ sự thay đổi vì nhóm Scrum luôn giữ một Sprint Goal duy nhất và không chấp nhận thay đổi trong suốt Sprint.

Lý do để *mọi Sprint nên có chung một khoảng thời gian* là *nhịp đập* của công việc được duy trì ổn định, tạo thành một chu kỳ đều đặn tăng trưởng cũng như cải thiện quy trình và hiệu suất làm việc của nhóm. Tôi đã gặp nhiều nhóm phát triển sử dụng Sprint 4 tuần cho giai đoạn hình thành kiến trúc, thiết kế... và sử

dụng Sprint 2 tuần cho những Iteration sau đó. Cách làm này thường không tốt – bởi nếu dành quá nhiều thời gian và phân chia rạch ròi thành giai đoạn thiết kế, giai đoạn hình thành kiến trúc... sẽ rất dễ khiến nhóm quay lại với phương pháp truyền thống và khiến nhóm cảm thấy lạ lẫm với những Sprint ngắn hơn sau đó.

Một Sprint sẽ kết thúc khi timebox của Sprint đã hết hoặc khi Product Owner quyết định dừng Sprint. Việc dừng một Sprint giữa chừng thường ít khi xảy ra và cần được cân nhắc rất kỹ lưỡng vì việc này nói chung sẽ ảnh hưởng tới tinh thần của cả nhóm Scrum. Tuy nhiên, nếu Sprint Goal không còn đảm bảo hoặc đã lỗi thời, Sprint nên được dừng lại vì mục tiêu tối thượng của Scrum là hướng tới giá trị; khi Sprint Goal đã không còn giá trị thì Sprint không nên tiếp tục. Vì vậy, việc giữ cho Sprint có khoảng thời gian ngắn là vô cùng quan trọng, Sprint với thời lượng 4 tuần thường có Sprint Goal cũng như giá trị tạo ra ít rõ ràng hơn, và cũng có xác suất bị huỷ bỏ giữa chừng nhiều hơn. Ví dụ, nhóm Scrum đặt Sprint Goal là *tích hợp hệ thống chat trong ứng dụng mobile sử dụng Parse*; nhưng chỉ 5 ngày sau, Parse ra thông báo sẽ dừng hoạt động - điều này đồng nghĩa với Sprint Goal đã trở nên lỗi thời và việc kết thúc Sprint là hoàn toàn dễ hiểu để nhóm Scrum tập trung vào một mục tiêu mới là chuyển đổi toàn bộ dữ liệu từ Parse sang Firebase.



Một Sprint thường bắt đầu vào thứ Hai với sự kiện đầu tiên là Sprint Planning, và kết thúc vào thứ Sáu cùng sự kiện Sprint Retrospective giúp nhóm có khoảng thời gian nghỉ ngơi hợp lý.

Ngoài việc tránh rủi ro, Sprint với timebox 2 tuần cũng giúp nhóm phát triển có động lực tốt hơn bởi nhóm nhanh chóng nhìn thấy kết quả công việc. Nhóm Scrum nên được làm việc chung với nhau tại một địa điểm cụ thể cùng khoảng thời gian xác định trong ngày. Tôi đã gặp nhiều khó khăn khi làm việc trong nhóm Scrum phân tán và khác nhau về múi giờ. Lúc này, nhóm Scrum cần xác định với nhau một khoảng thời gian đảm bảo mọi thành viên có thể cộng tác cùng nhau, gọi là *core time*, cùng một cơ chế *work out loud*, mọi vấn đề đều được đưa vào một kênh thông tin chung của toàn nhóm. Nếu nhóm Scrum có những thành viên bị lệch nhau về múi giờ, theo tôi, nhóm nên chọn phương án giảm giờ làm của một số thành viên nhằm đảm bảo cả nhóm cùng bắt đầu và kết thúc Sprint vào những thời điểm cố định, hơn là một số thành viên bắt đầu Sprint sớm hơn và kết thúc muộn hơn – việc này khiến nhóm Scrum không nhất quán.

Sprint Planning – Lập kế hoạch cho Sprint

Các công việc được thực hiện trong Sprint được lên kế hoạch tại buổi họp Sprint Planning, đây cũng là sự kiện đầu tiên của mọi Sprint.

Thành phần tham dự:

- ScrumMaster
- Product Owner
- Nhóm phát triển

Thời gian:

- Tối đa là 8 giờ cho Sprint 4 tuần

Kết quả:

- Sprint Backlog
- Sprint Goal

Sprint Planning được sử dụng để trả lời 2 câu hỏi: *phân tăng trưởng của Sprint là gì? Và, làm thế nào để hoàn thành phân tăng trưởng?*

Phân tăng trưởng của Sprint là gì?

Phân tăng trưởng được lấy ra từ Product Backlog theo thứ tự ưu tiên. Product Owner sẽ nêu mong muốn của Sprint, những gì được kỳ vọng, phần giá trị có thể tạo ra từ những hạng mục còn lại trong Product Backlog. Vì những hạng mục trong Product Backlog được sắp xếp theo độ ưu tiên nên nhóm phát triển chỉ cần nhận những hạng mục này theo thứ tự từ trên xuống dưới. Tuy nhiên, Product Backlog thường có quá nhiều hạng mục và Sprint lại có khoảng thời gian cố định, nên nhóm phát triển chỉ nhận một số hạng mục dựa trên các yếu tố:

- *Năng lực hiện tại của nhóm.* Nhóm có bao nhiêu thành viên, mỗi thành viên có bao nhiêu thời gian, kiến thức của nhóm đã đủ thực hiện ngay hay cần thời gian học hỏi...
- *Hiệu suất làm việc trước đây của nhóm.* Trong những Sprint trước, nhóm có thể thực hiện được bao nhiêu công việc với năng lực hiện tại...

Những câu hỏi này chỉ có Nhóm phát triển mới có câu trả lời chính xác. Ví dụ, 10 hạng mục trong Product Backlog rất dễ lập trình nhưng lại khó kiểm thử, và nhóm chỉ có 1 người có kỹ năng kiểm thử thì việc yêu cầu cả nhóm thực hiện 10 phần tăng trưởng trong Sprint là không khả thi.

ScrumMaster cũng có thể trợ giúp Nhóm phát triển thông qua những dữ liệu

lịch sử đã thu thập được từ những Sprint trước như: *tốc độ của nhóm phát triển (velocity), độ trưởng thành của nhóm qua các Sprint, ...* để dự đoán được khả năng làm việc của nhóm trong Sprint này nếu giải quyết được những vấn đề của Sprint trước. Tuy nhiên, quyền quyết định cuối cùng vẫn thuộc về nhóm phát triển.

Sau đó Nhóm Scrum thống nhất Sprint Goal là những mục tiêu cần đạt được trong Sprint. Nói chung đây chính là phần tăng trưởng của Sprint và những giá trị được mang tới cho người dùng hay khách hàng; ngoài ra, những cải tiến về cách làm việc trong nhóm cũng nên được kể tới. Thông thường, Sprint Goal nên theo tiêu chí SMART.

Ví dụ một Sprint Goal:

- Khách hàng có thể sử dụng chức năng thanh toán qua thẻ VISA hoặc hỗ trợ ít nhất 3 ngân hàng nội địa.
- Cải thiện trang chủ để thời gian hiển thị dưới 1 giây trong điều kiện kết nối Internet bình thường.
- Tăng test coverage từ 70% lên 80%.
- Tăng việc cộng tác, trao đổi trong nhóm lên 10%.

Nói chung việc lập một Sprint Goal theo phương pháp SMART khá tốt vì kết thúc Sprint nhóm Scrum có thể nhìn lại và đánh giá được việc nhóm có hoàn thành Sprint Goal hay không. Sprint Goal cũng nên được diễn đạt ngắn gọn để nhóm Scrum dễ dàng ghi nhớ và theo dõi.

Làm thế nào để hoàn thành phần tăng trưởng?

Nhìn chung, đây là nhiệm vụ chỉ dành cho Nhóm phát triển, bởi không ai ngoài nhóm phát triển biết chính xác cách thức thực hiện Sprint Goal.

Nhóm phát triển thực hiện việc *làm rõ yêu cầu của từng phần tăng trưởng*, đảm bảo không hiểu sai; việc này được trợ giúp bởi Product Owner. Sau đó Nhóm phát triển thực hiện chia nhỏ phần tăng trưởng thành những công việc cụ thể và ước lượng thời gian thực hiện. Các vấn đề có thể gặp phải là:

- Độ lớn của từng công việc cụ thể *nên đủ nhỏ để tiện cho việc đồng bộ công việc* và tránh rủi ro, thời gian thực hiện dưới 1 ngày làm việc là tốt nhất.
- Có những vấn đề nằm ngoài sự hiểu biết của cả nhóm Scrum (ví dụ, các luật hiện tại về việc thanh toán qua thẻ tín dụng), nhóm Scrum có thể cần

sự tư vấn của chuyên gia ngoài nhóm. Nếu việc này xảy ra quá thường xuyên, chuyên gia này nên là thành viên trong nhóm Scrum.

- Ước lượng việc thực hiện Sprint Goal không giống với ước lượng sau khi đã phân tách thành các công việc cụ thể. Ví dụ, chức năng thanh toán theo ước lượng ban đầu cần 48 giờ để thực hiện nhưng khi phân tách thành những công việc cụ thể như xây dựng giao diện, kiểm tra số thẻ tín dụng, kiểm tra tài khoản... cần tới 70 giờ để thực hiện. Khi đó nhóm phát triển thoả thuận lại với Product Owner về Sprint Goal bởi ước lượng sau khi đã phân tách phần tăng trưởng thành những công việc cụ thể thường chính xác hơn so với ước lượng ban đầu. Lúc này Sprint Goal có thể được điều chỉnh, hoặc *scope, acceptant criteria* của một số chức năng có thể được điều chỉnh.

Kết quả của phần thảo luận này là Sprint Backlog, bao gồm những công việc cần thực hiện để hoàn thành Sprint Goal. Trái với Product Backlog được quản lý bởi Product Owner, *Sprint Backlog* được quản lý bởi nhóm phát triển; và thậm chí, đôi khi, các phần tăng trưởng có độ ưu tiên không giống với độ ưu tiên trong Product Backlog, nếu nhóm phát triển tin rằng đấy là cách tốt nhất để làm việc và đạt được Sprint Goal.

Kết thúc sự kiện Sprint Planning, nhóm phát triển giải thích cho Product Owner và ScrumMaster về cách nhóm sẽ làm việc, tự tổ chức để đạt được Sprint Goal.

Một phương pháp ước lượng (estimation) không hẳn sinh ra cho phương pháp Agile nhưng được hầu hết những nhóm Agile áp dụng là Planning Poker. Những thành viên trong nhóm *chơi bài (planning poker)* với nhau khi độc lập đưa ra những con số cho biết ước lượng của mình đối với từng công việc cụ thể; dựa trên 2 đơn vị thường dùng là:

- Dãy Fibonacci: 1, 2, 3, 5, 8, 13...
- Kích cỡ T-shirt: S, M, L, XL...

Nói chung, dãy số này được đưa ra theo quy luật *tăng rất nhanh về sau*. Vì sao? Để tăng độ chính xác. Cách duy nhất để tăng độ chính xác của ước lượng là giảm thiểu sai số; khi con số ước lượng đã quá lớn như 8 hay 9 là rất khó xác định con số nào đáng tin hơn; ngược lại, việc lựa chọn giữa 8 và 13 sẽ trở nên khác biệt.

Việc này cũng giúp nhóm phát triển dễ đạt đến sự đồng thuận. Về nguyên tắc, khi những thành viên trong nhóm đưa ra những ước lượng khác xa nhau, đó là dấu hiệu cho thấy có những cách hiểu khác biệt giữa những thành viên về cùng

một công việc. Đó là lúc từng thành viên và Product Owner giải thích và thống nhất cách hiểu cũng như đạt sự đồng thuận trong ước lượng. Từ đó, dẫn đến cam kết thực hiện.



Ngoài dãy Fibonacci và kích thước T-shirt, có thể sử dụng kỹ thuật nào khác?

Đơn vị ước lượng được các nhóm hay sử dụng là *giờ* vì nó rất trực quan và dễ hiểu. Song Scrum khuyến nghị sử dụng một đơn vị *trung tính* hơn là *Story Point*. Đơn vị này giúp nhóm dễ so sánh độ lớn của những *User Story* với nhau trước khi thực hiện, bởi trước đó nhóm sẽ không biết chính xác *User Story* đơn vị (*1 story point*) cần bao nhiêu thời gian để thực hiện.



Để hiểu cách sử dụng Story Point, nhóm Scrum có thể thực hành như sau:

Liệt kê tất cả những *User Story* trong Product Backlog, cùng nhau thống nhất việc lựa chọn 1 *User Story* nhỏ nhất (theo nỗ lực) dựa trên việc trao đổi và đồng thuận, đặt *User Story* đó với đơn vị 1 Story Point và đánh giá Story Point của những *User Story* khác dựa trên kích thước tương đối so với *User Story* này. Sau đó, với *User Story* đơn vị, nhóm có thể phân tách thành những công việc cụ thể và thực hiện việc ước lượng thông qua số giờ cần thực hiện. Sau đó, nếu muốn ước lượng số giờ thực hiện những *User Story* khác, nhóm có thể nội suy theo *User Story* đơn vị.

Việc này giúp Story Point là một đơn vị *trung tính*, không phải là một đơn vị cố định như giờ, và chỉ sử dụng trong nhóm Scrum cố định.

Nhóm Scrum cũng có thể đặt 2 Story Point cho *User Story* nhỏ nhất được chọn ban đầu, để giữ 1 cho những *User Story* nhỏ hơn được thêm vào Product Backlog sau này.

Về cơ bản, *Sprint Planning* thường là nguồn gốc của những Sprint thất bại trong giai đoạn đầu, chủ yếu do cách ước lượng không chính xác. Cách duy nhất tổ chức có thể làm, là chấp nhận việc đó, ghi nhận thông tin và kỳ vọng những dữ liệu đó sẽ giúp ích cho những Sprint tiếp theo. Bởi đơn giản là, ước lượng có sự sai lệch lớn khi thông tin thiếu rõ ràng; sau một vài Sprint, nhóm sẽ có ước lượng tốt hơn bởi có nhiều dữ liệu hơn.

Daily Scrum

Daily Scrum là sự kiện Scrum được ngay cả những tổ chức không thực hành Scrum ưa chuộng và học theo, nhưng lại dễ làm sai nhất.

Thành phần tham dự:

- Nhóm phát triển
- *ScrumMaster*

Thời gian:

- Tối đa là 15 phút

Kết quả:

- Kế hoạch được cập nhật cho ngày

Daily Scrum là cuộc họp diễn ra trong tối đa 15 phút, tại một thời điểm cố định, tại một vị trí cố định, trong đó các thành viên trong nhóm phát triển lần lượt trả lời đúng 3 câu hỏi sau:

- Tôi đã làm gì trong 24 giờ đã qua để đạt được Sprint Goal?
- Tôi sẽ làm gì trong 24 giờ tiếp theo để đạt được Sprint Goal?
- Tôi thấy có trở ngại gì ảnh hưởng tới tôi hoặc nhóm phát triển để đạt được Sprint Goal?

Sau khi mọi thành viên của nhóm phát triển trả lời đủ 3 câu hỏi này, nhóm phát triển sẽ đồng bộ kế hoạch với nhau và có ngay kế hoạch cho 24 giờ tiếp theo, được cập nhật trong Sprint Backlog, để hướng tới Sprint Goal.

Chỉ nhóm phát triển tham dự Daily Scrum, và chỉ họ thực hiện việc đồng bộ công việc của nhóm bằng việc trả lời 3 câu hỏi trên. ScrumMaster cũng có thể tham gia với vai trò *quan sát viên* nhằm góp ý cho Nhóm phát triển nếu họ thực hiện không đúng phương pháp hoặc vượt quá thời gian cho phép. ScrumMaster cũng là người đảm bảo sự kiện Daily Scrum xảy ra và thúc đẩy sự kiện cho kết quả cao nhưng không phải là người chủ trì cuộc họp.

Daily Scrum thường được gọi là daily standup meeting. Cách tổ chức họp (đứng, ngồi) hoàn toàn do nhóm phát triển quyết định cho phù hợp, Scrum không quy định điều này. Nhưng các nhóm Scrum thường chọn cách *hop đứng* để hiệu quả cao hơn, thúc đẩy cuộc họp kết thúc trong timebox 15 phút. Thậm chí với những nhóm làm việc từ xa, việc họp đứng qua Internet cũng rất hiệu quả.

Daily Scrum được thực hiện vào một thời điểm cố định trong ngày, tại một địa điểm cố định. Bởi đây là công việc được thực hiện hàng ngày như một thói quen của nhóm phát triển, việc cố định thời điểm, địa điểm làm giảm những trao đổi không cần thiết; mọi thành viên trong nhóm đều đến hẹn lại lên, tiết kiệm thời gian cũng như sự phân tâm không biết ngày nay họp ở đâu, vào thời gian nào.

Daily Scrum chỉ diễn ra trong timebox 15 phút. Rất nhiều nhóm phát triển không thể thực hiện được điều này. Thông thường, với nhóm gồm 9 thành viên, mỗi thành viên nếu thực hiện đúng việc chỉ trả lời 3 câu hỏi này thì chắc chắn sự kiện sẽ diễn ra trong timebox. Nhưng rất nhiều nhóm Scrum nhận định câu hỏi thứ 3 của mỗi cá nhân như một câu hỏi mở và cố gắng giải quyết những vấn đề trở ngại ngay trong Daily Scrum, và khiến sự kiện này không có hồi kết, có khi mất vài giờ. Chúng ta cần hiểu rõ ý nghĩa của Daily Scrum là để đồng bộ công việc đã làm, lên kế hoạch thực hiện cho 24 giờ tiếp theo để đạt được Sprint Goal, không phải để cùng nhau giải quyết công việc. Ví dụ, trong Daily Scrum, An nói “Tôi không thể thiết lập được môi trường để kiểm thử cho chức năng này”, thì Bình, một thành viên khác, có thể lên tiếng “Tôi có thể trợ giúp, nhưng phải sau 3 giờ nữa, vì việc fix bug hiện tại quan trọng hơn”, và nếu Cường nói rằng “Tôi có thể trợ giúp sau 1 giờ nữa”, thì nhóm sẽ nhanh chóng cho ra được kế hoạch để An và Cường làm việc cùng nhau nhằm giải quyết vấn đề An đã đưa ra sau 1 giờ nữa, không phải giải quyết ngay tại Daily Scrum. Những trở ngại khác cũng được giải quyết theo cách tương tự. Tôi đã thấy một số tổ chức thực hiện một cuộc họp khác hàng ngày ngoài Daily Scrum để cả nhóm giải quyết những trở ngại. Tuy vậy, tôi không cho đó là phương án tốt; phương án tốt hơn là thực hiện một cơ chế *work out loud* nơi những thành viên trong nhóm phát triển có thể đưa ra vấn đề của mình bất cứ lúc nào và có thể nhận được sự hỗ trợ sớm nhất, chứ không phải chờ tới Daily Scrum. Bởi Scrum khuyến khích sự cộng tác và lên lịch tức thì, Daily Scrum chỉ là một phương thức cụ thể hoá việc cộng tác, đồng bộ công việc theo chu kỳ 1 ngày, hoàn toàn không phải việc báo cáo. Chúng ta sẽ bàn đến vấn đề này như một trong những cách để nâng cao hiệu suất làm việc vào các chương sau.

Daily Scrum chấp nhận những câu trả lời như “tôi không làm gì để đạt được Sprint Goal trong 24 giờ đã qua”, “tôi sẽ không làm gì để đạt được Sprint Goal trong 24 giờ tới”. Daily Scrum không phải một cơ chế báo cáo buộc mọi người phải “vẽ” ra một thứ nào đó giả tạo, Daily Scrum là cơ hội để nhóm phát triển hiểu thực chất

chuyện gì đang xảy ra, những công việc cần làm và cần được đồng bộ với nhau nhằm đạt được Sprint Goal. Vì thế, sẽ rất tốt nếu một thành viên trong nhóm phát triển cho biết “*tôi gặp rắc rối về mặt tinh thần và đã không làm gì trong 24 giờ đã qua để đạt được Sprint Goal*”, bởi cả nhóm phát triển sẽ biết cách trợ giúp thành viên đó có tinh thần tốt hơn để làm việc, hoặc có phương án nhận lại những công việc còn dang dở. Sẽ rất tệ nếu chúng ta đánh bóng bằng việc nêu những điều không đúng sự thật gây cản trở cho mục tiêu đồng bộ công việc. Những tổ chức với cách quản lý truyền thống yêu cầu nhân viên khai báo rõ ràng những khoảng thời gian nào trong ngày được sử dụng cho công việc gì, dẫn tới những báo cáo kiểu như *4 giờ cài đặt môi trường, 4 giờ kiểm thử chức năng A* trong khi nhân viên đó chỉ làm hết công việc đó có 1 giờ. Những thông tin sai lệch sẽ khiến những quyết định sai lệch; ví dụ, mọi người sẽ nhìn nhận nhân viên đó có trình độ rất kém khi mất tới 8 giờ làm công việc của 1 giờ. Việc liên tục có những câu trả lời “*tôi không làm gì để đạt được Sprint Goal trong 24 giờ đã qua*”, “*tôi sẽ không làm gì để đạt được Sprint Goal trong 24 giờ tới*”, sẽ giúp ScrumMaster và Product Owner nhận diện được rằng, có khi Sprint Goal không phù hợp với nhóm phát triển hoặc tinh thần và cam kết của thành viên đó đang gặp trở ngại và có sự điều chỉnh cho phù hợp.



Trong Daily Scrum, nhóm đứng thành một vòng tròn và thực hiện việc trả lời 3 câu hỏi trên theo thứ tự chiều kim đồng hồ. Nếu một thành viên không làm gì để hướng tới Sprint Goal trong 24 giờ đã qua và 24 giờ tiếp theo, họ nên lựa chọn cách nói nào, “*bỏ qua tôi đi*” hay “*tôi đã không làm gì trong 24 giờ đã qua và cũng sẽ không làm gì trong 24 giờ tới*”?

Sprint Review

Sprint Review là sự kiện được tổ chức vào cuối Sprint để đánh giá kết quả của Sprint.

Thành phần tham dự:

- Nhóm phát triển
- ScrumMaster
- Product Owner
- Các bên liên quan

Thời gian:

- Tối đa là 4 giờ với Sprint 4 tuần

Kết quả:

- Product Backlog được chỉnh sửa cho phù hợp với Sprint tiếp theo

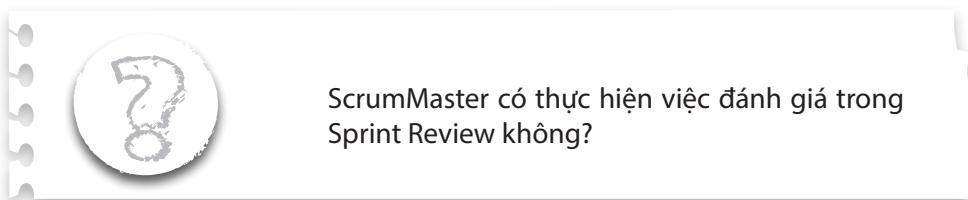
Trong Sprint Review, Nhóm phát triển trình bày những phần tăng trưởng đã được hoàn thành, những khó khăn gặp phải và cách nhóm đã giải quyết để cố gắng đạt được Sprint Goal.

Sau đó, lần lượt qua từng điểm trong Sprint Goal:

- Mọi người sử dụng sản phẩm, đặc biệt là phần tăng trưởng.
- Product Owner và các bên liên quan nhận định xem phần tăng trưởng đó:
 - có đáp ứng đúng được yêu cầu đặt ra?
 - có đáp ứng được nhu cầu của người dùng?
 - cần chỉnh sửa gì cho phù hợp với ngữ cảnh mới? Và đặt độ ưu tiên cho việc chỉnh sửa này (nếu có).

Thông thường nếu Product Owner làm việc hiệu quả cùng nhóm phát triển thì việc đáp ứng không đúng yêu cầu đã đặt ra là rất hiếm gặp. Nhưng phần tăng trưởng đó có thực sự phù hợp với ngữ cảnh hiện tại không thì có thể cần những bên liên quan đánh giá và tư vấn. Tuy nhiên, quyền quyết định cuối cùng vẫn thuộc về Product Owner.

Sau Sprint Review, sẽ có thể có thêm một số công việc mới cần hoàn thành phù hợp với ngữ cảnh hiện tại, Product Owner sắp xếp lại độ ưu tiên của những công việc này trong Product Backlog để sẵn sàng cho Sprint tiếp theo.



ScrumMaster có thực hiện việc đánh giá trong Sprint Review không?

Sprint Retrospective

Sprint Retrospective là một sự kiện đặc biệt quan trọng trong Scrum. Tuy vậy, rất ít tổ chức thực hiện hoặc thực hiện không đúng Sprint Retrospective khiến nhóm Scrum không đạt được hiệu quả như mong muốn.

Thành phần tham dự:

- Nhóm phát triển
- ScrumMaster
- Product Owner

Thời gian:

- Tối đa là 3 giờ với Sprint 4 tuần

Kết quả:

- Cải tiến quy trình

Sprint Retrospective là cơ hội để nhóm Scrum nhìn lại chính mình để tìm ra cách thức phù hợp và cải tiến quy trình cho chính mình. Thật ra phương pháp này không mới, quân đội Mỹ đã áp dụng một phương pháp tương tự từ khá sớm là AAR (*After Action Review*) tuy nhiên, trong lĩnh vực phát triển phần mềm thì việc đánh giá, tổng kết, học hỏi chưa bao giờ được đánh giá đúng mực. Các nhóm phát triển phần mềm theo phương pháp truyền thống sau khi “chiến đấu” cùng dự án trong thời gian dài thường có xu hướng nghỉ ngơi sau khi dự án được đóng lại; hoặc nếu họ có làm thì cũng không mấy hiệu quả bởi quá khó để nhìn lại những gì đã xảy ra trong khoảng thời gian dài đến vài tháng. Sprint Retrospective cũng giải thích tại sao Scrum hay Agile thường được coi là một mô hình đề cao và hướng tới tinh thần học hỏi, sáng tạo.

Trong Sprint Retrospective, nhóm Scrum cùng nhìn lại Sprint đã qua:

- Xảy ra như thế nào trên các khía cạnh về con người, sự cộng tác, quy trình và công cụ?
- Những điều gì đã được thực hiện tốt và cách thực hiện?
- Những điều gì có thể thực hiện tốt hơn và cách thực hiện cho lần tiếp theo?

Kết thúc Sprint Retrospective, nhóm Scrum đề ra được những hành động tiếp theo nhằm cải tiến quy trình của nhóm để đạt hiệu quả tốt hơn.

Sprint Retrospective

 <p>Sẽ làm:</p> <ul style="list-style-type: none"> - Sử dụng pull request - Luộc bỏ code 	 <p>Duy trì:</p> <ul style="list-style-type: none"> - Họp hàng ngày - Cách review code 	 <p>Hành động tiếp theo:</p> <ul style="list-style-type: none"> - Chuyển sang Github - Sử dụng Framework 3.0
 <p>Không làm:</p> <ul style="list-style-type: none"> - Chủ trịch công nghệ 	 <p>Đang làm</p> <ul style="list-style-type: none"> - Viết tài liệu 	

Hình 3.5: Sprint Retrospective

Sprint Retrospective không phải là một hoạt động chỉ trích. Dù kết quả của Sprint ra sao, Sprint Retrospective không bao giờ là một nơi để nhóm Scrum chỉ trích những cá nhân khác hoặc tổ chức. Đây là một không gian mở để nhận diện ra những vấn đề trong nhóm và cách giải quyết phù hợp với ngữ cảnh hiện tại của nhóm Scrum.

Sprint Retrospective hạn chế để cập tới những vấn đề công nghệ. Ví dụ, “nếu chúng ta sử dụng Java thì việc này đã xong bằng cách này..., cách này...”. Bởi Sprint Retrospective là nơi cải tiến cách nhóm Scrum hoạt động; vì vậy, sẽ là tốt hơn nếu nhóm Scrum trả lời câu hỏi “tại sao chúng ta lại không quyết định chọn Java?” và “làm thế nào để lần sau chúng ta sẽ đưa ra quyết định đúng trong trường hợp tương



Hãy nhớ rằng ScrumMaster không phải là người quản lý và đưa ra quyết định trong nhóm Scrum, vì thế ScrumMaster nên định hướng nhóm Scrum một cách khéo léo bằng cách cung cấp những thông tin cần thiết. Ví dụ, ScrumMaster đưa ra thông tin “số bug trong Sprint này là 10, tăng 3 so với Sprint trước” và nhóm phát triển sẽ phân tích vấn đề về đó qua việc quản lý chất lượng, việc cộng tác... và đưa ra hành động; thay vì ScrumMaster là người đưa ra ý tưởng.

tự?", hơn là chỉ đơn thuần dừng lại ở "đúng là lần sau nên chọn Java". Tất nhiên, đây là một việc khó, đòi hỏi ScrumMaster định hướng đúng và cả nhóm Scrum cùng thực hiện.

Những hành động tiếp theo rút ra từ Sprint Retrospective cần có sự đồng thuận trong cả nhóm Scrum. Việc thực hiện một công việc hay quy trình chỉ thực sự đạt hiệu quả cao nếu được xây dựng trên sự đồng thuận, không phải ép buộc. Vì vậy, ScrumMaster nên là người phân tích lợi ích của những công việc nên làm trong nhóm, không phải là người quyết định quy trình cho nhóm Scrum.

Sprint Retrospective có nhiều phương pháp để thực hiện. ScrumMaster có thể áp dụng những phương pháp khác nhau cho những sự kiện Sprint Retrospective khác nhau để tăng sức sáng tạo trong nhóm Scrum.

Sprint Retrospective cần khuyến khích xung đột tích cực. Bởi chỉ sự xung đột tích cực mới mang lại sự thật và cách làm mới phù hợp. ScrumMaster tuyệt đối không phải người phân xử khi xung đột xảy ra; ngược lại, họ thúc đẩy sự tự giải quyết xung đột giữa các thành viên trong nhóm, giúp họ tự tìm ra cách làm việc tốt nhất với chính mình cũng như gắn kết với nhau hơn.

Sprint Retrospective là hoạt động cần thông minh và tinh táo. Thông thường Sprint Retrospective là sự kiện nối tiếp ngay sau Sprint Review bởi khi đó nhóm Scrum còn nhớ được những gì đã xảy ra trong Sprint hiện tại. Vì thế, nhóm Scrum rất khó tránh khỏi tâm trạng hân hoan khi Sprint kết thúc hoặc chán nản nếu không đạt được Sprint Goal dẫn đến những tư tưởng như "quá tốt, không còn gì để bàn", "không thay đổi được gì đâu". ScrumMaster chính là người phải dẫn dắt nhóm Scrum giữ được sự tinh táo để biết những gì nhóm đã làm tốt, cần tiếp tục phát huy; những gì có thể cải thiện và hành động tiếp theo.

SCRUM ARTIFACTS

Artifact (tạo tác) trong Scrum được hiểu là những thể hiện của công việc hoặc giá trị, được minh bạch nhằm tạo khả năng thanh tra và thích ứng. Trong phần này tôi giới thiệu những Artifact được sử dụng nhiều, song bạn cần lưu ý những Artifact chuẩn của Scrum chỉ bao gồm: Product Backlog, Sprint Backlog, Product Increment.

User story

User Story (câu chuyện người dùng) là một thuật ngữ để nêu ra yêu cầu trong phát triển phần mềm với Agile.

Lý do gọi là User Story vì yêu cầu được viết dưới góc nhìn của người dùng, và vì vậy giúp nhóm phát triển hiểu được giá trị họ mang lại cho người dùng, chứ không phải là thực hiện những công việc khô khan, nhằm khuyến khích họ thực hiện công việc.

User Story thường có dạng:

Là <người dùng>, tôi muốn <mục tiêu>, để <giá trị>

Ví dụ: *Là nhân viên, tôi muốn xem lịch làm việc của đồng nghiệp cùng phòng, để biết chúng tôi có thể đổi lịch làm việc cho nhau nếu cần hay không.*



Hình 3.6: Một số tính chất của User Story

Một User Story được coi là tốt nếu đáp ứng được theo mô hình INVEST:

- Independent (độc lập): User Story này không bị ràng buộc với những User Story khác, và có thể triển khai một cách độc lập;
- Negotiable (có thể thương lượng): User Story có thể được thương lượng về mặt scope, accept criteria để đảm bảo được hoàn thành trong nhiều nhất một Sprint;
- Valuable (có giá trị): User Story phải cung cấp giá trị kinh doanh, nghiệp vụ cho người dùng; điều này cũng giúp Nhóm phát triển hào hứng hơn khi nhìn thấy công việc của mình không lãng phí;
- Estimable (có thể ước lượng): User Story phải có thể ước lượng được theo Story Point hoặc giờ để đảm bảo hoàn thành trong tối đa một Sprint;
- Small (nhỏ): User Story phải đủ nhỏ để dễ hiểu và hoàn thành trong tối đa trong một Sprint.

- Testable (có thể kiểm thử): User Story phải có thể định nghĩa việc kiểm thử, accept criteria nhằm đánh giá được mức độ hoàn thành trong Sprint Review.

Tất nhiên, để đạt được những yêu cầu này tương đối khó, khi User Story chỉ xuất phát từ một ý tưởng của Product Owner hay người dùng, bởi vậy Product Owner cần áp dụng những kỹ thuật khác nhau để làm rõ User Story, từ đó Nhóm phát triển có thể thực hiện việc biến User Story thành một phần tăng trưởng trong Sprint.

Vì vậy, User Story cần cập nhật liên tục và thường có những giai đoạn sau:

Giai đoạn	Thời điểm	Tính chất	Ví dụ
Ý tưởng	Chủ sản phẩm hoặc người dùng đưa ra ý tưởng	Sơ sài, không rõ ràng	Là nhân viên, tôi muốn biết lịch làm việc của đồng nghiệp cùng phòng ban để tiện cho việc cộng tác.
Cụ thể	Khi đưa vào Product Backlog	Cụ thể về yêu cầu, xác định được giá trị Theo định dạng chung, dễ quản lý	Là nhân viên, tôi muốn xem lịch làm việc của đồng nghiệp cùng phòng, để biết chúng tôi có thể đổi lịch làm việc cho nhau nếu cần hay không.
Sẵn sàng	Khi đưa vào Sprint Backlog	Cụ thể về yêu cầu, xác định được giá trị Đủ nhỏ để triển khai Có thể đề ra tiêu chí kiểm thử rõ ràng Thoả mãn SMART	Là nhân viên, tôi muốn xem lịch làm việc của đồng nghiệp cùng phòng, để biết chúng tôi có thể đổi lịch làm việc cho nhau nếu cần hay không. Lịch làm việc cần hiển thị theo dạng bảng, với các cột hiển thị cho 7 ngày trong tuần, các dòng là danh sách đồng nghiệp được sắp xếp theo alphabet.

Giai đoạn	Thời điểm	Tính chất	Ví dụ
Xác nhận	Trong Sprint, sau khi nhóm phát triển thương lượng với Product Owner về phạm vi của User Story	Có thêm những ghi chú về lịch sử thay đổi	<p>Là nhân viên, tôi muốn xem lịch làm việc của đồng nghiệp cùng phòng, để biết chúng tôi có thể đổi lịch làm việc cho nhau nếu cần hay không. Lịch làm việc cần hiển thị theo dạng bảng, với các cột hiển thị cho 7 ngày trong tuần, các dòng là danh sách đồng nghiệp được sắp xếp theo alphabet.</p> <p>XÁC NHẬN:</p> <ul style="list-style-type: none"> - Danh sách đồng nghiệp chỉ hiện trên một màn hình, không phân trang - Hiển thị cùng lịch làm việc của nhân viên hiện tại <p>ĐIỀU CHỈNH:</p> <ul style="list-style-type: none"> - Hiển thị cho 5 ngày làm việc



Hình 3.7: Các giai đoạn của User Story

Đôi khi có những User Story quá lớn, cần nhiều thời gian hơn một Sprint để thực hiện, lúc đó khái niệm *Epic* được sử dụng và được phân tách thành nhiều User Story. Ví dụ, “*là một nhà quản lý, tôi muốn quản lý lịch làm việc của nhân viên*” là quá lớn đối với một User Story, Epic đó nên được phân tách thành nhiều User Story như “*là một nhà quản lý, tôi muốn tạo lịch làm việc của một nhân viên*”, “*là một nhà quản lý, tôi muốn chỉnh sửa lịch làm việc của một nhân viên*”, “*là một nhà quản lý, tôi muốn xoá lịch làm việc của một nhân viên*”, ...

Một tập hợp những User Story có liên quan tới nhau có thể được tập hợp lại và gọi là *Theme*. Ví dụ, “*là một nhân viên, tôi muốn gửi email tới đồng nghiệp thông qua hệ thống*”, “*là một nhân viên, tôi muốn gửi SMS đến đồng nghiệp thông qua hệ thống*” có thể nằm trong 1 Theme là “*trao đổi giữa những nhân viên*”.

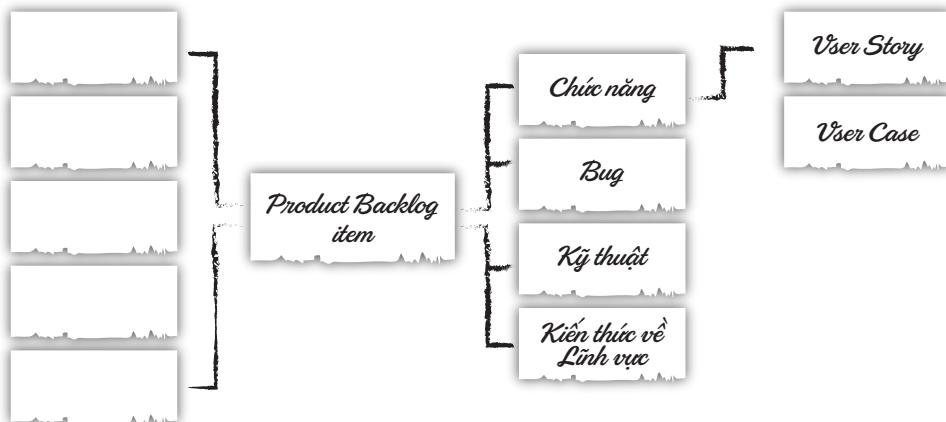
Có hai vai trò thường xuyên làm việc với User Story là Product Owner và Nhóm phát triển. Product Owner cần định giá được User Story, tức là biết được User Story nào mang lại giá trị lớn hơn và có độ ưu tiên cao hơn. Nhóm phát triển cần ước lượng được User Story, tức là dự đoán User Story đó cần bao nhiêu nguồn lực (thời gian) để hoàn thành, có gặp khó khăn gì không. User Story thường được ước lượng thông qua 1 trong 2 chỉ số là *Story Point* hoặc *số giờ*, đều đại diện cho nguồn lực cần thiết để hoàn thành User Story.



Hãy thử làm rõ qua những giai đoạn từ Ý tưởng tới Xác nhận cho User Story sau: *Là người bán hàng, tôi muốn quản lý số điện thoại của những khách hàng thân thiết nhằm hỗ trợ chương trình hậu mãi.*

Product Backlog

Product Backlog là một danh sách được sắp xếp theo độ ưu tiên, bao gồm mọi thứ để hoàn thiện sản phẩm. Product Owner là người duy nhất chịu trách nhiệm quản lý Product Backlog.



Hình 3.8: Product Backlog

Product Backlog là nơi duy nhất chứa yêu cầu về sản phẩm, nhưng không phải chỉ chứa những yêu cầu sản phẩm. Thực tế, Product Backlog sẽ chứa rất nhiều User Story với một nhóm thực hành tốt Scrum; ngược lại, rất nhiều bug với một nhóm thực hành không tốt Scrum.

Product Backlog tồn tại cùng dự án. Product Backlog còn tồn tại hạng mục tức là còn có những công việc phải làm để hoàn thành dự án hay sản phẩm. Thực tế thì Product Backlog dường như dài vô hạn, chỉ là Product Owner quyết định dừng dự án hay triển khai sản phẩm vào một thời điểm thích hợp mà thôi.

*Những hạng mục phía trên của Product Backlog thường rõ ràng hơn. Vì Product Backlog được sắp xếp theo độ ưu tiên, nên những hạng mục phía trên Product Backlog sẽ là những hạng mục được thực hiện trước, và nhu cầu cần sự rõ ràng cấp thiết hơn. Product Owner cần đảm bảo những User Story trên cùng của Product Backlog đạt trạng thái *Sẵn sàng* khi bắt đầu Sprint. Một công việc nếu chưa đạt tới trạng thái *Sẵn sàng* thì dù có độ ưu tiên cao nhất cũng không nên đưa vào Sprint, vì chúng ta không thể đánh giá được công việc đó vào cuối Sprint khi yêu cầu không rõ ràng và nguồn lực bỏ ra sẽ trở thành lãng phí.*



Độ ưu tiên của những hạng mục trong Product Backlog có thể được thực hiện dễ dàng thông qua phương pháp MoSCoW như sau:

- *Must*: Những hạng mục phải thực hiện để đạt được giá trị kinh doanh
- *Should*: Những hạng mục nên thực hiện để có thêm giá trị kinh doanh
- *Could*: Những hạng mục có thể thực hiện để bổ sung giá trị kinh doanh trong điều kiện cho phép
- *Won't*: Những hạng mục sẽ không thực hiện vì nỗ lực không tương xứng với giá trị kinh doanh đạt được

Product Owner nên chuẩn bị kỹ những hạng mục nằm trong danh sách Must (nói chung ít) và đánh giá độ ưu tiên của những hạng mục này. Một Sprint cũng có thể không chỉ gồm những hạng mục Must nếu một hạng mục Should có thể giúp tạo động lực hơn cho nhóm phát triển. Hãy lưu ý rằng việc đánh giá MoSCoW phải được Product Owner thực hiện lại trên Product Backlog sau mỗi Sprint ngay cả không có sự thay đổi trong yêu cầu, ví dụ một số bug được tạo ra trong Sprint trước ảnh hưởng tới giá trị kinh doanh và cần nằm trong danh sách *Must*.

Sprint Backlog

Sprint Backlog như tên gọi là Backlog cho Sprint, thay vì Backlog cho cả sản phẩm như Product Backlog.

Sprint Backlog không hẳn là một phần của Product Backlog. Khi bắt đầu Sprint, Sprint Backlog bao gồm những item được lựa chọn từ của Product Backlog. Tuy nhiên, ngay trong buổi Sprint Planning, nhóm phát triển đã cụ thể hoá những item này thành những công việc nhỏ hơn; do đó Sprint Backlog thường bao gồm những công việc rất cụ thể như viết code logic, tạo giao diện thay vì chỉ đơn giản là *là người dùng, tôi muốn đăng nhập vào hệ thống* như item trong Product Backlog.

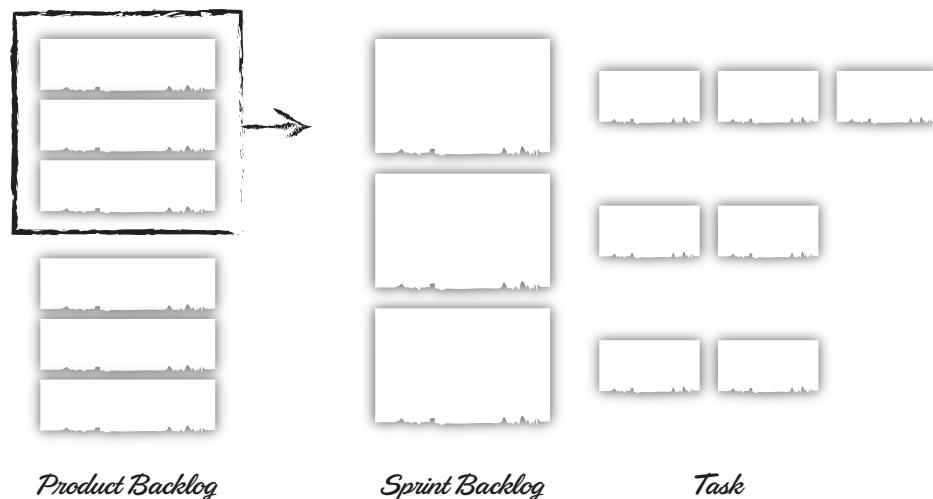
Sprint Backlog được quản lý chỉ bởi Nhóm phát triển, cũng giống như Product Backlog được quản lý chỉ bởi Product Owner. ScrumMaster có thể theo dõi, hướng dẫn

Nhóm phát triển để đảm bảo việc quản lý Sprint Backlog một cách đúng đắn và hiệu quả, song không được làm thay nhóm phát triển trong việc cập nhật, quản lý.

Sprint Backlog liên tục được cập nhật. Sprint Backlog thể hiện đúng tình trạng hiện tại của Sprint và được cập nhật liên tục, thậm chí từng giờ. Những công việc nào cần làm thêm, những công việc nào có thể loại bỏ để đạt được Sprint Goal đều cần được ghi nhận tức thì.

Những item trong Sprint Backlog có thể được xem xét và đưa vào Product Backlog khi kết thúc Sprint. Khi một User Story không được hoàn thành vào cuối Sprint, toàn bộ User Story đó sẽ được đưa trở lại Product Backlog, mặc dù một phần công việc cụ thể của User Story đã hoàn thành.

Những item trong Sprint Backlog phải được ước lượng trước khi bắt đầu. Điều này giúp nhóm phát triển đánh giá được khối lượng công việc còn lại và khả năng đạt được Sprint Goal ngay tại thời điểm bắt đầu.

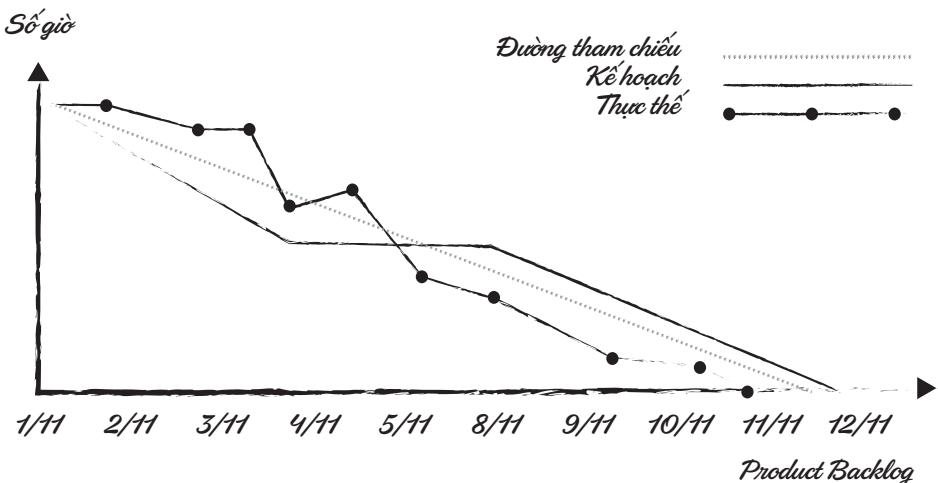


Hình 3.9: Sprint Backlog

Tương tự như Product Backlog, Sprint Backlog cũng phải được sắp xếp theo độ ưu tiên về giá trị. Có nên sử dụng MoSCoW để sắp xếp những item trong Sprint Backlog như ví dụ trên?

Burn-down Chart

Burn-down Chart (Biểu đồ đốt cháy) là một artifact đã được lược bỏ khỏi những artifact chuẩn của Scrum, tuy nhiên vẫn được ưa chuộng như một artifact để theo dõi tiến trình và khả năng đạt được Release Goal hay Sprint Goal.



Hình 3.10: Sprint Burn-down Chart

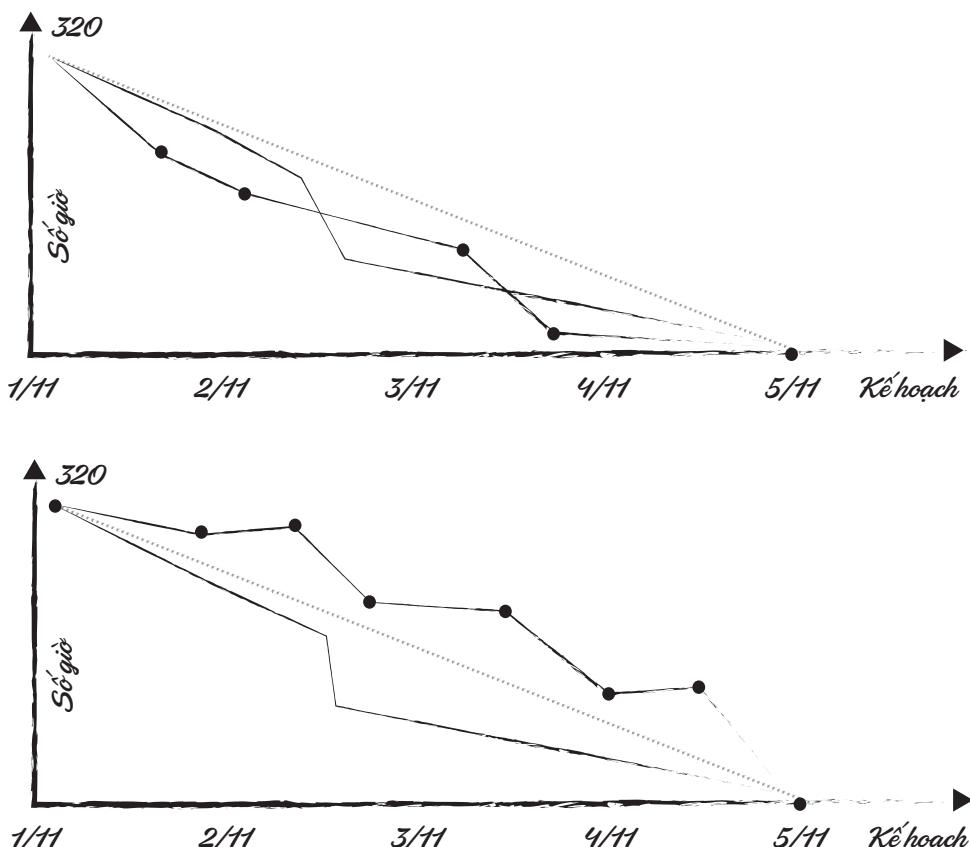
Có 2 loại Burn-down Chart phổ biến là Release Burn-down Chart và Sprint Burn-down Chart, hướng tới việc theo dõi tiến độ và khả năng đạt được của Release Goal và Sprint Goal. Như tên gọi, 2 loại biểu đồ này nhằm cung cấp thông tin về khả năng nhóm Scrum có thể đạt được những mốc để release sản phẩm hay Sprint.

Burn-down Chart không có chuẩn. Việc trực quan hóa thông tin gì hoàn toàn do nhóm Scrum quyết định; nhiều nhóm Scrum chỉ trực quan hóa tiến độ một cách đơn giản; nhiều nhóm Scrum trực quan hóa những thông tin khác như khối lượng công việc thêm vào, khối lượng công việc còn tồn đọng... Những thông tin bổ sung có thể cho nhóm Scrum biết được tần suất những công việc được tạo ra so với ước lượng ban đầu, số bug được tạo ra trong Sprint... Những thông tin này hữu ích cho việc thực hiện Sprint Planning cho Sprint tiếp theo. Tuy nhiên, quản lý quá nhiều thông tin là việc không đơn giản và mất nhiều thời gian nếu điều này không thực sự cần thiết với nhóm Scrum, chúng nên được loại bỏ để tránh lãng phí và tập trung vào Sprint Goal.

Burn-down Chart có thể được thay bằng Burn-up Chart. Thay vì trực quan hóa khối lượng công việc hiện tại, nhóm Scrum có thể trực quan hóa khối lượng

công việc đã hoàn thành. Về cơ bản, thông tin mang lại là như nhau, nhưng với góc nhìn khác Burn-up Chart tạo động lực dựa trên những thành công đã có trong Sprint; Burn-down Chart tạo động lực dựa trên nỗ lực cần thực hiện để đạt được Sprint Goal.

Burn-down Chart rất hữu dụng cho sự kiện Sprint Retrospective. Việc đạt được Sprint Goal rất quan trọng, nhưng cách thực hiện của nhóm Scrum để đạt được Sprint Goal còn quan trọng hơn; bởi chỉ khi tìm ra quy trình của riêng mình, nhóm Scrum mới đảm bảo có được nhiều sprint hiệu quả và thành công. Kết quả của 2 nhóm Scrum sau là như nhau vì đều đạt được Sprint Goal, tuy nhiên Burn-down Chart có thể cung cấp những thông tin cho thấy nhóm Scrum thứ nhất (có thể) hoạt động khoa học hơn, nhóm lưỡng trước được những công việc có thể xảy ra, ước lượng tốt hơn và không bị áp lực vào giai đoạn cuối.



Hình 3.11: Burn-down Chart của 2 nhóm thực hành Scrum

Definition of Done

Definition of Done (DoD: định nghĩa hoàn thành) là một artifact rất quan trọng trong Scrum và cần được thống nhất cũng như hiểu rõ trong nhóm Scrum.

DoD cho biết chính xác thế nào là một công việc được coi là “hoàn thành”. DoD được coi như một bài kiểm thử, nếu công việc hiện tại đáp ứng được tất cả những yêu cầu trong DoD, công việc được coi là hoàn thành. Ngược lại, chỉ cần một yêu cầu không được đáp ứng, cả công việc được coi là chưa hoàn thành.

DoD cần phải được diễn đạt đủ rõ ràng, không gây hiểu nhầm. Bởi có quá nhiều cách hiểu khác nhau về cách hoàn thành một công việc, DoD cần phải được diễn đạt rõ ràng, dễ hiểu, và bất cứ thành viên nào trong nhóm Scrum đều có thể dựa vào DoD để khẳng định được nhiệm vụ đó đã hoàn thành hay chưa. Ví dụ, DoD cho công việc thiết kế giao diện cho chức năng đăng nhập, nếu là có giao diện đẹp mắt, màu sắc hài hòa là không rõ ràng và không đánh giá được; DoD tốt hơn nên là có 2 textbox cho phép người dùng nhập thông tin đăng nhập, được thiết kế cân đối, thẳng hàng, sử dụng màu đơn sắc theo giao diện chung.

DoD cần được thống nhất trong toàn nhóm Scrum. Sẽ là không khả thi khi một người đặt ra DoD mà nhóm Scrum không thể hiểu, không thể làm theo hoặc không muốn làm theo. Khi đó những công việc sẽ được đánh giá theo cách cảm tính và không đạt được sự minh bạch cần thiết.

DoD có thể khác nhau giữa các công việc. Mỗi công việc cụ thể có thể rất khác nhau, vì thế DoD hoàn toàn có thể khác nhau giữa các công việc. Ví dụ, DoD của công việc thiết kế cơ sở dữ liệu sẽ là có lược đồ thiết kế và script để tạo cơ sở dữ liệu; trong khi, DoD của công việc đánh giá thiết kế cơ sở dữ liệu sẽ là kiểm tra thiết kế có đảm bảo chuẩn 3 hay không.

Nên có DoD chung cho các công việc. Dù những công việc khác nhau, chúng ta vẫn có thể tìm ra được những tiêu chí chung để đánh giá việc hoàn thành. Thông thường, nhóm Scrum sẽ thống nhất một DoD cho các User Story, đây được coi là phần dễ thực hiện nhất. Ví dụ, DoD của mọi User Story, nên được thiết kế dưới dạng một checklist.

- Được lập trình.
- Có unit test.
- Có end-to-end test được thực hiện tự động với ít nhất 3 case cơ bản.

- Không có lỗi
- Người dùng có thể tương tác thông qua giao diện
- Thời gian phản hồi cho mỗi thao tác người dùng dưới 3 giây
- *Thoả mãn DoD (hoặc accept criteria) bổ sung cho User Story*
- *Tất cả mọi công việc phải có DoD.* Một công việc không nên được tiến hành khi không có DoD, bởi những nỗ lực thực hiện công việc đó sẽ hoàn toàn vô ích, vì nhóm Scrum không biết khi nào công việc được hoàn thành.



Hãy thực hiện việc tạo ra DoD cho User Story *Là trưởng bộ phận chăm sóc khách hàng của Facebook, tôi muốn gửi email tới những người dùng cụ thể nếu họ không sử dụng Facebook trong 2 tuần*, theo từng bước sau:

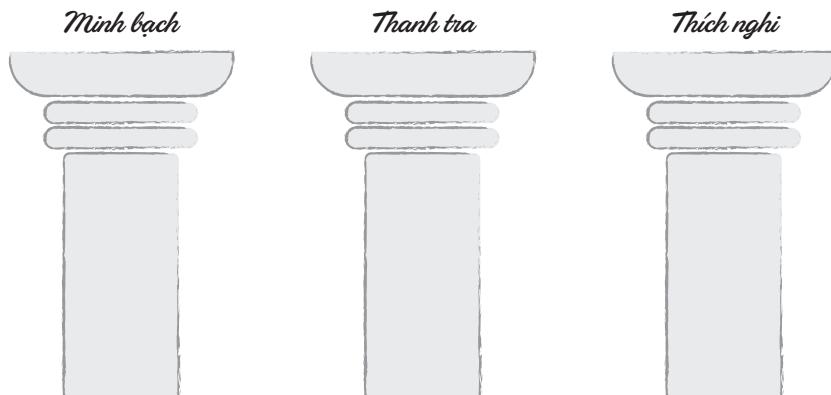
1. Liệt kê tất cả những yêu cầu để có thể triển khai được chức năng này trên hệ thống thật với chất lượng cao nhất. Một số gợi ý như:
 - Lập trình.
 - Kiểm soát lỗi qua logging.
 - Kiểm thử performance.
2. Đặt câu hỏi tại sao yêu cầu này lại cần thiết cho yêu cầu đã tạo ra từ bước 1.
3. Phân chia nhóm những yêu cầu trên. Ví dụ:
 - Lập trình: Viết code, check-in vào source code repository...
 - Kiểm soát chất lượng: Unit test, đảm bảo acceptance criteria...
 - Vận hành: Có phương pháp đo đạc, theo dõi...
4. Sắp xếp lại mức độ quan trọng của mỗi yêu cầu trong từng nhóm.
5. Chọn 2 yêu cầu quan trọng nhất trong mỗi nhóm.
6. Viết lại những yêu cầu trên để tạo thành DoD.

BA TRỤ CỘT CỦA SCRUM

Các phần trên đã mô tả rõ ràng về cách thức một nhóm Scrum thực hành Scrum. Nói chung, Scrum dễ hiểu để làm theo bởi Scrum cung cấp đủ những vai trò, sự kiện, artifact cần thiết để một nhóm phát triển phần mềm có thể thực hành được. Tuy vậy, nhóm Scrum chỉ thực sự làm chủ được Scrum nếu có hiểu biết sâu sắc

những nguyên lý đứng sau làm nền tảng lý thuyết cho Scrum vận hành hiệu quả.

Scrum được xây dựng dựa trên lý thuyết *kiểm soát quá trình thực nghiệm*; lý thuyết này khẳng định rằng kiến thức đến từ những quyết định dựa trên kinh nghiệm. Scrum sử dụng cách tiếp cận bằng những chu kỳ tuần hoàn nhằm tăng thêm kinh nghiệm, do đó tối ưu hóa khả năng dự báo và kiểm soát rủi ro, gọi là Sprint. Cách tiếp cận này dựa trên ba yếu tố của việc kiểm soát quá trình thực nghiệm: *minh bạch, thanh tra và thích nghi*; thường được gọi là *3 trụ cột của Scrum*.



Hình 3.12: Ba trụ cột của Scrum

Minh bạch

Minh bạch là tất cả mọi thông tin cần thiết cho việc hoàn thiện công việc đều được cung cấp đầy đủ, chính xác và nhất quán cho các bên liên quan.

Tại sao cần *minh bạch*? Vì quyết định được đưa ra dựa trên những thông tin có được, khi càng ít thông tin quyết định càng thiếu sáng suốt. Ví dụ, những kiến trúc sư phần mềm có thể đưa ra những kiến trúc tối ưu về mặt lý thuyết nhưng lại hoàn toàn không thể triển khai do đội lập trình viên không đáp ứng đủ khả năng, khiến phần mềm luôn tồn tại “trên giấy”. Nếu những kiến trúc sư phần mềm biết rõ khả năng, kiến thức của lập trình viên, họ có thể đưa ra một kiến trúc không tối ưu nhưng lại khả thi để có phần mềm chạy tốt.

Scrum minh bạch những gì? Mọi thứ. Cho những ai? Mọi thành viên.

Tất cả những artifact cũng như sự kiện của Scrum đều thúc đẩy việc minh bạch:

- Product Backlog minh bạch những công việc cần hoàn thiện để release sản phẩm và tiến độ công việc thông qua Release Burn-down chart;
- Sprint Backlog minh bạch những công việc cần hoàn thiện để đạt được Sprint Goal, tiến độ công việc thông qua Sprint Burn-down chart; những trở ngại thông qua Daily Scrum; kết quả thông qua Sprint Review;
- DoD minh bạch việc kiểm tra, đánh giá một công việc;
- Sprint Retrospective minh bạch những vấn đề gặp phải trong nhóm Scrum và giải pháp giúp nhóm tiến bộ hơn.

Mặc dù mỗi artifact cũng như sự kiện đều được chỉ định những vai trò chịu trách nhiệm tương ứng, nhưng mọi thông tin cần thiết cho việc hoàn thành sản phẩm đều được minh bạch cho mọi thành viên trong nhóm Scrum.

Thanh tra

Thanh tra là nhìn nhận những gì đã và đang xảy ra, đánh giá bằng cách so sánh với kỳ vọng.

Tại sao cần thanh tra? Để phát hiện những vấn đề và đưa ra cách giải quyết nhằm đạt được kỳ vọng.

Scrum thanh tra những gì? Mọi công việc. Với tần suất nào? Tối đa là 24 giờ.

Scrum thúc đẩy việc thanh tra thông qua các sự kiện và artifact:

- Sprint Backlog cho biết tình hình hiện tại, thậm chí với tần suất từng giờ; Sprint Burn-down Chart cho biết tương quan giữa tiến độ hiện tại và Sprint Goal (kỳ vọng);
- Daily Scrum cho biết tình hình hiện tại;
- Sprint Retrospective nhìn nhận những gì đã xảy ra, so sánh với kỳ vọng.

Thích nghi

Thích nghi là điều chỉnh hành vi, phương pháp để phù hợp với tình hình thực tế.

Tại sao cần thích nghi? Vì ngữ cảnh luôn thay đổi, chỉ việc thích nghi mới mang lại

hiệu quả cao nhất.

Thanh tra và thích nghi luôn phải song hành bởi thích nghi là hệ quả từ việc cần điều chỉnh hành vi, phương pháp cho phù hợp khi tình hình hiện tại không đáp ứng đúng kỳ vọng sau quá trình thanh tra. Nếu hoạt động thanh tra không tiến tới giải pháp thích nghi, nói chung không mang lại hiệu quả (và lãng phí).

Scrum thúc đẩy việc thích nghi thông qua các vai trò, sự kiện và artifact:

- Sprint Backlog được cập nhật tức thời, cùng những công việc mới nhằm đạt được Sprint Goal;
- Daily Scrum lên kế hoạch cho 24 giờ tiếp theo để phù hợp với tình hình hiện tại nhằm hướng tới Sprint Goal;
- Sprint Retrospective thúc đẩy cách nhóm Scrum làm việc để phù hợp với tình hình hiện tại nhằm đưa ra cách làm tốt nhất.

Việc duy trì 3 trụ cột này rất quan trọng. Một nhóm Scrum không thể hoạt động hiệu quả nếu một trong ba trụ cột này không “vững”. Khi đã làm chủ được Scrum, bạn sẽ hiểu chính xác tại sao Scrum lại cấu trúc những vai trò, sự kiện và Artifact như vậy; đó là tối thiểu những gì cần có để một nhóm làm việc hiệu quả; giảm bớt sẽ gây ra xáo trộn, thêm nữa sẽ gây lãng phí.

Ví dụ, nhóm phát triển là trung tâm của hoạt động phát triển phần mềm, nhưng họ sẽ không biết cách tạo ra một sản phẩm giá trị nếu thiếu Product Owner. Nhưng Product Owner thì thường không hiểu về quá trình phát triển phần mềm và luôn muốn mọi công việc được hoàn thành tức thì, khiến Sprint Goal có thể liên tục bị thay đổi; do đó ScrumMaster là người giúp công việc được thực hiện khoa học hơn.

NĂM GIÁ TRỊ CỐT LÕI SCRUM

Tất cả các công việc thực hiện trong Scrum cần một tập hợp các giá trị làm nền tảng cho quy trình và sự cộng tác của nhóm. Chính điều này khiến chúng ta khó làm chủ Scrum, bởi không chắc chắn mọi thành viên trong nhóm Scrum đều đáp ứng được tập hợp những giá trị này. Nếu 5 giá trị sau đây được thoả mãn, việc áp dụng Scrum sẽ thành công. Tuy vậy, Scrum luôn tạo không gian cho việc phát triển của nhóm, chúng ta không nên quá lo lắng nếu 5 giá trị này không được thoả mãn ngay khi nhóm Scrum đi vào hoạt động. Điều quan trọng là nhóm cần hướng tới 5 giá trị này để đạt hiệu quả cao nhất.



Hình 3.13: Các giá trị của Scrum

Dũng cảm

Đây là giá trị cao nhất của Scrum và giá trị này cần có mặt ở mọi nơi để đảm bảo Scrum thành công.

Một tổ chức khi triển khai Scrum cần lòng dũng cảm. Bởi theo kinh nghiệm của tôi, 99% nhóm Scrum sẽ thất bại với Sprint đầu tiên, 1% còn lại nghĩ rằng mình đã thành công. Rất nhiều nhóm Scrum sẽ thất bại (không đạt được Sprint Goal) trong 2 đến 3 Sprint, thậm chí trong dự án đầu tiên dù nhóm có những chuyên gia giỏi nhất. Những con số về tỉ lệ dự án thành công với Scrum không cho thấy điều nghiệt ngã này. Bởi việc thay đổi đột ngột cách làm việc chắc chắn không có hiệu quả trong thời gian ngắn. Tuy vậy, Scrum được xây dựng để thúc đẩy việc cải thiện giúp tìm ra phương pháp tối ưu cho nhóm, nên các tổ chức cần dũng cảm chấp nhận thất bại khi lần đầu triển khai Scrum. Chúng ta sẽ bàn về vấn đề này nhiều hơn trong những phần sau.

Khách hàng của nhóm Scrum cần lòng dũng cảm. Tương tự với tổ chức triển khai Scrum, khách hàng của nhóm Scrum cần chấp nhận chi phí cho những thất bại đầu tiên để đón nhận những phần có giá trị được chuyển giao thường xuyên hơn sau này.

Product Owner cần lòng dũng cảm. Bởi đôi khi họ sẽ gặp khó khăn trong việc nhóm phát triển không thể chuyển giao được phần tăng trưởng như đã cam kết. Khách hàng sẽ rất không hài lòng. Product Owner cần sự dũng cảm để đối mặt với thực tại, cung cấp những thông tin cần thiết cho các bên liên quan hiểu lý do đứng sau kết quả nhằm thúc đẩy một thoả thuận mới.

ScrumMaster cần lòng dũng cảm. Họ cần tin tưởng và bảo vệ nhóm phát triển trước những áp lực thay đổi đến từ khách hàng, những trở ngại có thể ảnh hưởng tới Sprint Goal.

Nhóm phát triển cần sự dũng cảm. Nhiều quyền hạn hơn đồng nghĩa với nhiều trách nhiệm hơn; và họ phải dũng cảm đón nhận trách nhiệm. Họ cần dũng cảm loại bỏ những yêu cầu vượt quá khả năng trong Sprint; cần dũng cảm để tự tổ chức; cần dũng cảm để đưa ra giải pháp; cần dũng cảm nhìn nhận vấn đề và thay đổi trong Daily Scrum và Sprint Retrospective.

Nói chung, lòng can đảm cần ở mọi thành viên và các bên liên quan để đảm bảo Scrum có cơ hội đi đến thành công.

Cởi mở

Bởi 1 trong 3 trụ cột của Scrum là minh bạch, nên sự cởi mở là giá trị rất quan trọng.

Scrum thúc đẩy sự minh bạch thông qua việc chia sẻ cởi mở những thông tin cần thiết để đạt được mục tiêu đề ra.

Các cá nhân được khuyến khích cộng tác với nhau để hoàn thành công việc. Những xung đột tích cực được chấp nhận nhằm giúp mọi thành viên hiểu nhau và đưa ra cách làm việc hiệu quả nhất. Cá nhân trong nhóm Scrum cần cởi mở trong việc chia sẻ những khó khăn gặp phải trong Daily Scrum và Sprint Retrospective để nhóm có thể trợ giúp hoặc điều chỉnh nhằm thích nghi với tình hình hiện tại.

Giá trị này cũng có nghĩa là cá nhân nên lắng nghe ý kiến từ những cá nhân khác.

Cam kết

Scrum hoạt động hiệu quả khi các thành phần liên quan *thực hiện đúng cam kết*.

Tổ chức khi triển khai Scrum cần cam kết trao quyền tự tổ chức cho nhóm Scrum. ScrumMaster cam kết dẫn dắt nhóm thực hiện đúng Scrum và tối đa hiệu quả làm việc của nhóm phát triển thông qua việc trợ giúp nhóm vượt qua những trở ngại.

Chủ đầu tư sản phẩm cam kết trao quyền định hướng giá trị sản phẩm cho Product Owner. Product Owner cam kết duy trì Product Backlog để hướng tới release sản phẩm và giúp nhóm phát triển hiểu rõ yêu cầu.

Nhóm phát triển cam kết tối đa hoá công việc có thể thực hiện được trong Sprint với việc hoàn thành từng công việc dựa trên DoD.

Cam kết là cách tốt nhất để đạt được thành công trong mọi công việc, dự án Scrum cũng vậy.

Tập trung

Giá trị này có liên quan chặt chẽ tới giá trị *cam kết*.

Nhóm Scrum chỉ có thể thực hiện được cam kết nếu họ chỉ tập trung cho Sprint Goal hiện tại và không bị xao nhãng bởi những công việc khác. Nhóm sẽ hoàn thành User Story X vào cuối Sprint đồng nghĩa với việc nhóm phát triển giả định rằng mình chỉ tập trung cho User Story X, nếu họ có một công việc khác nữa (và không thể ước lượng) thì với lòng tự trọng của mình, nhóm sẽ không bao giờ đưa ra cam kết như vậy.

Ngược lại, tổ chức khi triển khai Scrum cần cam kết duy trì sự tập trung cho nhóm, ít nhất trong Sprint để đạt được Sprint Goal.

Tôn trọng

Đây là giá trị cuối cùng, nhưng luôn là một phần trong những giá trị khác và định hướng trong mọi hành vi, hoạt động.

Bởi Scrum thúc đẩy sự trao đổi, cộng tác, nên các xung đột sẽ xảy ra thường xuyên; tôn trọng là giá trị cần được ghi nhớ trong mọi hành vi.

ScrumMaster cần tôn trọng cách thức vận hành của nhóm, lắng nghe những thông tin từ những thành viên trong nhóm và thúc đẩy quá trình tự cải tiến quy trình, hoàn toàn không phải sự ép buộc.

Các thành viên trong nhóm khi đưa ra quyết định, cần thể hiện sự tôn trọng với ý kiến và cá nhân khác. Họ tôn trọng sự thành công, nhận trách nhiệm cho sự thất bại và tích cực tìm ra phương án giải quyết, không quy trách nhiệm và hướng chỉ trích vào cá nhân.

Tôn trọng cũng có nghĩa là các cá nhân có mặt trong các sự kiện đúng giờ, tập trung hoàn thành công việc với hiệu suất và chất lượng cao nhất.

Tôn trọng là điều kiện cần để có một nhóm Scrum tốt, thúc đẩy mọi người gắn kết và muốn làm việc cùng nhau.



Làm thế nào để đánh giá được nhóm Scrum có thoả mãn 5 giá trị trên hay không? Và ở mức độ nào?

SCRUMBUT

ScrumBUT là một khái niệm rất đáng lưu tâm với những nhóm thực hành Scrum. Tôi đã tiếp xúc với rất nhiều nhóm Scrum không đạt được hiệu quả cao bởi tư tưởng *Scrum và Agile nói chung chào đón những thay đổi, bản thân Scrum cũng là một framework nên việc thay đổi Scrum cho phù hợp là đương nhiên*. Ý kiến đó không sai, bởi Scrum cung cấp cách thức giúp nhóm tìm ra quy trình hợp lý cho chính mình; song việc chỉnh sửa Scrum khi nhóm chưa thực sự làm chủ Scrum thường là một quyết định thiếu sáng suốt. Scrum đã rất tối giản, các tổ chức khi triển khai Scrum nên làm theo Scrum Guide để có sự ổn định trước khi đưa ra những điều chỉnh, bởi những điều chỉnh chỉ đạt hiệu quả tốt khi chúng ta thực sự làm chủ bằng cách thực hành đúng để hiểu Scrum có phù hợp hay không.

Khái niệm ScrumBUT để chỉ việc thực hành Scrum không theo Scrum tiêu chuẩn:

We use Scrum BUT we don't have Daily Scrum – chúng tôi thực hành Scrum nhưng không thực hiện Daily Scrum.

We use Scrum BUT we do Sprint Retrospective for 5 hours – chúng tôi thực hành Scrum nhưng sử dụng 5 giờ cho sự kiện Sprint Retrospective.

ScrumBUT đương nhiên không được khuyến nghị từ các tổ chức đào tạo Scrum. Nhưng theo tôi, việc thực hành Scrum hay ScrumBUT không quan trọng, nếu việc làm đó mang lại hiệu quả cao. Điều quan trọng là: *Nhóm có nhận ra mình đang thực hành Scrum hay ScrumBUT không? Nhóm thực hành ScrumBUT vì không thể thực hành đúng Scrum hay cần sự điều chỉnh? Có lý do, bằng chứng nào cho thấy việc thực hành ScrumBUT cho hiệu quả cao hơn không?* Nếu 3 câu hỏi đó được trả lời thỏa đáng, việc thực hành ScrumBUT sẽ cho hiệu quả cao và nên được khuyến khích. Tuy vậy, việc thực hành ScrumBUT nếu thiếu những lý thuyết thường là cạm bẫy, hãy cẩn thận.

Nhưng nếu có quá nhiều BUT, nhóm cần biết rằng mình đã không còn thực hành ScrumBUT nữa, mà đang đi theo 1 quy trình hẫu như không còn liên quan tới

Scrum. Lúc đó nhóm cần một tên gọi khác cho quy trình của mình, đừng sử dụng Scrum. Nhưng nếu vậy, nhóm cần Scrum để làm gì? Chỉ có 2 trường hợp dẫn tới kết quả này: hoặc nhóm đã có hiểu biết và thực hành rất cao để đưa ra một phương pháp khác (rất ít); hoặc nhóm đang trong giai đoạn *chưa hiểu gì về Scrum*.

Thậm chí nhiều người gặp tôi và khẳng định “nhóm của tôi đã thực hành Scrum vì chúng tôi có *standup meeting hàng ngày*”. “Không, bạn thậm chí còn không thực hành Scrum BUT, bạn chỉ thực hành 1 trong nhiều sự kiện của Scrum mà thôi.”



Để thực hành đúng Scrum, nhóm Scrum luôn cần xem lại *Scrum Guide* (nhiều lần trong một năm) để hiểu hơn và đánh giá lại cách thực hiện của mình. Với những nhóm mới thực hành Scrum và không có *ScrumMaster* giỏi, tôi khuyến nghị nhóm nên xem lại *Scrum Guide* sau một hoặc một vài Sprint.

Nhóm nên áp dụng đúng Scrum tiêu chuẩn trước khi nghĩ đến việc thay đổi; và việc thay đổi, nếu có, luôn cần bắt đầu từ câu hỏi tại sao và có mang lại giá trị hơn không?

HIỂU ĐÚNG

Nhóm liên chức năng gồm những thành viên có khả năng làm mọi công việc độc lập.

Đây là sai lầm phổ biến khi hiểu Nhóm phát triển gồm những thành viên có thể làm được mọi công việc như lập trình, kiểm thử... độc lập. Thực tế, nhóm liên chức năng là nhóm có đầy đủ kỹ năng để hoàn thành được mục tiêu đề ra; mỗi thành viên có những kỹ năng riêng biệt, do đó việc cộng tác để phát huy tối đa kỹ năng của từng thành viên để đạt được mục tiêu chung là rất quan trọng.

Daily Scrum không cần thực hiện hàng ngày.

Thật đáng tiếc, đây là câu hỏi rất hay gặp đặc biệt ở những nhóm thực hành Scrum BUT. Nếu nhóm của bạn không thực hiện hàng ngày, hãy chọn một tên khác, hãy để cái tên mô tả đúng sự kiện.

Các thành viên trong nhóm phát triển báo cáo với ScrumMaster trong Daily Scrum.

ScrumMaster không phải phải người quản lý nhóm hay quản lý dự án. ScrumMaster không chịu trách nhiệm cho sự thành công của Sprint, chính Nhóm phát triển chịu trách nhiệm cho Sprint Goal thông qua cam kết của nhóm. Các thành viên trong nhóm phát triển báo cáo công việc cho cả nhóm trong Daily Scrum theo cơ chế thanh tra – thích nghi nhằm hướng tới Sprint Goal, không phải cơ chế tập trung thông tin.

Scrum chào đón mọi thay đổi trong yêu cầu tức thì.

Agile chào đón mọi thay đổi trong yêu cầu, và cố gắng phản ứng kịp thời với những thay đổi đó nhưng *tức thì* hay không là cách giải quyết của từng phương pháp cụ thể. Scrum không chào đón bất cứ sự thay đổi nào trong yêu cầu khiến Sprint Goal bị thay đổi. Mọi thay đổi sẽ được Product Owner ghi nhận và xem xét vào Sprint tiếp theo, giúp nhóm phát triển tập trung vào Sprint Goal hiện tại.

TỔNG KẾT

Scrum là một phương pháp phát triển phần mềm theo nhóm nhằm cung cấp giá trị cho việc phát triển phần mềm. Các thành viên trong nhóm làm việc cùng nhau để đạt được một mục tiêu chung với quy trình và cách làm cụ thể bởi từng công cụ do chính những thành viên trong nhóm thiết lập với sự đồng thuận tối đa, nhằm tạo ra giá trị cao nhất.

3 trụ cột của Scrum: *minh bạch, thanh tra, thích nghi*.

5 giá trị của Scrum: *dũng cảm, cởi mở, cam kết, tập trung, tôn trọng*.

3 vai trò: *Product Owner, ScrumMaster, nhóm phát triển*.

Artifact: *Product Backlog, Sprint Backlog, Release Burn-down Chart, Sprint Burn-down Chart, Definition of Done*

Thông thường, Scrum có thể được mô tả trong 1 trang A4, tôi khuyến nghị bạn nên để hình ảnh này ở mọi nơi nhằm giúp những nhóm mới thực hành Scrum hiểu hơn về phương pháp này.

DA

KANBAN &
SCRUMBAN

Scrum là một phương pháp tuyệt vời trong phát triển sản phẩm phần mềm. Mặc dù Scrum hoàn toàn tuân theo Tuyên ngôn Agile, đôi khi Scrum được coi là quá **cứng nhắc** trong việc phản hồi với những thay đổi trong yêu cầu. Điểm mấu chốt ở đây là, Scrum vận hành theo Sprint, và khi Sprint Goal đã được xác định, nhóm phát triển sẽ không bao giờ muốn thoả hiệp với những thay đổi trong yêu cầu hay những công việc phát sinh làm thay đổi Sprint Goal; bởi vậy *tập trung* là một trong 5 giá trị cốt lõi của Scrum.

Việc **cứng nhắc** dù trong thời gian một Sprint rất ngắn nhưng trong nhiều hoàn cảnh vẫn chưa đủ tốt, đặc biệt trong ngành dịch vụ hoặc trong việc nâng cấp, bảo trì hệ thống. Ngay khi khách hàng thông báo rằng họ không thể thực hiện được việc thanh toán bằng thẻ ngân hàng, khách hàng muốn chúng ta có phản ứng và sửa chữa kịp thời, nếu ScrumMaster nói với Product Owner rằng "*anh hãy dừng việc yêu cầu nhóm phát triển thực hiện sửa đổi chức năng đó, nó không nằm trong Sprint Goal của Sprint này*", điều đó nghe thật nực cười.

Vì vậy, nhiều tổ chức giờ đây lựa chọn Kanban, phương pháp tốt hơn nhằm phản ứng nhanh (gần như tức thì) hơn với những thay đổi trong yêu cầu.

Hầu hết với những người mới tìm hiểu, Kanban đơn giản chỉ là Scrum nhưng không chia theo Sprint. Nhưng chúng ta cần nhớ rằng, trái tim của Scrum là Sprint, nếu một phương pháp không hướng sự tập trung vào Sprint thì phương pháp đó không có gì giống Scrum.

TODO	Nghiên cứu Công nghệ	Lập trình	Đánh giá Mã nguồn	Kiểm thử	Tích hợp	DONE
		<p>Thanh toán qua thẻ tín dụng</p> <p>Thanh toán qua Paypal</p>				Thanh toán qua Apple Pay

Hình 4.1: Bảng Kanban

Tư tưởng Kanban xuất phát sớm hơn rất nhiều so với Scrum. Trong những năm 1940, Toyota tìm cách tối ưu quy trình của mình bằng cách tối thiểu số sản phẩm đủ để đáp ứng nhu cầu người dùng trong thực tế giúp hạn chế vật liệu tồn kho với những giả định trong một tương lai không quá dài. Và một phương pháp được đưa ra với cách tiếp cận *lên kế hoạch tức thì* (*just-in-time planning*) với những hiểu biết tại thời điểm hiện tại. Sau này Kanban cũng được áp dụng trong việc phát triển phần mềm bởi những đặc tính tương tự: *lên kế hoạch tức thì để phản ứng với những thay đổi nhanh, nhằm tối thiểu những công việc cần thiết để hoàn thành sản phẩm, qua đó giảm thiểu những lãng phí không cần thiết*. Do vậy, Kanban là một phương pháp theo tư tưởng Lean.

Việc mô tả cách thức một nhóm phát triển phần mềm thực hành Kanban khá đơn giản:

- Nhóm định nghĩa một quy trình để thực hiện công việc. Ví dụ cách thực hiện một User Story thường là: *Nghiên cứu công nghệ, lập trình, đánh giá mã nguồn, kiểm thử, tích hợp*. Quy trình này được gọi là *workflow* (*luồng công việc*), được trực quan hoá thông qua các cột trong bảng, được gọi là *các trạng thái (state)* công việc;
- Bất cứ một công việc nào cần làm đều được thêm vào danh sách *TO DO* (*cần làm*), ngay lập tức danh sách này được sắp xếp lại theo thứ tự ưu tiên;
- Các công việc được lần lượt chuyển qua các cột tương ứng với trạng thái công việc đó đang diễn ra, đến khi được hoàn thành. Khi đó, công việc sẽ nằm ở cột *DONE* (*hoàn thành*).

NHỮNG GIÁ TRỊ CỦA KANBAN

Linh hoạt trong việc lên kế hoạch

Nhóm thực hành Kanban luôn tập trung vào những công việc đang được thực hiện và danh sách công việc còn tồn đọng. Bất cứ khi nào một công việc mới được phát hiện hoặc đề xuất, công việc đó ngay lập tức được đưa vào danh sách *TO DO* với độ ưu tiên xác định.

Khác với nhóm thực hành Scrum bị giới hạn trong Sprint Goal, nhóm thực hành Kanban có một mục tiêu chung là hoàn thành sớm nhất những công việc còn tồn đọng.

Khác với nhóm thực hành Scrum bị giới hạn công việc trong Sprint Backlog, nhóm thực hành Kanban chỉ có một Backlog duy nhất.

Trực quan hoá trạng thái công việc

Ưu điểm lớn nhất của Kanban chính là trực quan hoá. Kanban giúp trực quan hoá 3 thứ:

Luồng công việc. Luồng công việc là các bước phải thực hiện để hoàn thành một công việc. Trong ví dụ trên, luồng công việc để thực hiện một User Story bao gồm: *nghiên cứu công nghệ, lập trình, đánh giá mã nguồn, kiểm thử, tích hợp.* Bất cứ một công việc nào cũng cần trải qua những trạng thái này một cách tuần tự, tương ứng với tiến độ thực hiện công việc đó. Bằng cách trực quan hoá luồng công việc này, nhóm phát triển sẽ biết ghi nhớ chính xác và luôn đảm bảo luồng công việc đã đặt ra. Ví dụ, mọi thành viên trong nhóm phát triển đều biết rằng, một mã nguồn sau khi được lập trình, không được phép kiểm thử trừ khi công việc đó được hoàn thành ở bước tiếp theo là đánh giá mã nguồn.

Danh sách công việc. Mọi công việc đều được đặt trong danh sách *TO DO*. Tại bất cứ thời điểm nào, nhóm phát triển đều biết khối lượng công việc còn lại và thứ tự ưu tiên giữa chúng.

Trạng thái công việc. Trạng thái mọi công việc và cả hệ thống luôn được thể hiện và biết rõ tại bất cứ thời điểm nào. Chức năng *thanh toán qua thẻ tín dụng* hiện đang được lập trình, chức năng *thanh toán qua Paypal* đang trong quá trình kiểm thử, chức năng *thanh toán qua Apple Pay* đã thực hiện xong.

Cải tiến quy trình

Bằng cách trực quan hoá quy trình và trạng thái công việc của cả hệ thống, chúng ta có cơ hội cải tiến quy trình vì dễ dàng nhận ra những vấn đề trong quy trình hiện có.

Trong ví dụ trên, 5 công việc đang ở trạng thái đánh giá mã nguồn, trong khi những trạng thái khác chỉ có 2 hoặc 3 công việc. Điều này có thể dẫn đến một giả định rằng, nhóm phát triển đang không đánh giá cao công việc này, đồng thời khi những công việc này được hoàn thành cùng lúc, nhóm sẽ có từ 5 đến 7 chức năng cần kiểm thử, điều này có thể khiến việc kiểm thử bị quá tải.

Giới hạn công việc đang thực hiện

Công việc đang thực hiện (WIP: Work-In-Progress) theo nguyên lý Lean được coi là sự lãng phí. Cũng với ví dụ trên, khi 5 đến 7 chức năng cần kiểm thử trước khi được tích hợp để được coi là *DONE (hoàn thành)*; đây là một tín hiệu tốt rằng chúng

ta “sắp” có 7 chức năng mới nhưng cũng là một tín hiệu rất tệ rằng *có thể không* chức năng nào được hoàn thành, bởi thực tế là các chức năng này vẫn chưa hoàn thành. Nếu chúng ta giới hạn số công việc đang được thực hiện xuống 3, Nhóm phát triển sẽ nỗ lực hoàn thành từng chức năng trước khi những chức năng mới được bắt đầu, điều này đảm bảo các chức năng này *hoàn thành* theo đúng nghĩa.



Tại sao những công việc đang thực hiện lại được coi là *lãng phí*?



Bạn có thể tạo ra một bảng Kanban đơn giản theo phương pháp sau:

1. Phân loại các công việc theo đầu vào, ví dụ

- User Story
- Bug
- ...

2. Định nghĩa workflow chung cho mọi loại công việc ở trên.

3. Trong trường hợp tốt nhất, mọi loại công việc ở trên có chung workflow. Nếu không, hãy trả lời câu hỏi sau với những sự khác biệt của workflow:

- Có thể sử dụng một trạng thái chung cho những sự khác biệt này không? Ví dụ sử dụng Doing cho trạng thái Coding của User Story hay Fixing của Bug.
- Có thể rút ngắn workflow mà không ảnh hưởng tới việc cộng tác không? Ví dụ gộp trạng thái Technical Analysis và Coding thành một trạng thái là Implement.

4. Đảm bảo mỗi workflow chỉ chứa tối đa 5 trạng thái cần xử lý quan trọng nhất.

5. Nếu các workflow cho từng loại công việc có thể sử dụng chung một workflow, 1 bảng Kanban sẽ được hình thành; ngược lại chúng ta cần tạo nhiều bảng Kanban, mỗi bảng Kanban cho một loại công việc:

- Cột đầu tiên là TO DO;
- Các cột tiếp theo là trạng thái của workflow đã xây dựng ở trên theo thứ tự từ trái qua phải;
- Cột cuối cùng là DONE.

KANBAN KHÔNG PHẢI LÀ AGILE

Hiện giờ đa số mọi người vẫn coi Kanban là một phương pháp nằm trong tập hợp những phương pháp Agile. Nhưng vài năm trước, David Anderson đã đưa ra một lý lẽ (mà theo tôi là thuyết phục) để Kanban không còn nằm dưới chiếc ô Agile. Mặc dù việc này có thể nảy sinh câu hỏi: *Kanban là một phương pháp theo tư tưởng Lean, và Lean Software Development thì nằm trong chiếc ô Agile.*

Lý do chính là Agile là tập hợp những phương pháp nhằm giúp việc phát triển phần mềm tốt hơn bởi việc linh hoạt hơn. Tuy vậy, Kanban không giúp gì cho việc này. Tôi xin khẳng định là không. Kanban không phải là một phương pháp giúp cho việc phát triển phần mềm linh hoạt hơn, việc phát triển phần mềm vẫn giữ nguyên, dù có sử dụng Kanban hay không; Kanban đơn giản chỉ giúp chúng ta trực quan hóa luồng công việc hiện có, tìm ra những điểm có thể cải tiến; các nhóm phát triển phần mềm hoàn toàn không bị ảnh hưởng bởi Kanban. Hay nói đúng ra, Kanban không giúp cho việc phát triển phần mềm linh hoạt hơn. Kanban chỉ đơn giản như một kaizen, hoặc trợ giúp việc tạo ra nhiều kaizen trong việc phát triển phần mềm của nhóm mà thôi.

Nếu bạn còn nghi ngờ, hãy so sánh với Scrum để hiểu rõ hơn.

Scrum chỉ rõ: sau mỗi Sprint, ít nhất một phần tăng trưởng cần phải được chuyển giao; phần tăng trưởng này phải là một chức năng được thao tác bởi người dùng; và trong một Sprint, nhóm Scrum hoàn toàn tự chủ để đạt được Sprint Goal, với những vai trò rõ ràng. Kanban thì không như vậy, nhóm phát triển hoàn toàn có thể triển khai theo cách phát triển phần mềm truyền thống, với những trạng thái theo dòng chảy của phương pháp thác nước. Tất nhiên, nhiều nhóm thực hành Kanban có thể chọn cách chuyển giao từng phần tăng trưởng này; nhưng công bằng mà nói, đó là do nhóm phát triển chọn cách làm như vậy, Kanban hoàn toàn không quy định cũng như hỗ trợ điều này.

Chính vì vậy, thực tế là không nhóm phát triển phần mềm nào ngay lập tức sử dụng Kanban vì thực hành Kanban chỉ giúp cải thiện, không thể thay đổi tình hình. Những nhóm phát triển phần mềm thường bắt đầu bằng việc thực hành Scrum, và khi nhận thấy việc phát triển theo Sprint không phù hợp với hoàn cảnh hoặc khi muốn mở rộng nhóm để tập trung vào luồng công việc, nhóm sẽ chuyển sang thực hành Kanban kết hợp với Scrum như một sự cải tiến. Vậy nên, rất nhiều người khi nhìn và thực hành theo những nhóm thực hành Kanban, hiểu sai rằng Kanban giống với Scrum, chỉ không có Sprint. Điều này hoàn toàn sai lầm, thực tế Kanban trong việc phát triển phần mềm chỉ như những gì tôi đã mô tả ở trên.

SCRUMBAN

Tư tưởng của Kanban hoàn toàn có thể áp dụng cho bất cứ một lĩnh vực nào trong đời sống cũng như ngành công nghiệp; và việc phát triển phần mềm cũng không phải là ngoại lệ.

Thực tế thì Scrum và Kanban đều có những điểm mạnh và điểm yếu riêng trong việc phát triển phần mềm. Vì thế, một xu hướng hiện nay là các nhóm thực hành Scrum chuyển dịch sang thực hành Kanban bằng một phương pháp gọi là *Scrumban (Scrum và Kanban)*.

Hãy cùng làm một so sánh nhanh giữa Scrum và Kanban

	Scrum	Kanban
Thời lượng (Iteration)	Thời lượng của mỗi Sprint là cố định và được xác định trước.	Không giới hạn thời lượng.
Cam kết	Nhóm phát triển cam kết phần tăng trưởng và khối lượng công việc theo Sprint.	Không cam kết.
Đo lường	Tốc độ (velocity) của nhóm phát triển là yếu tố chính.	Thời gian thực hiện (lead time).
Vai trò	Có 3 vai trò: Product Owner, Scrum Master và nhóm phát triển.	Không đề cập.
Nhóm phát triển	Nhóm tự tổ chức và liên chức năng.	Không đề cập. Nhóm có thể được chia theo chức năng.
Burndown Chart	Để theo dõi tiến độ đạt được Sprint Goal, Release Goal	Không đề cập.
WIP	Giới hạn trong Sprint.	Giới hạn trong từng trạng thái.
Ước lượng	Bắt buộc (để tính được lượng công việc cam kết và Sprint Goal).	Không đề cập.
Board	Nhóm phát triển sở hữu.	Chia sẻ giữa một hoặc nhiều nhóm.
Backlog	Product Backlog được quản lý bởi Product Owner. Sprint Backlog được tạo ra theo từng Sprint.	Chỉ có một Backlog.
Sprint Backlog	Nhóm phát triển sở hữu.	Chia sẻ giữa một hoặc nhiều nhóm.
Sự kiện	Có 4 sự kiện: Daily Scrum, Scrum Planning, Scrum Review, Scrum Retrospective.	Không đề cập.

Hầu hết những yếu tố của Scrum rất tốt cho việc phát triển sản phẩm và quản lý. Tuy vậy, *nhóm tự tổ chức* và *liên chức năng* lại có thể là một vấn đề, thường ở 3 điểm:

- *Nhóm nhỏ*: Bởi chú trọng vào sự cộng tác, nhóm Scrum không được phép quá lớn. Một nhóm Scrum được khuyến nghị từ 3 – 9 thành viên. Việc thực hiện lượng công việc lớn trong thời gian ngắn là không khả thi.
- *Nhóm tự tổ chức*: Đây là một ý tưởng rất hay về mặt trao quyền; tuy vậy, để nhóm có thể thực sự tự tổ chức được với sự hỗ trợ của ScrumMaster cần thời gian, và không có hiệu quả tức thì. Nếu bạn có thời gian, nên kiên nhẫn bởi đây là cách tốt nhất; ngược lại, đây sẽ là vấn đề.
- *Nhóm liên chức năng*: Ý tưởng này cũng rất hay với nhóm nhỏ, bởi bằng việc cộng tác, nhóm phát triển sẽ thực hiện công việc rất nhanh, năng suất hơn hẳn những nhóm phân rã theo chức năng.

Đến đây bạn có thể hiểu vì sao Scrum chỉ khuyến nghị thực hành với nhóm nhỏ. Việc mở rộng quy mô nhóm sẽ không đảm bảo được sự cộng tác cho việc *tự tổ chức* và *liên chức năng*. Nhưng nếu chúng ta bắt buộc phải mở rộng nhóm để thực hiện nhiều công việc hơn? Scrumban có thể là một chìa khoá.

Scrumban = Scrum + Kanban

Scrumban là một phương pháp kết hợp giữa Scrum và Kanban nhằm đạt được hiệu quả cao bằng cách tận dụng tối đa lợi thế của cả 2 phương pháp này:

- sử dụng Scrum như một phương pháp, quy tắc trong phát triển phần mềm (bởi Kanban không định nghĩa việc này);
- sử dụng Kanban như một phương pháp trong cải tiến quy trình của nhóm phát triển (bởi Scrum không chỉ định rõ việc này).

Tức là: nhóm phát triển vẫn thực hiện theo cách vận dụng Scrum nhưng:

- không cam kết với Sprint Goal;
- chỉ ước lượng với những công việc hiện có trong Backlog khi bắt đầu một Sprint. Thậm chí có thể bỏ qua việc ước lượng;
- chấp nhận mọi công việc được thêm vào tức thì; (chính điều này khiến Sprint Goal là không cố định, và không thể cam kết);
- định hình rõ workflow;
- chấp nhận nhóm tồn tại các nhóm nhỏ phân rã theo chức năng.

Chú ý là, với quá nhiều *nhưng*, Scrumban không phải ScrumBUT.

Cách làm này mang lại cho Scrumban những lợi thế sau:

- *Chất lượng:* Sẽ là không đúng để khẳng định rằng nhóm thực hành Scrumban cho chất lượng sản phẩm cao hơn nhóm thực hành Scrum. Nhưng bằng cách thể hiện rõ và cải tiến workflow, cách thức thực hiện công việc trong nhóm sẽ rõ ràng hơn khiến việc quản lý chất lượng tốt hơn.
- *Tức thì:* Những công việc quan trọng sẽ được lên kế hoạch và thực thi ngay tức thì, thay vì phải đợi đến Sprint sau.
- *Lead time ngắn:* Đây là hệ quả của lợi thế tức thì khi bỏ qua khoảng thời gian công việc phải nằm trong danh sách *TO DO* ít nhất là tới Sprint sau.
- *Cải tiến workflow:* Bằng cách trực quan hóa luồng công việc, nhóm sẽ biết cách cải tiến workflow liên tục cho phù hợp.
- *Giảm thiểu dư thừa:* Bởi mọi công việc đều được thêm vào hoặc loại bỏ tức thì, việc bỏ qua những công việc không cần thiết là rất rõ ràng.

Khi nào dùng Scrumban?

Câu hỏi này thật ra rất dễ trả lời nếu chúng ta nhìn vào bản chất của Scrum và Kanban: *Scrum được điều hướng bởi giá trị; Kanban được điều hướng bởi sự kiện.*

Khi bắt đầu một Sprint, nhóm thực hành Scrum được điều hướng để mang lại *phân tăng trưởng có giá trị cao nhất*, được xác định qua Sprint Goal.

Khi một công việc phát sinh, nhóm thực hành Kanban được điều hướng để *lead time của sự kiện đó là nhỏ nhất*.

Do đó, Scrumban được điều hướng bởi *sự kiện và giá trị*; có nghĩa là: nhóm thực hành Scrumban được điều hướng để lead time của những sự kiện mang lại giá trị nhất là nhỏ nhất. Scrumban đặc biệt phù hợp với những giai đoạn mà công việc không được xác định trước, được điều hướng bởi sự kiện như:

- vận hành, bảo trì, hỗ trợ khách hàng;
- kiểm thử toàn hệ thống, đóng gói và triển khai;
- nghiên cứu, phát triển.

Ví dụ, khi nhóm phát triển đang thực hiện việc phát triển chức năng *thanh toán qua Paypal*, nhóm phát triển phát hiện ra hệ thống đang triển khai có lỗi trong chức năng đăng nhập và *thanh toán qua thẻ tín dụng*; nhóm sẽ dồn mọi nỗ lực

vào việc fix bug cho chức năng đăng nhập trước, sau đó fix bug cho chức năng *thanh toán qua thẻ tín dụng* đã có trước khi quay lại thực hiện tiếp chức năng *thanh toán qua Paypal*. Hai sự kiện trên có độ ưu tiên cao hơn; trong đó, việc fix bug *chức năng đăng nhập* có giá trị cao hơn, bởi việc fix bug *chức năng thanh toán qua thẻ tín dụng* hoàn toàn không có giá trị nếu người dùng không thể đăng nhập để sử dụng hệ thống. Tất nhiên những chức năng này có thể thực hiện song song nếu nhóm có đủ nguồn lực nhưng đây là một gợi ý về việc xác định giá trị công việc.

So sánh chi tiết

Những điểm khác biệt giữa Scrumban và Scrum cũng như Kanban được tóm gọn như sau:

	Scrumban	Kanban
Vai trò	Nhóm phát triển và những vai trò khác trong phát triển phần mềm nếu cần.	Không xác định.
Họp hàng ngày	Có. Nhằm đồng bộ công việc giữa những thành viên để hoàn thành công việc theo workflow.	Không xác định.
Họp Review và Retrospective	Có. Nhằm chia sẻ thông tin trong nhóm, cải tiến quy trình. Không xác định thời điểm.	Không xác định.

	Scrumban	Scrum
Artifacts	Team Board	Backlog, Team Board, Burn-down Chart.
Sự kiện	Daily Scrum. Các cuộc họp khác không bắt buộc, không xác định thời gian.	Daily Scrum, Scrum Planning, Sprint Review, Sprint Retrospective.
Iteration	Không xác định.	Sprint.
Ước lượng	Có thể (thường là không).	Bắt buộc.
Nhóm	Chấp nhận nhiều nhóm nhỏ phân rã theo chức năng.	Nhóm liên chức năng.
Vai trò	Nhóm phát triển và những vai trò khác trong phát triển phần mềm nếu cần.	Product Owner, Scrum Master, nhóm phát triển
Công việc	Theo workflow.	Theo Sprint.
Thay đổi	Chấp nhận bất cứ thay đổi nào.	Không chấp nhận. Chờ đợi Sprint tiếp theo.
Product Backlog	Tồn tại trên Team Board, được thêm vào tức thì.	Được sắp xếp theo độ ưu tiên.

SCRUMBAN LÀ CHƯA ĐỦ?

Vào một thời điểm, bạn sẽ thấy Scrumban là rất tuyệt vời trong những dự án phát triển phần mềm, đặc biệt trong giai đoạn vận hành và bảo trì, khi những sự kiện xảy ra hoàn toàn bất ngờ, nhưng cần được xử lý để có lead time ngắn. Tuy vậy, trong nhiều trường hợp, Scrumban là không đủ tốt. Điểm yếu của Scrumban là không tận dụng được Sprint vì vậy những sự kiện liên quan tới Sprint đều bị bỏ qua (Sprint Planning, Sprint Review, Sprint Retrospective); điều này hoàn toàn hợp lý vì Scrumban hướng vào workflow và khi không có Sprint Goal thì Sprint cũng không còn ý nghĩa gì để tồn tại.

Tuy vậy, một quy trình sẽ không bao giờ được cải tiến nếu không có hoạt động đánh giá và rút kinh nghiệm từ những gì đã xảy ra. Như tôi đã nói, Sprint Retrospective là điều tuyệt vời nhất Scrum mang lại cho việc phát triển phần mềm bằng việc bắt buộc nhóm Scrum thực hiện sự kiện này để rút kinh nghiệm sau mỗi Sprint. Thật tiếc, điều tuyệt vời này lại không được đưa vào Scrumban. Lý do chính ở đây là, Scrumban không hoạt động theo Sprint, vì thế những công việc trong *TO DO* có thể không bao giờ hết, và nhóm phát triển có thể không bao giờ có một “khoảng lặng” để nhìn lại những gì đã làm và cải tiến workflow.

Do đó, tôi thấy nhiều nhóm thực hành Agile chọn lựa một phương pháp tiếp cận khác, là “ép” Sprint vào Scrumban. Có nghĩa là, dù nhóm phát triển sẽ không có Sprint Goal nhưng khi thời gian này kết thúc, nhóm phải thực hiện việc Retrospective bất kể hiện trạng công việc ra sao. Cách làm này khiến nhóm phát triển luôn đều đặn nhìn lại những gì đã qua để cải tiến workflow nếu cần. Cách làm này không được định nghĩa trong Scrumban hay Kanban, càng không phải là Scrum; nhưng lại khá hiệu quả.

Một số nhóm chọn thực hành với tất cả những vai trò, sự kiện, tạo tác của Scrum nhưng chấp nhận mọi thay đổi trong yêu cầu được thêm vào tức thì, hướng theo workflow; và tất nhiên, sẽ không hoặc có Sprint Goal hoặc có một cách rất mơ hồ. Tuy vậy, do không xác định được cụ thể Sprint Goal, đây lại không phải là ScrumBUT.

Vậy nên, việc kết hợp những phương thức nào và những best practices ra sao cho phù hợp và hiệu quả trong nhóm phát triển là một câu chuyện không đơn giản. Chúng ta sẽ bàn tới điều này kỹ hơn trong những chương tiếp theo.

HIỂU ĐÚNG

Scrumban là cách mở rộng Scrum cho nhóm lớn.

Thật tiếc, Scrumban không phải là cách mở rộng Scrum. Scrum tập trung vào những nhóm nhỏ, và khuyến nghị với 3-9 thành viên. Khi nhóm lớn hơn, việc duy trì nhóm liên chức năng và tự tổ chức là rất khó khăn, bởi nhóm khó đạt được sự đồng thuận về cách làm, cam kết, cũng như tìm ra cách giao tiếp phù hợp. Lúc này, nhu cầu hình thành những *nhóm theo chức năng* như lập trình, kiểm thử... trở nên rõ ràng hơn. Và Scrumban hay Kanban là cách để trực quan hóa quy trình, luồng làm việc giữa những nhóm theo chức năng. Nên về cơ bản, Scrumban là cách tiếp cận khác, chỉ sử dụng những practice của Scrum và Kanban thay vì thực sự là cách mở rộng Scrum.

Kanban và Scrumban tốt hơn Scrum vì xu hướng tăng trưởng hơn.

Thật khó nói rằng phương pháp nào tốt hơn, song lý do để Kanban và Scrumban phát triển (về mặt thi phần) hơn Scrum trong những năm gần đây, theo tôi, là sự trưởng thành hơn của những tổ chức thực hành Agile. Ngày càng nhiều những tổ chức đã trưởng thành từ việc sử dụng Scrum và họ tìm kiếm phương pháp mới, vừa để phù hợp hơn (với việc vận hành, hỗ trợ,...) vừa để tạo ra động lực phát triển. Điều đó không có nghĩa là những nhóm mới thực hành Agile nên “đi tắt đón đầu” bằng cách thực hành Scrumban. Tôi vẫn khuyến nghị mọi nhóm mới thực hành Agile nên chọn eXtreme Programming, Scrum trước khi nghĩ đến những phương pháp khác.

TỔNG KẾT

Kanban là phương pháp theo tư tưởng Lean nhằm trực quan hóa workflow, giúp linh hoạt trong việc lên kế hoạch, được điều hướng bởi lead time.

Scrumban là phương pháp kết hợp những điểm mạnh của Scrum và Kanban giúp nhóm phát triển phần mềm được điều hướng bởi *giá trị* và *lead time*.

TH

KỸ THUẬT
& CÔNG CỤ

Theo quan sát của tôi, hầu hết những nhóm thực hành Agile ở Việt Nam đều chọn bắt đầu bằng việc thực hành Scrum. Đây là một phương pháp rất tốt so với những phương pháp còn lại trong tập hợp những phương pháp Agile bởi Scrum chú trọng vào việc quản lý quy trình phát triển phần mềm đúng nghĩa, với những vai trò, sự kiện được quy định rõ ràng hơn với bài toán quy trình so với những phương pháp khác. Tuy vậy, Scrum chỉ là một framework, nên những quy định trong Scrum nói chung dễ hiểu nhưng khó thực hành với những nhóm mới bắt đầu và chưa có hiểu biết đầy đủ về Agile. *Nhóm liên chức năng, cộng tác với nhau để cùng đạt được Sprint Goal tuy hay nhưng khó thực hành bởi cộng tác như thế nào?* Tương tự, *nhóm cải tiến cách làm thông qua Sprint Retrospective* là khá mơ hồ. Việc đưa ra mục đích chung mà không có cách làm cụ thể khiến Scrum linh hoạt nhưng lại gây lúng túng cho những nhóm mới thực hành. Xét về những điều cụ thể, XP (*eXtreme Programming*), một phương pháp khác dưới chiếc ô Agile, làm tốt hơn rất nhiều; nhưng XP lại rất kém trong việc định nghĩa rõ ràng về cách tổ chức dự án. Do đó, cách làm thông thường là kết hợp Scrum hay Kanban với những best practice của XP. Tuy vậy, tôi không định dành toàn bộ chương này để nói về XP, vì chúng ta còn có nhiều điều thú vị hơn vậy.

BẢNG, GIẤY DÁN

Không nhóm thực hành Agile nào không có bảng. Đây là một tiên đề. Dù hiểu theo cách gì, bảng vật lý hay bảng điện tử, tôi dám khẳng định rằng: *nếu nhóm không có bảng, nhóm chắc chắn không thực hành Agile.*

Nhóm thực hành Agile chú trọng vào việc cộng tác giữa những thành viên trong nhóm, và điều quan trọng nhất là *minh bạch thông tin* bao gồm những công việc đang thực hiện, trạng thái những công việc này, mục tiêu cần đạt được... Và bảng là nơi tốt nhất để chia sẻ những điều này. Trong Scrum hay Kanban bạn có thể thấy nhóm chia sẻ chung một bảng gọi là *Team Board*.

Độ linh hoạt và tần suất cập nhật thông tin trong nhóm thực hành Agile là rất lớn. Thay vì chỉ một người quản lý công việc, cả nhóm phát triển phần mềm đều tự quản lý, chia sẻ hiện trạng công việc thông qua Team Board. Do đó, Team Board được cập nhật liên tục và trạng thái thay đổi nhanh ngay trong một ngày làm việc khiến bảng là một nơi phù hợp.

Tất nhiên, hiệu quả của việc sử dụng bảng trong thảo luận nhóm dưới bất cứ quy trình nào cũng đều phát huy tác dụng; nhưng trong Agile, bảng đóng vai trò quan trọng hơn nhiều. Điều này còn phát triển thành *cơn cuồng bảng và giấy dán* trong những nhóm thực hành Agile và bỗng trở nên một quy chuẩn rằng

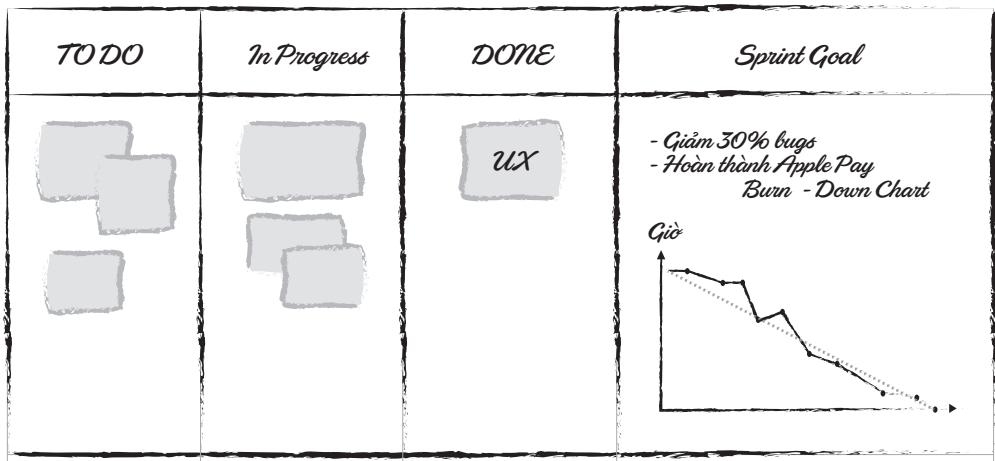
càng nhiều bảng và nhiều giấy dán thì càng agile, càng linh hoạt và nhiều nhóm săn sìng biến bất cứ thứ gì thành bảng: tường, kính, cửa... Nghe khá hài hước, nhưng dù sao quy chuẩn này cũng có lý do, ít nhất là:

- *Những công việc, yêu cầu, thay đổi trong nhóm thực hành Agile thường xuất hiện với tần suất rất cao và tồn tại trong thời gian ngắn.* Bảng trắng và giấy dán chẳng phải sinh ra với mục đích đó sao? Chúng ta có việc mới: viết lên một mẫu giấy dán vào danh sách TO DO. Bắt đầu làm việc: chuyển mẫu giấy dán sang danh sách DOING. Công việc bị huỷ bỏ: Vứt giấy dán vào sọt rác. Thật đơn giản.
- *Thông tin được chia sẻ khắp nơi.* Điều này khiến nhóm thực hành Agile tận dụng mọi khoảng trống để truyền tải thông tin trong nhóm.
- *Cộng tác giữa các thành viên cần sự vui vẻ.* Một nhóm các thành viên không vui vẻ thì không thể thực hành Agile, bởi họ sẽ không dễ dàng chia sẻ và cộng tác với nhau. Giấy dán và bảng là nơi các thành viên rất dễ viết, vẽ lên đó (thậm chí cả những thứ chẳng liên quan) nhằm tạo phấn khích, sự vui vẻ trong nhóm.

Tôi không phải là một tín đồ của bảng và giấy dán, vì vậy số lượng bảng hay giấy dán với tôi không có nhiều ý nghĩa; nhưng tôi đồng ý rằng sử dụng bảng và giấy dán một cách hiệu quả mang lại lợi ích lớn trong nhóm. Khi bước vào một nhóm thực hành Agile, tôi hay quan sát bảng, bởi nơi đó nói lên rất nhiều điều về nhóm. Nhóm càng linh hoạt thì sự tập trung của các thành viên vào Team Board càng cao, và độ chuyển động của Team Board càng lớn. Nhóm càng linh hoạt, càng trao đổi nhiều thì tần suất viết-xoá bảng càng cao.

Team Board nói chung tồn tại muôn hình vạn trạng và rất khác biệt giữa các nhóm nhưng thông thường được tổ chức theo nguyên tắc:

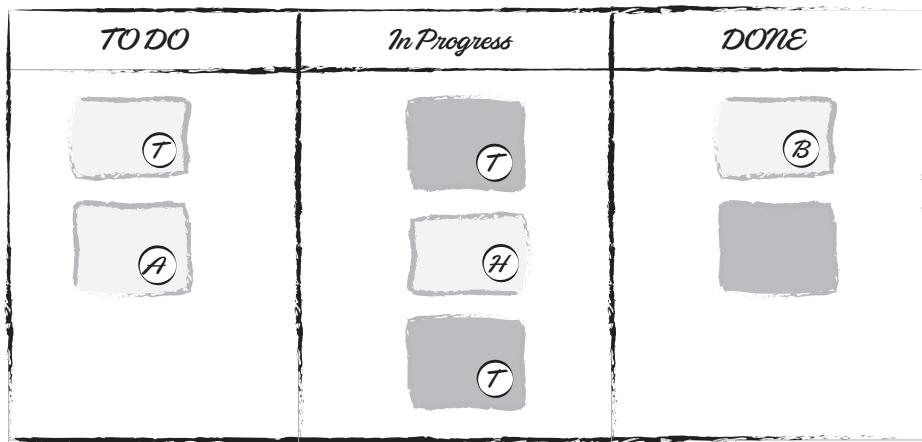
- Các cột được chia theo trạng thái của công việc, cột đầu tiên luôn là TO DO và cột cuối cùng là DONE. Các cột có thể thể hiện trạng thái rất chung chung như *In Progress* (*đang thực hiện*), hoặc rõ ràng hơn như *Technical Analysis* (*phân tích kỹ thuật*), *Development* (*đang phát triển*), *Testing* (*đang kiểm thử*);
- Các dòng có thể được sử dụng để phân nhóm công việc theo một tiêu chí phù hợp, ví dụ theo User Story hay theo thành viên trong nhóm.



Hình 5.1: Team Board cơ bản

Khi nhóm cần phân nhóm những công việc theo nhiều tiêu chí, việc sử dụng linh hoạt giấy dán và những phụ kiện sẽ mang lại hiệu quả cao. Bạn có thể thấy như ví dụ dưới đây:

- Giấy dán màu vàng được sử dụng cho User Story.
- Giấy dán màu xanh được sử dụng cho bug.
- Giấy dán màu đỏ được sử dụng cho những bug nghiêm trọng.
- Một miếng dán nhỏ hoặc nam châm có điền tên thể hiện thành viên trong nhóm đang thực hiện công việc trên giấy dán chính.

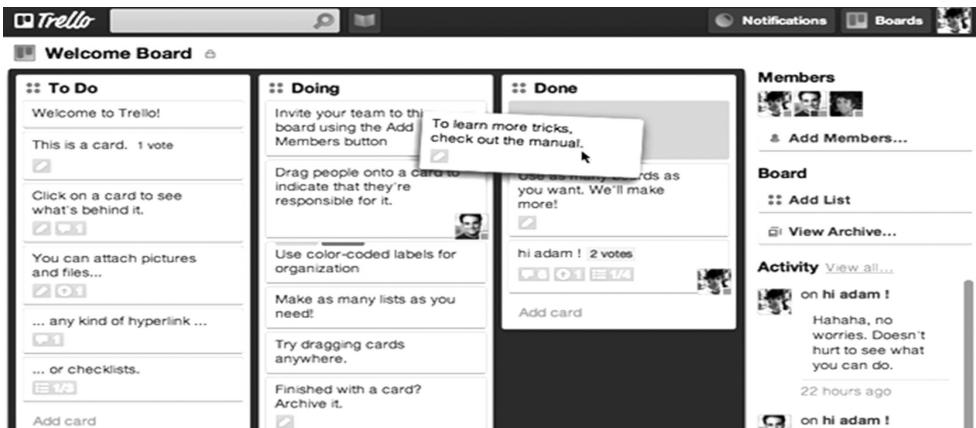


Hình 5.2: Team Board sử dụng giấy nhiều màu kết hợp tên thành viên

Dù nhóm phát triển sử dụng Team Board dưới dạng nào thì những nguyên tắc sau cũng cần được tuân theo để đảm bảo hiệu quả:

- *Tại bất cứ thời điểm nào, Team Board cũng là một ánh xạ chính xác những gì đang được thực hiện trong nhóm.* Thật là tệ nếu một User Story nằm ở cột Testing và được thực hiện kiểm thử bởi một thành viên trong nhóm, trong khi vẫn được phát triển bởi một thành viên khác và 2 thành viên này hoàn toàn không biết về sự mâu thuẫn này.
- *Quy trình thực hiện, workflow phải được thống nhất trong nhóm.* Việc này đảm bảo mọi thành viên trong nhóm thực hiện quy củ. Nhóm có thể tạo nhiều cột để workflow rõ ràng hơn; nhưng đừng làm vậy chỉ để phục vụ công tác giám sát, quản lý của cấp cao hơn. Đừng ngần ngại sử dụng tối thiểu số cột nếu cách làm này phù hợp.
- *Các thành viên trong nhóm cân nhắc sự tập trung phù hợp với Team Board.* Đặc biệt với những thành viên thực hiện những phần sau của công việc (kiểm thử) để không khiến công việc đình trệ. Bảng điện tử rất hay khi hỗ trợ những thành viên trong nhóm nhận thông báo khi có một hoạt động xảy ra trên Team Board; nhưng đây cũng là điểm yếu, bởi nó dẫn đến sự mất tập trung của các thành viên với công việc mình đang làm, cũng như làm giảm tính cộng tác trong nhóm. Lựa chọn chính xác phương pháp và cân bằng sự tập trung với Team Board giúp nhóm hiệu quả hơn.

Ngày nay, có rất nhiều công cụ quản lý dự án hỗ trợ Team Board theo 2 phương pháp chính là Scrum và Kanban, cũng như cung cấp cách cấu hình Team Board cho phù hợp với nhóm. Phổ biến nhất có lẽ là Jira, Gitlab, Microsoft Team Foundation, Pivotal Tracker, Trello... mỗi sản phẩm đều có những ưu và nhược điểm riêng. Nhiều nhóm mới thực hành Agile mãi phân vân lựa chọn công cụ nào bởi công cụ nào cũng không mang đến cho họ sự thỏa mãn về chức năng; trong khi nhiều nhóm thực hành thành công Agile lại có xu hướng quay về với bảng và giấy dán vật lý. Theo tôi, trừ khi bạn cần tích hợp những công cụ trên với những hệ thống hiện có (chat, email...) hoặc theo dõi những thống kê; bạn không cần sử dụng đến bảng điện tử. Nếu bạn thực sự cần bảng điện tử, hãy chọn lấy một công cụ nhóm dễ sử dụng nhất. Bởi mục tiêu tối thượng của Agile và Team Board không phải là một công cụ quản lý dự án phức tạp. Agile hướng đến sự linh hoạt và cộng tác; sử dụng một công cụ nặng nề với hàng chục chức năng không cần đến chỉ làm giảm sự linh hoạt và cộng tác trong nhóm phát triển.



Hình 5.3: Bảng điện tử Trello

Trello là một bảng điện tử được ưa chuộng bởi nhiều nhóm thực hành Agile vì sự đơn giản

CONTINUOUS INTEGRATION – TÍCH HỢP LIÊN TỤC

Theo tôi, đây là công cụ quan trọng bậc nhất trong những nhóm thực hành Agile với quy mô vừa tới lớn. *CI (Continous Integration – Tích hợp liên tục)* là một quy trình / công cụ giúp nhóm phát triển ngay lập tức nhận diện được những ảnh hưởng của một commit (một đoạn code hay một chức năng được thêm vào) với toàn bộ hệ thống nhằm phản ứng tức thì để đảm bảo toàn hệ thống hoạt động như mong đợi.

Khác với mô hình phát triển phần mềm truyền thống khi việc những module được thiết kế rất tỉ mỉ, phát triển độc lập và được thực hiện việc tích hợp vào giai đoạn cuối của vòng đời phát triển nhằm phục vụ việc kiểm thử tích hợp và kiểm thử hệ thống; Agile coi trọng việc phát triển những chức năng liên tục theo kiểu *bồi đắp*. Cá biệt, trong Scrum, một vài phần tăng trưởng phải được chuyển giao sau mỗi Sprint – là những chức năng phải vận hành được bởi người dùng. Do đó, việc kiểm thử tích hợp và kiểm thử hệ thống diễn ra thường xuyên hơn (tối thiểu là trong mỗi Sprint), khiến việc thực hiện công việc tích hợp thủ công không khả thi. Ý tưởng về một công cụ thực hiện việc tích hợp vì thế đã xuất hiện.

Tuy vậy, chúng ta có thể hiểu CI là một quy trình, yêu cầu những thay đổi trên hệ thống phải được nhanh chóng nhận biết sự ảnh hưởng của chúng thông qua việc tích hợp sớm với hệ thống đang có, và thực hiện việc kiểm thử tích hợp cũng như kiểm thử hệ thống. Lúc này CI liên quan mật thiết đến khái niệm *daily build: toàn bộ mã nguồn của hệ thống phải được build hàng ngày nhằm nhận biết những lỗi tiềm năng và khắc phục sớm*.

Tại sao phải là *daily build*? Hãy nhớ rằng, bất cứ thành viên nào trong nhóm phát triển đều có thể là một nhà thiết kế phần mềm, và công việc của họ có thể ảnh hưởng rất nhiều tới hệ thống. Ví dụ, nhóm phát triển chạy nước rút với Sprint 2 tuần (10 ngày làm việc), với 2 lập trình viên cùng thực hiện việc thay đổi cơ sở dữ liệu nhằm hoàn thành 2 User Story song song và mất 5 ngày thực hiện, việc tích hợp vào ngày thứ 6 với hàng tá xung đột về ràng buộc trong cơ sở dữ liệu có thể là một cơn ác mộng. Daily build chính là giải pháp để phát hiện xung đột xảy ra ngay từ ngày thứ 2 để nhóm phát triển có cách giải quyết thích hợp.

Nói chung, *daily build* giống như *Daily Scrum* về khía cạnh *source code*. Trong *Daily Scrum*, nhóm thực hiện việc tích hợp, đồng bộ công việc với nhau cho các task hay User Story. Trong *daily build*, nhóm thực hiện việc tích hợp, đồng bộ công việc với nhau cho mã nguồn và những thành phần liên quan như database, service... Xét cho cùng, kết quả của những task hay User Story trên cũng là mã nguồn và những thành phần liên quan; việc thực hiện *Daily Scrum* mà không có *daily build* giống như chúng ta đang đồng bộ những vấn đề “trên trời” mà không giải quyết những xung đột cụ thể.

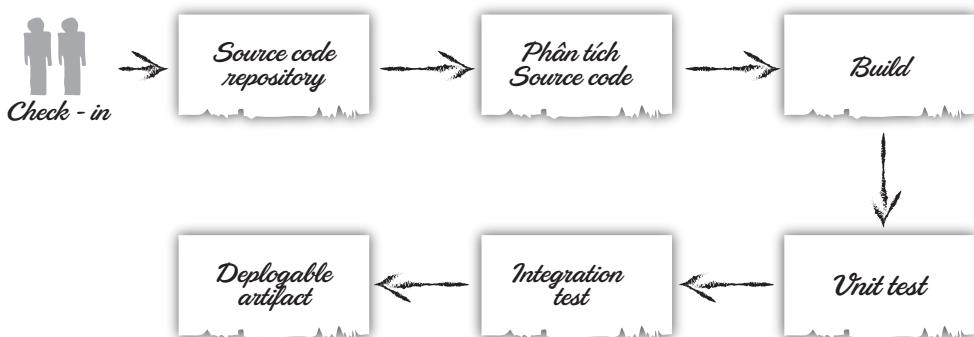
Với nhiều nhóm thực hành Agile, *daily build* thậm chí là không đủ, nhóm cần tới *commit build* – thực hiện việc tích hợp, kiểm thử cho mỗi commit vào *source code repository*.

Một hệ thống CI thông thường thường thực hiện những tác vụ sau:

1. Phát hiện thay đổi trong *source code repository* (xuất hiện commit mới).
2. Phân tích chất lượng *source code*.
3. Thực hiện build.
4. Chạy toàn bộ unit test.
5. Chạy toàn bộ integration test.
6. Sinh ra những tạo tác có thể triển khai được, gọi là *deployable artifact*.
7. Có thể, deploy những artifact này và thực hiện những kiểm thử khác nếu cần.

Nếu một trong những bước trên không thành công:

- tùy thuộc vào mức độ nghiêm trọng, việc tích hợp có thể dừng lại hoặc đi tiếp;
- kết quả tích hợp được thông báo tới nhóm phát triển qua email, hệ thống chat... Thông qua source code repository, CI có thể nhận biết cá nhân đã thực hiện việc commit gây ra lỗi trong việc tích hợp;
- Nhóm phát triển hoặc cá nhân thực hiện commit thực hiện sửa lỗi và commit;
- CI phát hiện thay đổi trong source code repository và thực hiện lại những bước trên.



Hình 5.4: Các tác vụ được thực hiện bởi CI

Trong những tác vụ trên, tác vụ 1, 3, 6 thực sự rất dễ thực hiện và được hỗ trợ bởi hầu hết những công cụ CI hiện có trên thị trường. Những tác vụ 2, 4, 5 không đơn giản chỉ là sự hỗ trợ của công cụ, tư tưởng và cách thực hiện phía sau quan trọng hơn rất nhiều; chúng ta sẽ bàn về những vấn đề này ở phần sau. Tác vụ 7 lại liên quan tới một hệ thống CD (Continuous Delivery – chuyển giao liên tục) với tư tưởng giống với CI nhưng dưới góc độ *triển khai* và lại phụ thuộc vào tần suất release của sản phẩm và độ phức tạp của môi trường nên không hẳn là một công cụ tiên quyết trong việc thực hành Agile.

Tuy vậy, nhiều nhóm thực hành Agile vẫn lựa chọn việc triển khai CI và CD đồng thời bởi một trong những best practice của CI là đảm bảo môi trường đồng nhất (hoặc gần giống nhất) giữa môi trường kiểm thử (tích hợp) và môi trường production.

Chúng ta dễ dàng nhận thấy ưu điểm của commit build so với daily build vì cò

lập được thay đổi theo từng commit khiến việc xử lý xung đột đơn giản hơn. Tuy vậy, thời gian để CI thực hiện toàn bộ 7 tác vụ trên có thể rất nhiều với một số môi trường của dự án khiến việc này không khả thi; nên nhiều nhóm thực hành Agile vẫn lựa chọn daily build. Tôi là một người theo trường phái commit build và luôn cố gắng duy trì CI theo cách này. Với những dự án phức tạp, tôi thường sử dụng nhiều *build agent* để thực hiện việc tích hợp song song với những commit cùng thời điểm. Một giải pháp khác là, lược bỏ những tác vụ không cần thiết (ví dụ tác vụ 7) để vẫn thực hiện việc tích hợp với từng commit và thực hiện một *long build* vào cuối ngày.

Hiện nay có rất nhiều công cụ CI cho phép nhóm thực hành Agile lựa chọn với những tác vụ cơ bản như trên, phổ biến nhất có lẽ là Jenkins – hệ thống mã nguồn mở với rất nhiều plugin phù hợp với nhiều điều kiện hệ thống khác nhau. Tôi cũng là một kẻ hâm mộ Jenkins nhưng lại thường sử dụng những công cụ khác khi có thể, chỉ bởi giao diện của Jenkins quá tệ và cũng vì có quá nhiều người sử dụng.



Những nhóm quen thuộc với các hệ thống CI thường thích Jenkins vì Jenkins rất mạnh mẽ và có cộng đồng lớn cùng nhiều plugin miễn phí có chất lượng cao. Song Jenkins không thực sự đơn giản với người mới tiếp cận.

Với những nhóm mới thực hành CI, tôi khuyến nghị sử dụng TeamCity bởi:

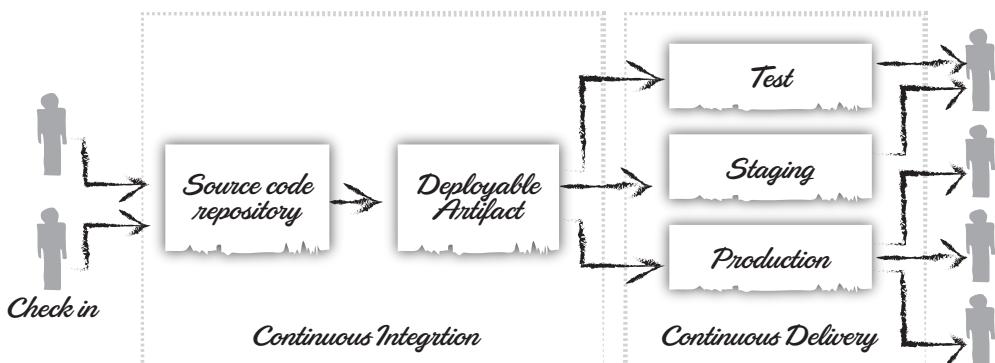
- TeamCity dễ cài đặt và trực quan;
- TeamCity có chế độ Wizard, tự động nhận diện loại source code version control như SVN, Git, Mercurial... cũng như cấu trúc của project, workspace thông dụng như Visual Studio, XCode, Gradle,... và đưa ra những cấu hình tương ứng phù hợp với project hay workspace của bạn. Thật ra những cấu hình này đều được thực thi dưới dạng dòng lệnh; song với những người chỉ quen thao tác qua IDE, chế độ Wizard của TeamCity thực sự tuyệt vời.

Nói chung, bạn có thể cài đặt và cấu hình một hệ thống CI dùng được với TeamCity dưới 15 phút.

CONTINUOUS DELIVERY – TRIỂN KHAI LIÊN TỤC

Bên cạnh Continuous Integration (CI), Continuous Delivery (CD) đang trở thành một công cụ không thể thiếu cho những nhóm thực hành Agile. Bài toán đặt ra với CD cũng giống như với CI: với phương pháp phát triển phần mềm truyền thống, việc tích hợp và chuyển giao là những công đoạn gần cuối cùng trong vòng đời phát triển sản phẩm; theo phương pháp Agile, việc tích hợp và chuyển giao được thực hiện liên tục sau mỗi Iteration.

CD được hiểu là khả năng nhanh chóng và đều đặn triển khai những thay đổi từ bug fix tới những chức năng mới tới môi trường được sử dụng bởi người dùng một cách an toàn và hiệu quả.



Hình 5.5: Các tác vụ được thực hành bởi CI và CD

Theo phương pháp phát triển phần mềm truyền thống, triển khai được coi là một công đoạn tách biệt với việc phát triển và được thực hiện không đều đặn phụ thuộc vào tần suất triển khai những phiên bản mới (thường là sau vài tháng hoặc một năm), được thực hiện bởi nhóm vận hành chuyên nghiệp với một kịch bản và kế hoạch chi tiết. Những nhóm thực hành Agile tiếp cận theo phương pháp khác: việc triển khai được thực hiện liên tục bởi những thành viên trong nhóm, sau mỗi Iteration, thậm chí hàng ngày. Do tần suất thực hiện việc triển khai tăng lên, việc sử dụng những công cụ tự động hóa trở thành một nhu cầu tất yếu.

Việc sử dụng những công cụ tự động hóa mang lại những hiệu quả vượt trội trong việc triển phát triển sản phẩm:

- *Chất lượng tốt hơn.* CD cho phép nhóm phát triển liên tục triển khai và thử nghiệm những phần tích hợp trên môi trường thật và nhận biết những

vấn đề có thể xảy ra cũng như nhanh chóng triển khai những phiên bản vá lỗi.

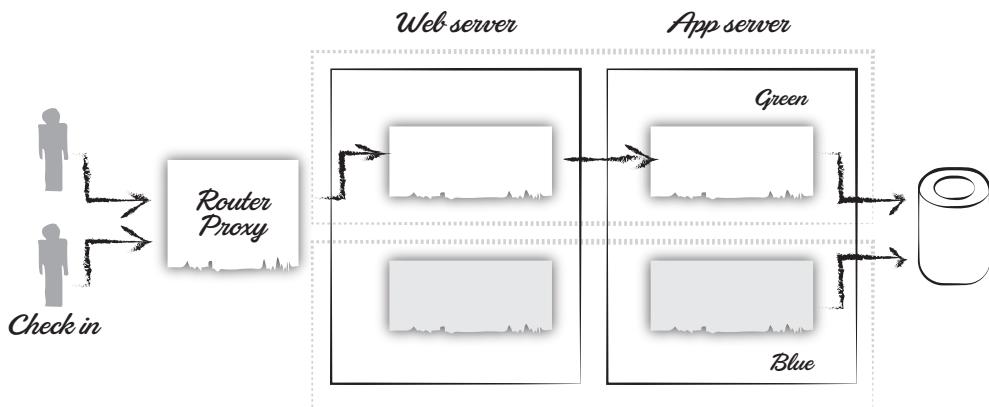
- *Giảm thiểu rủi ro.* Cũng giống như việc tích hợp liên tục, CD giúp giảm thiểu rủi ro qua việc liên tục kiểm tra khả năng tương thích của những thành phần mới trên môi trường thực tế. Việc triển khai bằng những công cụ tự động cũng đảm bảo *trăm lần như một* và hạn chế những sai sót do con người thực hiện.
- *Tối ưu time-to-market.* Triển khai thường là một trong những công việc nặng nhọc đòi hỏi sự tỉ mỉ cùng tính cẩn trọng cao; những nhóm sử dụng phương pháp phát triển phần mềm truyền thống có thể cần rất nhiều thời gian để đưa ra một phiên bản sửa lỗi. CD giúp việc sửa lỗi cũng như những tính năng mới nhanh chóng được triển khai nhằm tối ưu trải nghiệm người dùng.
- *Nhóm phát triển hiệu quả hơn.* Thay vì đặt trách nhiệm triển khai cho một nhóm cụ thể - những người không thực sự hiểu về việc phát triển phần mềm, CD giúp bất kỳ thành viên nào trong nhóm phát triển sản phẩm có thể triển khai và kiểm thử ngay những thay đổi, sửa lỗi hay chức năng mới, cũng như giá trị mang lại cho người dùng chỉ sau một vài phút.

Cũng giống như CI, việc triển khai CD trong phát triển phần mềm luôn mang lại kết quả tích cực trong việc nhanh chóng phản hồi với những sự kiện từ fix bug tới triển khai một chức năng mới. Ngày nay có rất nhiều công cụ hỗ trợ nhóm trong việc triển khai CD như Ansible, Puppet, Octopus... giúp bất cứ thành viên nào trong nhóm phát triển cũng có thể chịu trách nhiệm trong việc triển khai cả hệ thống chỉ với một nút nhấn. Những công cụ này cũng giúp việc triển khai hệ thống không còn là gánh nặng mà trở nên dễ dàng và an toàn hơn. Có rất nhiều phương pháp giúp triển khai hệ thống đảm bảo độ an toàn, một trong số đó là phương pháp blue-green deployment, trong đó (giả sử):

- Green: Là phần hệ thống đang hoạt động ổn định;
- Blue: Phần hệ thống đang được deploy;
- Red: Phần hệ thống được deploy không thành công.

Trong mỗi lần triển khai, phần blue sẽ được khởi tạo, triển khai. Nếu việc triển khai không thành công (red), phần green vẫn đảm bảo hệ thống được hoạt động bình thường. Nếu việc triển khai mới thành công (blue), hệ thống ngay lập tức được trả sang phần blue. Nếu nhóm phát triển ngay lập tức phát hiện ra vấn đề, phần green sẽ được phục hồi nhằm đảm bảo hệ thống trở lại trạng thái cũ. Những công

cụ CD trên thị trường hiện nay đều cung cấp việc versioning những phiên bản được triển khai nên việc rollback hệ thống trở nên tương đối đơn giản. Trong môi trường web, bằng việc sử dụng proxy và multi tenancy theo phương pháp blue-green deployment, hệ thống gần như không có down time.



Hình 5.6: Blue-green deployment

Tuy nhiên, cũng giống như CI, công cụ chỉ là bể nổi của kỹ thuật thực hiện. Những tư tưởng, phương pháp và rộng hơn là văn hoá cộng tác trong nhóm phát triển để việc áp dụng CD quan trọng và khó khăn hơn. Lúc đó, triển khai không được coi là một công đoạn độc lập, được thực hiện bởi một nhóm vận hành độc lập, mà là một công việc được thực hiện hàng ngày, bởi những thành viên trong nhóm phát triển.



Cùng với CD, thế giới phát triển phần mềm đang chuyển dịch sang một phương pháp mới cộng tác chặt chẽ giữa việc phát triển và triển khai phần mềm, là DevOps. Bạn hãy tìm hiểu thêm về DevOps.

DATABASE VERSIONING

Theo tôi quan sát, *database versioning* (*DB versioning – phiên bản hoá cơ sở dữ liệu* – CSDL) thường là vấn đề khó khăn nhất đối với các nhóm triển khai Agile trong việc thực hành CI và CD; thậm chí nhiều nhóm phát triển không nhận ra vấn đề này.

Mọi lập trình viên ngày nay đều hiểu sự quan trọng của những công cụ quản lý source code và có thể sử dụng thành thạo SVN, Git..., tuy nhiên ít người nhận biết được tầm quan trọng và biết cách quản lý những thay đổi trên CSDL. Tại sao? Bởi theo phương pháp phát triển phần mềm truyền thống, CSDL thường được coi như một phần đặc biệt quan trọng và được thiết kế rất chi tiết trong giai đoạn thiết kế phần mềm; và gần như không được phép thay đổi trong quá trình phát triển. Việc thiết kế CSDL thường được thực hiện bởi những người đặc biệt quan trọng nhằm đảm bảo thiết kế tối ưu và đáp ứng tầm nhìn lâu dài của ứng dụng.

Tuy nhiên, phương pháp phát triển phần mềm mới đã thay đổi hoàn toàn cách suy nghĩ và sử dụng CSDL; phương pháp phát triển hướng CSDL (DB oriented programming) đã không còn là duy nhất. Có nghĩa là, CSDL giờ đây chỉ là một thành phần trong phần mềm đơn thuần, không phải là một phần “bất khả xâm phạm”, càng không phải là thành phần đầu tiên được nghĩ tới khi thiết kế hệ thống. Dữ liệu vẫn luôn đặc biệt quan trọng, nhưng cấu trúc của dữ liệu (thiết kế CSDL) thì không hẳn. Trong phương pháp phát triển phần mềm Agile, mọi thành viên trong nhóm phát triển đều có thể thay đổi thiết kế CSDL bất cứ lúc nào nhằm phục vụ cho chức năng mình đang phát triển. Điều này dẫn đến tần suất thay đổi thiết kế CSDL lớn hơn; và vấn đề quản lý thiết kế CSDL cũng phát sinh.

Thứ nhất, khi một lập trình viên thay đổi CSDL, những thành viên khác trong nhóm phát triển không nhận biết được thay đổi này. Vấn đề này được thực hiện rất tốt với source code version control như SVN hay Git; những thành viên trong nhóm chỉ cần check-out là nhận được những thay đổi trên source code. Thứ hai, khi những thay đổi này gây ra xung đột (conflict), nhóm sẽ giải quyết như thế nào? Vấn đề này phức tạp hơn rất nhiều; vì mỗi thay đổi trên CSDL sau khi được áp dụng sẽ không thể roll-back; không đơn giản như cách chúng ta giải quyết xung đột trên source code.

Theo tôi quan sát, những nhóm phát triển thường áp dụng những phương pháp sau:

- *Sử dụng môi trường chia sẻ.* Cách dễ nhất (và đôi khi là cách tốt nhất) là sử dụng môi trường phát triển chia sẻ (shared environment); mỗi lập trình viên làm việc trên máy tính của mình với source code được check-out

và phát triển hoàn toàn độc lập, nhưng sử dụng chung một DB server. Phương pháp này có rất nhiều điểm lợi, đặc biệt là đảm bảo một môi trường chung với mọi sự thay đổi được cập nhật kịp thời tới mọi lập trình viên. Tuy nhiên, vẫn có ít nhất 2 vấn đề có thể phát sinh:

- *Hiệu năng.* Trong nhiều dự án (không xử lý dữ liệu lớn), hiệu năng từ việc sử dụng môi trường chia sẻ thấp hơn nhiều hiệu năng xử lý trên máy tính cá nhân (vấn đề đường truyền, phân chia module...); gây ảnh hưởng tới năng suất làm việc.
- *Quản lý revision.* Đây là vấn đề không có giải pháp toàn diện. Chúng ta sẽ xử lý thế nào với trường hợp sau: Sau một vài thay đổi, nhóm phát hiện ra rằng một số thay đổi gần đây không đúng và muốn roll-back? Vấn đề này thường xuyên xảy ra giống như việc chúng ta phải revert những commit lỗi khi sử dụng source code version control. Nhưng chúng ta không thể làm vậy với CSDL.
- *Tạo những script tương ứng với từng thay đổi.* Ý tưởng ở đây là, nhóm lưu trữ một CSDL ổn định và tương ứng với mỗi thay đổi, từng script sẽ được tạo ra. Trong trường hợp cần revert commit, nhóm sẽ restore CSDL này và áp dụng từng script theo tuần tự tới trước commit được revert. Vấn đề khác có thể nảy sinh, bằng cách nào những lập trình viên biết cách tạo những script theo thứ tự hợp lý khi việc phát triển, commit của họ được thực hiện song song trong khi sự thay đổi trên CSDL cần thực hiện tuần tự? Bằng cách nào nhóm có thể phát hiện ra những xung đột trên CSDL?

Tuy vậy, ý tưởng về việc scripting DB (kịch bản hoá CSDL) là một ý tưởng hay, và được phát triển thành DB versioning. Cơ bản có thể được mô tả như sau:

- *Mọi thay đổi trên CSDL phải được tạo bằng script.* Thay vì thực hiện tạo một bảng mới qua công cụ có giao diện người sử dụng, chúng ta nên viết thành câu lệnh CREATE TABLE và lưu trong file SQL.
- *CSDL cũng được quản lý bởi source code version control.* Khi mọi thay đổi trên CSDL là script, chúng ta hiểu rằng CSDL cũng là source code và những file này được quản lý bởi source code version control là điều hợp lý.
- *Mỗi script thay đổi trên CSDL gắn với từng phần phát triển phải được commit đồng thời.* Ví dụ, lập trình viên thực hiện User Story quản lý khách hàng cần check-in source code và script tạo ra bảng KhachHang trong một commit. Điều này giúp cho việc quản lý (đặc biệt là revert) trở nên đơn giản hơn.
- *Mọi thay đổi trên CSDL được kiểm tra trong quá trình build hoặc start-up của ứng dụng.* Theo tôi, việc kiểm tra nên được thực hiện sớm nhất có thể (tốt

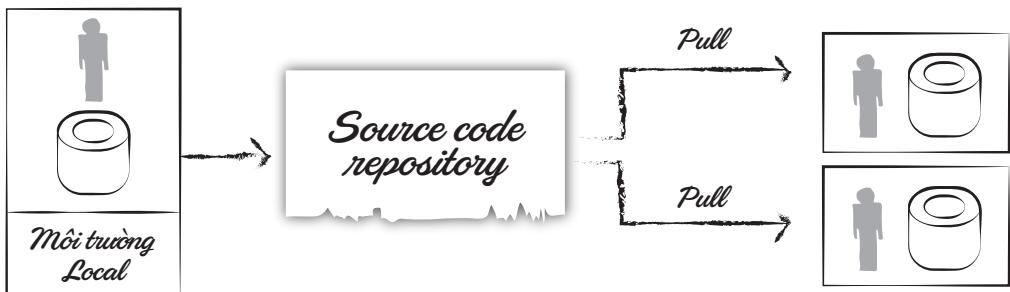
nhất là trong quá trình build) nhằm giảm thiểu thời gian lập trình viên nhận biết những thay đổi trên CSDL. Nếu việc áp dụng những script này không thành công, quá trình build hoặc ứng dụng khởi tạo sẽ thất bại; qua đó lập trình viên nhận biết được sự thay đổi và những xung đột đang xảy ra trên CSDL.

- *Các version của sự thay đổi được lưu trữ tại chính CSDL đó.* Một cách tiếp cận đơn giản là tạo ra một bảng lưu trữ lịch sử những script đã được áp dụng trên chính CSDL đó.

Kỹ thuật trên sẽ khiến CSDL và những thay đổi trên CSDL (script) được coi là source code; giúp lập trình viên dễ dàng quản lý hơn rất nhiều, cả về tư tưởng lẫn kỹ thuật.

Hiện nay, thị trường cung cấp nhiều công cụ hoặc thư viện hỗ trợ việc versioning DB như dbUp, RoundhousE, các sản phẩm của Red Gate..., hầu hết đều có chung tư tưởng trên, và có thể có thêm một số chức năng khác như:

- *Restore DB.* Khởi tạo hoàn toàn một CSDL từ các scripts là điều tuyệt vời khi chúng ta có bộ lịch sử đầy đủ; tuy nhiên, quá trình này thường tốn khá nhiều thời gian thực hiện. Cách làm tốt hơn là nhóm phát triển “chốt” một CSDL ổn định, được coi là điểm bắt đầu và chỉ quản lý những thay đổi bắt đầu từ phiên bản này. Sau một khoảng thời gian, một phiên bản ổn định khác có thể được thay thế. Tính năng này nhằm giảm bớt thời gian khởi tạo CSDL nguyên thuỷ.
- *Có thể roll-back khi gặp lỗi.* Khi tạo một script cho sự thay đổi, lập trình viên cũng tạo một *roll-back script*. Khi gặp lỗi, công cụ này sẽ thực hiện những *roll-back script* này theo thứ tự ngược lại giúp CSDL quay lại trạng thái trước khi bị thay đổi. VD: script *ALTER TABLE ADD COLUMN Address NVARCHAR(50)*; có roll-back script là *ALTER TABLE Customer DROP COLUMN Address*;
- *Sử dụng chính CSDL hiện có lưu trữ sự thay đổi.*
- *Tích hợp với những công cụ CI, CD.* Đây là một chức năng tuyệt vời khi kết hợp với những công cụ CI, CD. Đây là một chức năng tuyệt vời khi kết hợp với những công cụ CI, CD. Đây là một chức năng tuyệt vời với khi kết hợp với những công cụ CI, CD. Đây là một chức năng tuyệt vời với khi kết hợp với những công cụ CI, CD. Đây là một chức năng tuyệt vời với khi kết hợp với những công cụ CI, CD.



Hình 5.7: Database được versioning qua source code repository

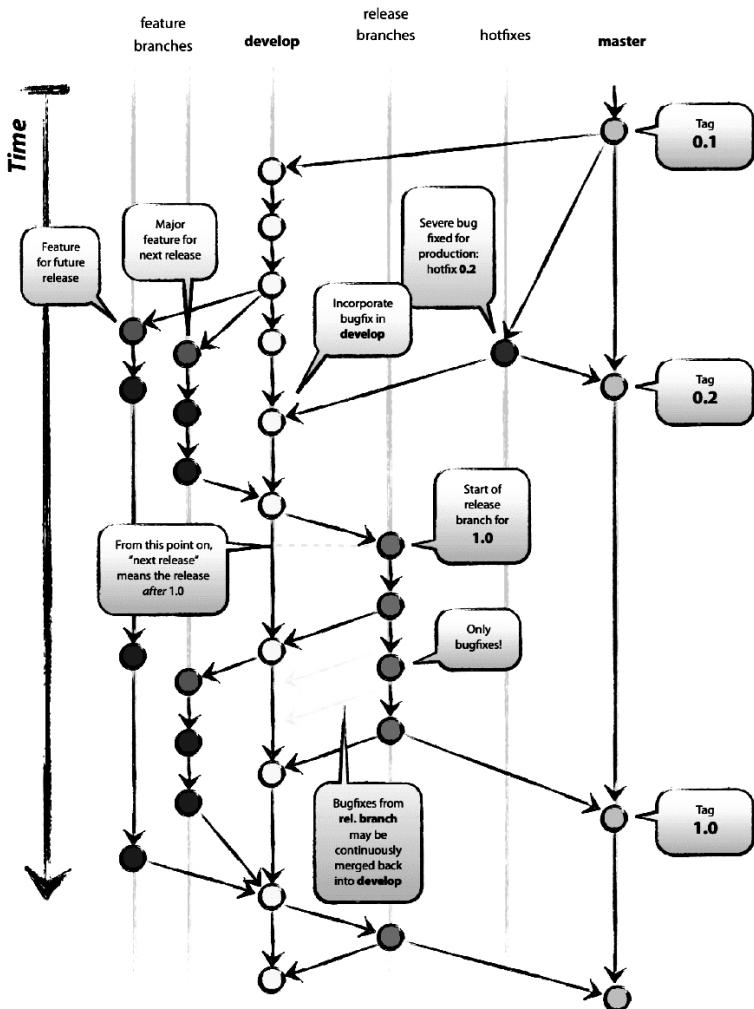
BRANCH STRATEGY

Khái niệm branch (nhánh) trong source code version control đã quá phổ biến nhưng sử dụng hiệu quả branch thì lại là một câu chuyện khác. Khi nào thì branch off? Khi nào thì merge một branch vào branch khác? Cách làm này được gọi là *branch strategy* (chiến lược phân nhánh). Cần lưu ý, branch strategy là vấn đề của việc phát triển phần mềm nói chung, không chỉ Agile; nhưng branch strategy là một trong những kỹ thuật quan trọng và liên quan trực tiếp tới khái niệm *collective ownership source code* (source code được sở hữu bởi tập thể) trong những nhóm thực hành Agile. Collective ownership source code chỉ ra rằng, toàn bộ source code được sở hữu bởi toàn nhóm phát triển, không phải sở hữu cá nhân (mỗi thành viên sở hữu một phần source code theo chức năng, module,...), nghĩa là bất cứ thành viên nào trong nhóm cũng có thể chỉnh sửa vào bất cứ phần source code nào để thực hiện công việc phát triển của mình. Cách làm này gây ra rất nhiều conflict (xung đột) khi nhóm thực hiện việc phát triển những chức năng đồng thời vì một file có thể được chỉnh sửa cùng lúc bởi nhiều thành viên, phục vụ cho nhiều mục đích khác nhau.

Có rất nhiều branch strategy khác nhau tuỳ thuộc vào tình hình cụ thể của dự án, nhưng nổi tiếng nhất là Gitflow (được khuyến nghị bởi Github), bạn có thể tham khảo như sau.

Gitflow

Gitflow có thể được tóm lược sau:



Hình 5.8: Gitflow

Những branch chính:

- *master*: Branch chính, luôn ở trạng thái deploy-ready (có thể deploy) tức là đạt sự ổn định cao. Branch master chỉ được merge vào từ branch *develop*.
- *develop*: Branch phục vụ cho việc phát triển. Chừng nào một commit còn tồn tại trên branch *develop* mà không được merge vào branch *master*, chức năng đó đang trong quá trình phát triển và chưa đạt sự ổn định để deploy.

Những branch khác:

- *hotfix*: Thường được sử dụng để thực hiện hotfix và ngay lập tức có thể được deploy, vậy nên branch hotfix cần được branch off (tạo) từ branch master và merge vào branch master để deploy, cũng như merge vào branch develop nhằm đảm bảo những hotfix này tồn tại cho lần release tiếp theo
 - Branch off từ: master
 - Merge vào: master, develop
 - Đặt tên: hotfix-*
- *feature*: Thường được sử dụng để phát triển một chức năng hoàn chỉnh
 - Branch off từ: develop
 - Merge vào: develop
 - Đặt tên: feature-*
- *release*: Thường được sử dụng để chuẩn bị cho các bản release. Một khi release được merge vào master, một bản release mới đã sẵn sàng.
 - Branch off từ: develop
 - Merge vào: master, develop
 - Đặt tên: release-*

Bạn có thể tham khảo thêm về Gitflow tại một bài viết khá nổi tiếng (<http://nvie.com/posts/a-successful-git-branching-model/>) và tham khảo thêm một số branching strategy khác tại <https://www.atlassian.com/git/tutorials/comparing-workflows>

Branching strategy cho nhóm

Gitflow trông thật tuyệt và hoàn hảo? Gần như vậy. Gitflow hoạt động hoàn hảo trong một môi trường lý tưởng và thực sự tốt trong rất nhiều dự án phần mềm. Trong 1 dự án của công ty, chúng tôi có branch strategy riêng và nó tỏ ra hiệu quả hơn nhiều. Chúng tôi sửa đổi một chút từ Gitflow như sau:

- *Loại bỏ branch hotfix*, mọi hotfix được tạo branch từ master và merge trực tiếp vào master

- *Branch release thực tế mới hơn master và được merge vào master.* Ngoại trừ hotfix, chúng tôi thực hiện release 2 lần / 1 tuần (thứ 3 và thứ 5).
 - trước ngày release, branch release được tạo mới từ branch master;
 - mọi chức năng đã sẵn sàng trên những feature branch được merge vào branch release;
 - thực hiện kiểm thử trên release branch;
 - nếu việc deploy thành công, branch release được merge vào branch master;
 - reset release branch.

Như vậy, branch master không chỉ là deploy-ready branch (đã sẵn sàng deploy) mà còn là deployed branch (đã được deploy). Có một số lý do chính để chúng tôi thực hiện như vậy:

- hotfix được thực hiện nhanh;
- luôn đảm bảo mọi commit tới branch master đã được deploy;
- do đặc thù của dự án, một số việc kiểm thử chỉ được thực hiện và xác định khả năng thành công sau khi deploy lên production server. Việc sử dụng branch release *ahead* (đi trước) branch master đảm bảo nếu việc deploy không thành công, một bản deploy đã ổn định sử dụng branch master nhanh chóng được triển khai.

So với Gitflow đã được chúng tôi sử dụng, branch strategy mới hoạt động tốt hơn nhiều bởi sự đơn giản và an toàn trong việc release liên tục. Ví dụ này cho thấy, dù Gitflow rất tuyệt để làm một branch strategy tham chiếu, nhưng tuỳ thuộc vào tình hình dự án, các nhóm phát triển có thể tự định nghĩa ra branch strategy của riêng mình. Bạn có thể cân nhắc tới một số yếu tố sau khi định nghĩa ra branch strategy phù hợp:

- *Chúng ta không nhất thiết phải sử dụng branch trừ khi việc phát triển trong nhóm cần đồng thời làm việc trên một tập source code nhất định.* Tôi đã thấy nhiều nhóm phát triển chỉ làm việc trên master branch bởi họ có source code được tổ chức tốt với ít sự phụ thuộc lẫn nhau trong công việc của mình. Tuy nhiên, cách làm này gần như chỉ hoạt động tốt với những nhóm phát triển nhỏ (2-3 thành viên). Điều quan trọng là chúng ta cần biết chính xác bài toán cũng nhu cầu của mình và có chiến lược đúng đắn.
- *Cấu trúc cây phân cấp branch một cách rõ ràng, tốt nhất là hình ảnh hoá nó và có những ví dụ cho một số trường hợp cụ thể.* Gitflow như trên là một

ví dụ tốt. Việc này nhằm đảm bảo những branch con được merge vào branch cha của chúng, tránh việc merge chéo giữa những branch không đúng.

- Đừng đặt tên hay tổ chức source code gây hiểu nhầm giữa cây phân cấp branch với cây phân cấp các thư mục trong source code.
- Đừng branch quá sâu. Một chức năng *sign-up* trong trang chủ của website có thể được phân cấp như master -> develop -> feature-homepage -> task-sign-up hoặc master -> develop -> feature-homepage-task-sign-up. Tuỳ từng trường hợp chúng ta nên sử dụng cây phân cấp thấp. Lý do là, việc merge branch sẽ được thực hiện ngược lại, task-sign-up -> feature-homepage -> develop -> master sẽ tốn nhiều thời gian, và có thể gây ra nhiều conflict hơn so với việc merge branch feature-homepage-task-signup -> develop -> master.
- *Luôn thực hiện việc tạo pull request và kiểm soát source code chặt chẽ trước khi một branch được merge vào branch cha.*
- *Thực hiện forward-intergration (tích hợp trước).* Ví dụ, thay vì merge branch feature-homepage -> develop và thực hiện việc test tích hợp trên branch develop; nhóm có thể merge develop -> feature-homepage và thực hiện việc test tích hợp trên branch feature-homepage. Việc này sẽ khiến branch develop ổn định, đảm bảo độ an toàn cần thiết, và cô lập được những bug đang được phát triển trên branch feature-homepage thay vì chỉ phát hiện ra lỗi sau khi được merge vào branch develop và phát sinh nhiều *dirty commit*.
- *Cân nhắc sử dụng fast-forward merge.* Git có một chức năng khá hay (mặc định) nhưng không nhiều người biết cách sử dụng, là fast-forward merge, cho phép chúng ta merge một branch A vào branch B và giữ luôn lịch sử commit vào A như những commit đã được thực hiện trên branch B. Thông thường các nhóm sẽ sử dụng **--no-ff** (no fast-forward merge) để luôn tạo một commit riêng biệt theo cách truyền thống. Tuy vậy, tận dụng fast-forward merge trong một số trường hợp để dễ dàng theo dõi lịch sử commit ngay trên branch cha cũng là một ý hay; ví dụ cho các hotfix được thực hiện nhanh.

Một điều quan trọng cần nhớ, *hãy giữ cho source code được đồng bộ và tích hợp liên tục vào các branch nhằm sớm phát hiện những lỗi tích hợp và có phương án sớm*. Chúng ta hoàn toàn có thể duy trì nhiều CI, mỗi CI cho một branch, và khi chúng

ta cảm thấy một chức năng đã ổn định ở một branch, branch đó ngay lập tức nên được merge vào main branch để giữ việc tích hợp trở nên liên tục.



Có nhiều tranh cãi xung quanh vấn đề này như:

Tại sao lại là Git mà không phải là SVN? Có phải vì branch trong SVN phức tạp hơn (nằm ở phần cuối cùng của hướng dẫn sử dụng, phần nâng cao) so với Git (xuất hiện ngay trong hướng dẫn cơ bản, như một phần công việc hàng ngày)? Không hẳn như vậy, nếu chúng ta có một branch strategy hiệu quả và phù hợp với những đặc thù của dự án và nhóm phát triển, thì SVN, SVC hay Git... đều mang lại hiệu quả như nhau. Điều khác biệt có chăng nằm ở:

- SVN ra đời khi phương pháp phát triển phần mềm truyền thống vẫn được coi trọng và việc phân rã source code được tính toán tỉ mỉ ngay từ bước thiết kế và có độ độc lập cao giữa những thành phần source code và các thành viên (*individual ownership – sở hữu cá nhân*). Branch strategy khi đó không được sử dụng nhiều bởi các nhóm phát triển phần mềm, bởi họ chọn phương pháp phân chia công việc cho các lập trình viên theo từng phần độc lập về source code để làm việc đồng thời. Cách làm này nói chung tiết kiệm thời gian hơn bởi nhóm phát triển không mất thời gian giải quyết các conflict.
- Git ra đời khi Agile đã là lựa chọn cần thiết, và với việc *collective ownership source code* và *continuous integration* thì việc sử dụng branch là không thể tránh khỏi và bỗng trở thành công việc hàng ngày của mỗi lập trình viên; do đó *hướng dẫn làm việc với branch* nằm ngay trong phần cơ bản của Git. Vậy nên, tôi thường khuyến nghị các nhóm phát triển sử dụng Git chỉ bởi *hướng dẫn làm việc với branch* khá đầy đủ; với những nhóm đã quen với công cụ và có branch strategy tốt thì Git hay SVN đều không phải vấn đề.

CODE REFACTORING

Code refactoring là hoạt động chỉnh sửa khiến source code dễ đọc hơn, được tổ chức khoa học hơn, và (có thể) có kiến trúc / cấu trúc tốt hơn nhưng không làm thay đổi hành vi của hệ thống về mặt chức năng.

Việc này giống như chúng ta sắp đặt lại hệ thống điện trong nhà theo một cách khoa học hơn nhưng vẫn đảm bảo giữ nguyên vị trí và chức năng của những công tắc, ổ cắm trên tường. Tôi muốn lấy ví dụ này để bạn hiểu rằng, những gì nhóm phát triển làm với code refactoring hoàn toàn nằm trong bức tường, nơi mà khách hàng hoàn toàn không nhìn hay cảm nhận được; nhưng lại rất quan trọng, đặc biệt trong dự án thực hành Agile. Tôi muốn có một ổ cắm điện ở vị trí này, sau 10 lần hoàn thành yêu cầu đó từ khách hàng, hệ thống dây điện chắc chắn sẽ chứa nhiều bất cập và không dễ bảo trì. Việc sắp đặt lại những dây điện này một cách hợp lý nhưng vẫn đảm bảo được chức năng hiện có giúp chúng ta sẵn sàng cho yêu cầu về một ổ cắm điện thứ 11. Và thật may là code refactoring thì thường không “tốn kém” và phức tạp như việc đục các bức tường để sắp đặt lại hệ thống dây điện. Vì vậy, chúng ta cũng có thể (và nên) làm việc này thường xuyên.

Thực hiện code refactoring như thế nào? Vấn đề này thậm chí là quá nhiều cho cả một cuốn sách. Những cách thức đơn giản nhất bạn có thể tham khảo tại <http://refactoring.com> của huyền thoại Martin Fowler. Tại đây bạn có thể tham khảo những kỹ thuật đơn giản nhất và dấu hiệu nhận biết một đoạn code có thể cần được refactor; từ chuyện đơn giản nhất như chuyển 2 đoạn code giống nhau thành một hàm đến sự liên kết giữa các đối tượng nhằm đảm bảo tính hướng đối tượng của chương trình. Trang web này thực sự hữu ích với những hệ thống thiết kế theo tư tưởng hướng đối tượng (phù hợp với đa số những mã nguồn hiện giờ), nhưng cũng rất tốt với những tư tưởng lập trình khác. Một chú ý hay là, đôi khi bạn thấy hướng dẫn refactor một đoạn code từ A sang B và nơi khác lại hướng dẫn refactor đoạn code từ B sang A. Điều này không mâu thuẫn, bởi *A hay B tốt hơn?* thì chỉ chính bạn mới có câu trả lời xác đáng trong ngữ cảnh của source code hiện tại. Tuy vậy, vẫn sẽ có những chuẩn chung để một đoạn code được coi là “tốt” hay “dở”; ví dụ, đặt tên biến là *a* là điều không chấp nhận được trong phát triển phần mềm (nơi duy nhất tôi thấy cách đặt tên biến này nên được sử dụng là trong những cuộc thi lập trình với source code ngắn và thời gian ganh đua tính bằng giây). Và hãy nhớ rằng, code refactoring không làm thay đổi hành vi của chức năng hay hệ thống; do đó, kết quả của việc kiểm thử phải không đổi.

Khi nào thực hiện code refactoring? Về lý thuyết, hãy thực hiện code refactoring bất cứ khi nào có thể. Trước khi commit, mỗi lập trình viên cần đọc lại những đoạn code mình đã viết và xem có thể cải tiến được không. Sau một thời gian, nhóm phát triển cần cùng nhau nhìn lại xem có thể cải tiến ở những điểm nào và cùng thực hiện code refactoring. Tuy nhiên, vấn đề không đơn giản như vậy.

Điều gì ngăn cản code refactoring? Đây là một câu hỏi rất thú vị. Tôi đã gặp rất nhiều nhóm thực hành Agile nhưng không bao giờ thực hiện code refactoring, với những lý do chính như sau:

- *Trình độ kém.* Khi nhóm phát triển không có hiểu biết sâu sắc về thiết kế thì đương nhiên những đoạn code ban đầu viết ra sẽ rất “dở”, nhưng quan trọng là họ hoàn toàn không biết rằng nó “dở”. Việc này càng nguy hại nếu không thực hiện code refactoring bởi nhóm sẽ mãi duy trì năng lực hiện có.
- *Chấp nhận.* Sau một thời gian dài, nhóm phát triển nhận ra có rất nhiều đoạn code “dở” nhưng nhóm vẫn chấp nhận bởi số lượng code “dở” là quá nhiều và có tư tưởng chấp nhận *sống chung với lũ*, hoặc nghĩ tới việc viết lại toàn bộ hệ thống.
- *Không có thời gian.* Đây là lý do khá xác đáng; bởi như tôi nói ở trên, khách hàng hoàn toàn không nhận được lợi ích trực tiếp từ code refactoring, nên khó thuyết phục họ trả tiền cho nhóm phát triển thực hiện code refactoring. Tuy vậy, việc lắp ổ điện thứ 11 mất 10 giờ, thay vì 2 giờ cho ổ điện thứ 1, thì cũng là tiền của khách hàng mà thôi (và điều này có thể nảy sinh nghi ngờ từ khách hàng rằng năng lực hoặc thái độ làm việc của nhóm đã kém đi).

Tuy vậy, những lý do này sẽ đẩy cả nhóm phát triển vào một vòng luẩn quẩn không hồi kết: *trình độ kém và sức ép thời gian* đưa ra những đoạn code “dở”, không thực hiện code refactoring khiến trình độ không được cải thiện, sau một thời gian dành *chấp nhận*, khiến *sức ép thời gian* càng lớn, không thể thực hiện code refactoring, và trình độ không được cải thiện... Và dự án, từ đam mê bỗng thành gánh nặng với nhóm phát triển, khiến động lực làm việc không còn đúng.

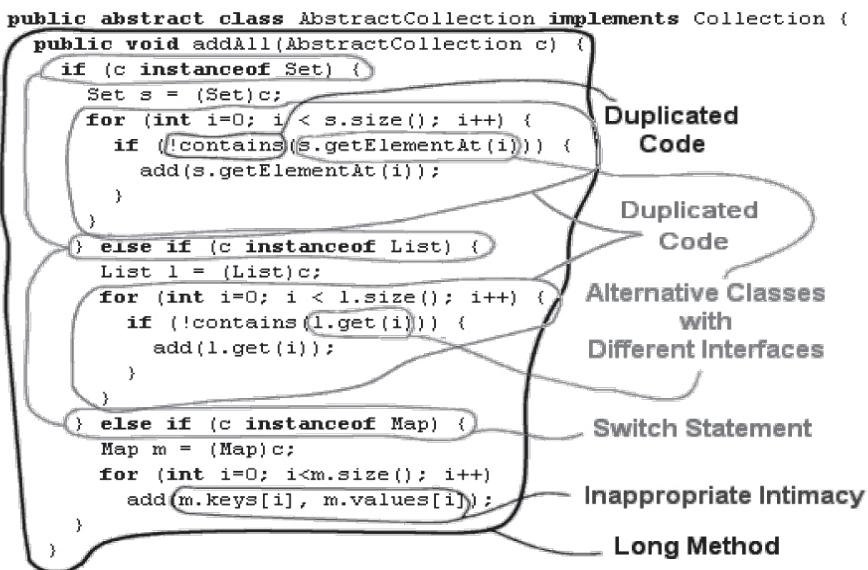
Vậy giải pháp là gì? Từ góc độ một lập trình viên, tôi cho rằng việc không thực hiện code refactoring là do lập trình viên; do họ không đủ đam mê và trách nhiệm cần thiết với đứa con tinh thần của mình; không khác một nhà văn viết ra những tác phẩm rỉ tai. Tuy vậy, người *lãnh đạo* trong dự án Agile cũng phải có trách nhiệm tạo ra những *khoảng lặng* về những chức năng cần bổ sung để nhóm phát triển thực hiện code refactoring. Việc này diễn ra càng đều đặn, trình độ và năng suất của lập trình viên càng cao bởi code refactoring chính là một cách nâng cao tay nghề và hiểu biết sâu sắc dựa trên những best practice giúp họ tốt hơn. Một ngày dành cho code refactoring hôm nay có thể giảm bớt 10 ngày phát triển buồn tẻ sau này.

Giải pháp cho source code đã quá “cũ” (được gọi là legacy code)? Khi chúng ta

động đâu cũng thấy vấn đề trong source code, *chấp nhận* hoặc *làm lại* từ đầu thường là giải pháp; tuy vậy, cả 2 phương án này đều rất tốn kém. Code refactoring có thể là một giải pháp tốt hơn:

- Sử dụng công cụ phân tích source code (tôi sẽ đề cập ở phần sau) để tìm ra những đoạn code “dở”;
- Nhóm phát triển cùng quét nhanh qua mã nguồn để đánh giá và tìm thêm những vấn đề;
- Ước lượng tổng thời gian cần cho code refactoring;
- Định nghĩa và lên kế hoạch việc kiểm thử. Việc này rất quan trọng vì code refactoring phải đảm bảo không thay đổi hành vi của chức năng và hệ thống. Lúc này automation test được ưu tiên bởi khối lượng kiểm thử nhiều. Không nên (thậm chí là nghiêm cấm) thực hiện code refactoring nếu không có kế hoạch kiểm thử tốt;
- Lên kế hoạch và thực hiện dần, từng phần. Thật tuyệt vời nếu chúng ta có toàn bộ thời gian để thực hiện; nếu không, hãy thực hiện từng phần song song với quá trình phát triển tiếp. Và hãy kiên nhẫn, chúng ta không thể thấy kết quả chỉ sau 1 vài ngày.

Thật ra, code refactoring là công việc rất đơn giản, đến mức người ta dễ dàng bỏ qua code refactoring để nghĩ tới architect refactoring hay structure refactoring. Nhưng theo tôi, khi thực hiện code refactoring tốt, những design pattern sẽ dần được hình thành và từ đó kiến trúc mới cũng sẽ được hình thành. Rất ít khi chúng ta cần tới architect refactoring; và tôi cũng không tham vọng giới thiệu trong cuốn sách cơ bản này.



Hình 5.9: Một đoạn source code có thể thực hiện code refactoring

Một trong những cuốn sách rất hay về code refactoring được viết bởi Michael C. Feather, một diễn giả từng xuất hiện tại Vietnam Scrum Gathering 2015, là *Working Effectively with Legacy Code*; tôi khuyến nghị bạn đọc cuốn sách này. Tại đây bạn có thể tìm thấy những chỉ dẫn hay để tìm ra những phần legacy code (những đoạn code bạn không cảm thấy tự tin khi sửa đổi) và cách thức để refactor chúng (viết test và sửa đổi).



Đoạn source code sau có nên được thực hiện code refactoring không? Nếu có, bạn sẽ thực hiện thế nào?

```

if ( (platform.toUpperCase().indexOf("MAC") > -1) &&
    (browser.toUpperCase().indexOf("IE") > -1) &&
    wasInitialized() && resize > 0 )
{
    // do something
}

```

KIỂM SOÁT CHẤT LƯỢNG SOURCE CODE

Việc kiểm soát chất lượng source code không phải là vấn đề mới, nhưng đặc biệt quan trọng trong những dự án Agile, ít nhất bởi những lý do sau:

- *Bất cứ thành viên nào* trong nhóm phát triển cũng có thể là *người thiết kế phần mềm*, điều này dẫn đến source code có khả năng phân mảnh lớn.
- *Tần suất cập nhật source code cao* và mức độ ảnh hưởng tới toàn hệ thống tương đối lớn.
- Tài liệu chi tiết ít khi được hình thành và bảo trì trong thời gian dài, dẫn tới phần lớn những nghiệp vụ, logic được đặt trong source code; việc kiểm soát chất lượng, đặc biệt là *độ dễ hiểu* của source code rất quan trọng.

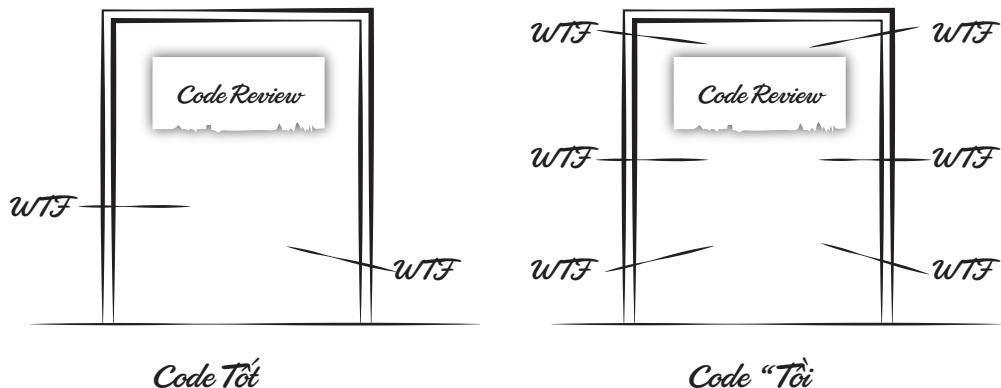
Do đó, việc đảm bảo chất lượng source code là việc không thể bỏ qua của mọi nhóm thực hành Agile. Code refactoring là một kỹ thuật nhằm đảm bảo chất lượng source code cũng như của toàn hệ thống đã được đề cập ở trên. Tuy vậy, trước khi thực hiện code refactoring, chúng ta cần biết chính xác những vấn đề gì đang tồn tại trong source code khiến chúng không được coi là *chất lượng và sạch sẽ (clean code)*. Có hai kỹ thuật để biết những vấn đề đang tồn tại:

- *Thủ công*. Source code được những thành viên trong nhóm phát triển xem lại, đánh giá. Thông thường, trước khi source code được merge vào branch master, các thành viên trong nhóm phát triển thường thực hiện việc đánh giá chéo (peer review). Kỹ thuật này rất hiệu quả bởi khi một thành viên trong nhóm có thể đọc hiểu được source code do một thành viên khác trong nhóm viết ra, source code đó đương nhiên là *chất lượng* bởi nó được hình thành bởi sự đồng thuận của 2 cá nhân, cũng như đảm bảo sự *dễ hiểu*. Điểm yếu của kỹ thuật này là thời gian và công sức, nhóm cũng cần có một chuẩn chung để các thành viên tham chiếu.
- *Tự động*. Một công cụ phân tích source code thực hiện phân tích dựa trên những luật được định nghĩa trước, nhằm tìm ra những đoạn code “tồi”, không đúng quy chuẩn như: câu lệnh quá dài, biểu thức quá phức tạp... Ưu điểm của kỹ thuật này là tốc độ và sự nhất quán bởi việc thực hiện tự động trên tập các luật được định nghĩa từ trước. Điểm yếu của kỹ thuật này là công cụ khó thực hiện được việc phân tích những đoạn code phức tạp.

Code review thủ công

Code được review thông qua những lập trình viên khác. Các lập trình viên thường truyền tai nhau một thước đo về chất lượng source code là *tần suất câu wtf (what the fuck? – cái gì vậy?)* trong khi thực hiện code review. Code review thủ công dù

mang lại lợi ích lớn về việc tìm ra bug nhưng lại phụ thuộc nhiều vào con người. Do đó, những nguyên tắc sau cần đảm bảo để có code review tốt.



Hình 5.10: Chất lượng source code được đánh giá bằng số lượng “wtf” / lần review

Review thật chậm, không review quá nhiều với thời lượng hợp lý. Theo nghiên cứu từ Cisco, một lập trình viên chỉ nên review từ 200-400 dòng code một lần trong không quá 1.5 giờ liên tục; và tốc độ không nên vượt quá 300-500 dòng trong 1 giờ. Bởi review code không phải là công việc “thú vị” như giải quyết vấn đề bằng việc viết ra source code; chất lượng review (số bug, code “dở” tìm được trong một khoảng thời gian) giảm đi rất nhanh. Và tất nhiên, chất lượng là điều chúng ta muốn khi thực hiện code review.

Code Review cần được coi như một phần hoàn thành của công việc và cần có DoD rõ ràng. Tiêu chí như *tăng chất lượng, tìm thấy nhiều bug hơn...* không thực sự rõ ràng; *giảm 20% số ticket gửi tới bộ phận hỗ trợ kỹ thuật* là một tiêu chí tốt hơn.

Nói về code. Không thể phủ nhận rằng, source code là sản phẩm của mỗi thành viên trong nhóm; nhưng khi thực hiện code review, nhóm đang nói về source code, về những cải tiến, không phải về hiểu biết và trình độ của mỗi thành viên. Sẽ thật không hay nếu một lập trình viên có kinh nghiệm hơn nói rằng *tôi có kinh nghiệm hơn, và theo tôi, đoạn code này phải thế này*.

Ghi chú với những trích dẫn. Khi bình luận về một đoạn code, người review nên ghi chú những nguyên tắc như OOP: abstraction, Design: Open-Close... Những bình luận vui cũng giúp hướng sự chú ý vào source code thay vì người viết ra source code, như *method này dài hơn bài báo về Messi tối qua rồi...*

Đưa ra những bình luận mang tính mở và tích cực. Ví dụ, *method này dài hơn 50 dòng* (quy chuẩn là 50 dòng) sẽ không tốt bằng có lý do gì để *method này dài hơn quy chuẩn không?*

Code review không chỉ tìm ra source code “tối”. Một đoạn source code hay và người review có thể học được từ đó, cũng nên để lại một bình luận tốt như *cả hình thức và chức năng, hoàn hảo như Ronaldo* vậy.

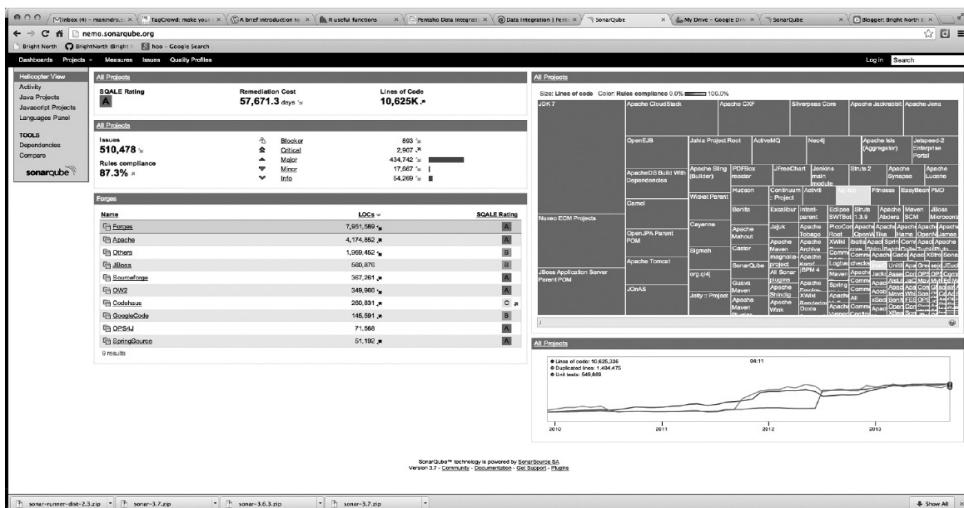
Code review không phải là nơi giải quyết vấn đề. Hãy mặc định rằng giải pháp đã được thống nhất, chúng ta chỉ nói về source code đã thực hiện đúng và tốt theo giải pháp đó không. Tranh luận về giải pháp vào lúc này không giúp ích gì, chỉ khiến góc nhìn đánh giá source code bị sai.

Code review tự động

Có rất nhiều công cụ giúp thực hiện việc phân tích source code tự động trên thị trường, bạn có thể dễ dàng tìm kiếm trên Internet. Một trong những công cụ phổ biến được sử dụng hiện nay là SonarQube (<http://www.sonarqube.org/>), bởi những ưu điểm:

- *Miễn phí, mã nguồn mở.*
- *Độc lập với platform, ngôn ngữ.* SonarQube có thể phân tích được nhiều ngôn ngữ khác nhau, thậm chí với một project có nhiều ngôn ngữ, thông qua những plugin có thể được cài đặt thêm.
- *Tích hợp với CI.* SonarQube có thể chạy độc lập hoặc được tích hợp với các công cụ CI trên thị trường, nên *chất lượng source code* có thể được coi là một tiêu chí để vượt qua việc tích hợp liên tục vào hệ thống chính.

Những gì mà bạn quan tâm có thể tìm thấy tại SonarQube.



Hình 5.11: Giao diện SonarQube khi phân tích 1 ứng dụng

Thường thì việc áp dụng những công cụ phân tích source code như SonarQube sẽ rất dễ dàng với những dự án mới, với source code được xây dựng từ đầu. Nhưng nếu bạn làm việc với legacy code thì sao? Ngay sau cài đặt SonarQube và áp dụng một số chuẩn như FxCorp, OCLint..., bạn sẽ ngay lập tức bị sốc với những thông tin: *số lượng dòng code trùng lặp là 10.000, cần 50 ngày để thực hiện code refactoring...* bởi rất nhiều phần code được coi là *không đạt chuẩn*. Những phần này được coi là *technical debt* (*nợ kỹ thuật*): trong quá trình phát triển, chúng ta sinh ra nhiều đoạn code thừa, nhiều đoạn code trùng lặp, nhiều method quá dài... những phần này phải được sửa đổi để đảm bảo việc phát triển về sau; nếu chúng ta không thực hiện code refactoring từ bây giờ, chúng ta sẽ phải trả giá trong tương lai. Đó chính là lý do đây được coi là *nợ*.

Khi bạn nhìn thấy phần nợ kỹ thuật quá lớn với legacy code, đừng sợ hãi. Nếu bạn thấy rằng những phần source code này vẫn đang hoạt động tốt, không có nhu cầu sửa đổi trong tương lai, thì cũng không cần lo lắng về việc trả nợ. Ngược lại, đây chính là chỉ dẫn để bạn biết những gì nhóm cần làm và chuẩn bị cho phương án thực hành code refactoring; dần dần, từng phần một. Và điều quan trọng là, *hãy giữ cho phần nợ kỹ thuật tăng chậm nhất có thể bằng cách áp dụng những chuẩn thiết kế, lập trình ngay từ bây giờ*. Những ai đã vay nợ với lãi suất cao sẽ hiểu nợ đáng sợ như thế nào. Và đó chính là lý do *tăng thêm người vào giai đoạn cuối của dự án* sẽ chỉ làm dự án thêm tồi tệ vì cả nợ gốc và lãi suất đều có xu hướng tăng lên rất nhanh, chuyện mất kiểm soát dẫn đến vỡ nợ rất dễ xảy ra.

Bạn đánh giá và bình luận thế nào về đoạn source code sau với đồng nghiệp?

```
public abstract class AbstractCollection implements Collection {  
    public void addAll(AbstractCollection c) {  
        if (c instanceof Set) {  
            Set s = (Set)c;  
            for (int i=0; i < s.size(); i++) {  
                if (!contains(s.getElementAt(i))) {  
                    add(s.getElementAt(i));  
                }  
            }  
        } else if (c instanceof List) {  
            List l = (List)c;  
            for (int i=0; i < l.size(); i++) {  
                if (!contains(l.get(i))) {  
                    add(l.get(i));  
                }  
            }  
        } else if (c instanceof Map) {  
            Map m = (Map)c;  
            for (int i=0; i<m.size(); i++)  
                add(m.keys[i], m.values[i]);  
        }  
    }  
}
```

The code is annotated with several design smell labels:

- Duplicated Code (appears twice)
- Alternative Classes with Different Interfaces
- Switch Statement
- Inappropriate Intimacy
- Long Method

TDD – LẬP TRÌNH HƯỚNG KIỂM THỬ

TDD (Test-Driven Development – lập trình hướng kiểm thử) là một trong những kỹ thuật quan trọng bậc nhất của Agile. TDD được hiểu đơn giản là: Định nghĩa phương pháp, các test case cho các đoạn code hoặc chức năng; đoạn code hoặc chức năng được coi là đúng nếu vượt qua được những test case này. Hoàn toàn không có gì đặc biệt? Có lẽ là như vậy.

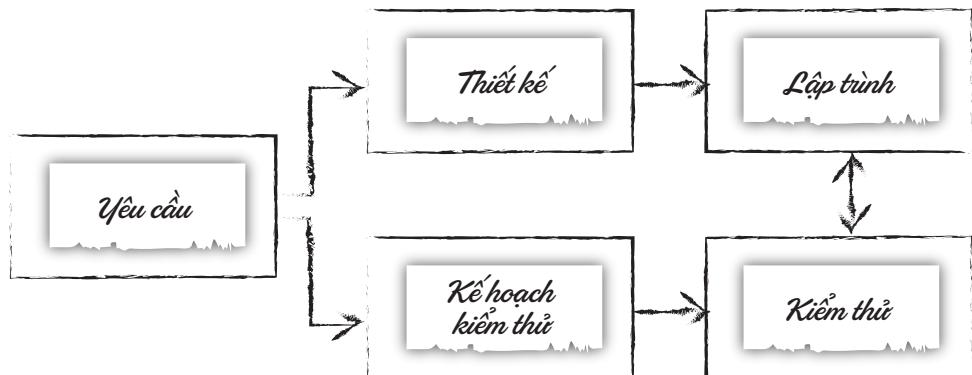
Quay lại với phương pháp phát triển phần mềm truyền thống, chúng ta có mô hình như sau:



Hình 5.12: Kiểm thử trong phương pháp phát triển phần mềm truyền thống

Các lập trình viên thực hiện việc viết ra source code, tester thực hiện kiểm thử những chức năng trong giai đoạn cuối cùng. Công việc này thường được gọi là *kiểm thử hộp đen (blackbox testing)*, tức là chúng ta hoàn toàn không quan tâm tới source code, chỉ thực hiện những chức năng như một người dùng thông thường. Nếu phát hiện ra lỗi, các lập trình viên sẽ là người thực hiện việc sửa đổi và tester thực hiện việc kiểm thử lại.

Điểm yếu của phương pháp phát triển phần mềm truyền thống có thể được mô tả:



Hình 5.13: Kiểm thử trong phương pháp phát triển phần mềm truyền thống

Vấn đề chính gặp phải là xuất phát từ một yêu cầu, những người thiết kế và lập trình hoàn toàn có thể hiểu theo cách khác với một thành viên hoặc nhóm kiểm thử. Điều gì xảy ra nếu vào giai đoạn cuối của việc phát triển phần mềm, giai đoạn kiểm thử, yêu cầu phần mềm không được đáp ứng? Toàn bộ công đoạn phải bắt đầu lại từ đỉnh thác. Điều này hoàn toàn có thể xảy ra, vì theo phương pháp phát triển phần mềm truyền thống, từ sau khi yêu cầu phần mềm được xác định, việc phát triển và kế hoạch kiểm thử nói chung đi trên hai con đường riêng biệt. Bất cứ khi nào yêu cầu chức năng bị thay đổi, có ít nhất 3 nơi cần phải được cập nhật (và thường là độc lập): *tài liệu yêu cầu, thiết kế / source code, test case*. Bởi chúng được duy trì trên những thành phần riêng lẻ và cập nhật độc lập, không phương pháp nào đảm bảo được tính đồng nhất. Gắn kết những thành phần này với nhau, và tốt nhất khi chỉ duy trì một thành phần – source code – là tư tưởng của TDD.

Mỗi khi một chức năng mới được phát triển, nhóm thực hiện việc mô tả chức năng thông qua những test case cụ thể, tương ứng với những use case của người dùng trong thực tế. Chức năng được coi là chạy đúng nếu vượt qua những test case này. Mỗi khi yêu cầu chức năng bị thay đổi, những test case này sẽ được cập nhật và khiến source code hiện tại không thể vượt qua được; nhóm phát triển sẽ buộc phải chỉnh sửa source code để đảm bảo source code tiếp tục thỏa mãn những test case cũ. Cách làm này mang đến hiệu quả tuyệt vời khi ràng buộc 3 thành phần trên vào một nơi duy nhất: Yêu cầu chức năng, việc kiểm thử và source code nhanh chóng bị phát hiện ra sự không hợp lý khi không vượt qua được các test case.

Phương pháp này được gọi là *ATDD (Acceptance-Test-Driven Development)* hay *BDD (Behavior-Driven Development)* và được thực hiện tốt nhất khi các test case là một phần của source code và được thực hiện tự động. Ví dụ, chức năng Đăng nhập của một trang web có thể được mô tả theo hành vi người dùng như sau:

- Nhập “admin” vào ô username
- Nhập “1234” vào ô password
- Nhấn nút “Login”
- Một popup hiện ra “Username / password không đúng”
- Nhập “123456” vào ô password
- Nhấn nút “Login”
- Trang web chuyển tới trang profile.html

Test case này có thể được viết thành 1 đoạn Javascript:

```
tester.let("username").text("admin");
tester.let("password").text("1234");
tester.let("login").click();
assert(POP_UP).text("Username / password không đúng");
tester.let("password").text("1234");
tester.let("login").click();
assert(LOCATION).href("profile.html");
```

Đoạn Javascript trên hoàn toàn tương ứng với mô tả use case ở trên? Chắc chắn rồi. Vì chúng ta không cần yêu cầu từ use case ở trên nữa, vì đoạn Javascript đó chính là tài liệu yêu cầu, kiêm test case. Khi chúng ta thay đổi yêu cầu từ việc hiển thị "*Username / password không đúng*" sang "*Thông tin không chính xác*", việc kiểm thử ngay lập tức không thành công; là dấu hiệu cho thấy source code cần phải được cập nhật.

Nếu ví dụ trên cho thấy việc kiểm thử được thực hiện ở mức chức năng, theo quan điểm người dùng (nên gọi là ATDD) thì TDD cũng hoạt động theo cách thức tương tự ở mức module với trái tim là unit test. Unit test chính là test case để kiểm thử một đoạn code, function, class, module... thực thi được đúng chức năng mong muốn. Ví dụ, chúng ta muốn viết một function tính thuế thu nhập cá nhân với đầu vào là thu nhập của cá nhân trong tháng như sau:

```
function calculatePersonalIncomeTax(long income) {
    //...
}
```

Làm sao để biết function này chạy đúng? Hãy định nghĩa việc kiểm thử:

```
Assert.equals(calculatePersonalIncomeTax(100000), 0);
Assert.exception(calculatePersonalIncomeTax(-1), InvalidParameter);
```

Như vậy, nếu function *calculatePersonalIncomeTax* vẫn thực hiện tốt với tham số -1 thì function không đảm bảo yêu cầu (vì không thể có thu nhập âm). Bất cứ khi nào công thức tính thuế thu nhập cá nhân bị thay đổi, những test case trên sẽ được cập nhật theo công thức mới, function *calculatePersonalIncomeTax* tiếp tục không thoả mãn test case; và nhóm phát triển biết họ phải sửa đổi function *calculatePersonalIncomeTax*.

Lợi ích của TDD

Bạn có thể đã hình dung ra những lợi ích của TDD qua những ví dụ trên. Trong

thực tế, TDD còn mang lại nhiều lợi ích hơn thế, và tôi coi đó là một trong những phương pháp quan trọng bậc nhất trong tập hợp những practice của Agile.

Luôn giữ hệ thống được cập nhật. Lợi ích lớn nhất mà TDD mang lại chính là khả năng tài liệu hóa và khiến mọi thành phần của hệ thống không bao giờ lỗi thời. Bất cứ khi nào yêu cầu phần mềm bị thay đổi, nơi duy nhất chúng ta phải cập nhật là testcase. Nếu xuất hiện những test case bị fail, nhóm phát triển ngay lập tức biết rằng sản phẩm đang không thoả mãn nhu cầu hiện tại, và họ biết phải chỉnh sửa chức năng, thành phần nào.

Việc phát triển tự tin hơn. Chừng nào tất cả các test case đều được thoả mãn, nhóm phát triển biết rằng hệ thống đang chạy đúng.

Việc phát triển nhanh hơn. Bằng việc viết test cho từng function, module, chức năng; những test case fail sẽ cho thấy phần nào của hệ thống (cụ thể tới từng function, module) đang thực hiện không đúng như kỳ vọng giúp nhóm phát triển cô lập và hạn chế khu vực cần sửa đổi. Thay vì phải debug qua từng dòng code, nhóm phát triển biết chính xác nơi nào nên đặt break point.

Giúp source code dễ đọc hơn. Đọc hiểu source code chưa bao giờ là một vấn đề dễ dàng. Thay vì phải mất hàng giờ đọc hiểu một function, module... thực sự làm gì, lập trình viên chỉ cần đọc test case (thường là dễ hiểu hơn) để biết cách thức sử dụng cũng như mục đích, kết quả thực sự trong từng trường hợp cụ thể.

Nâng cao chất lượng thiết kế. Bất cứ thành phần nào của hệ thống không thể test được, thành phần đó không đáng tin; và đồng nghĩa với một thiết kế tồi. Một trong những tiêu chuẩn thiết kế là decoupling để tăng sự độc lập, giảm phụ thuộc giữa các thành phần (nhằm trừu tượng hoá, tái sử dụng...); việc nhóm phát triển không thể viết test cho những thành phần riêng lẻ cho thấy các thành phần đang không thoả mãn decoupling; và đó đương nhiên là một việc không tốt. Tất nhiên, decoupling chỉ là một ví dụ đơn giản; tôi tin là những nhóm phát triển sẽ nhanh chóng nhận ra lợi ích của TDD với những tư tưởng thiết kế khác.

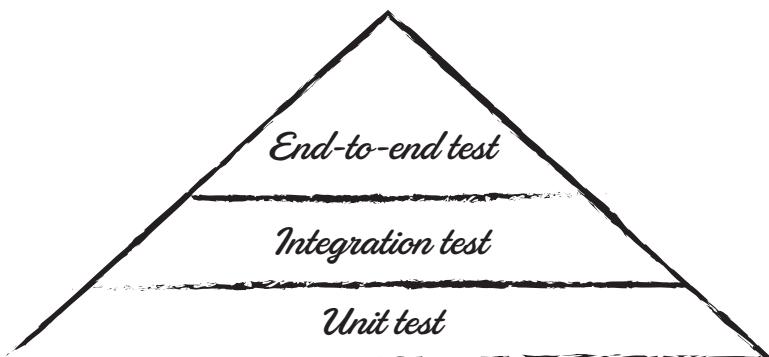
Nhận biết việc code refactoring. Mỗi khi phát triển hoặc sửa đổi một chức năng, nhóm phát triển sẽ ngay lập tức biết những ảnh hưởng trên toàn hệ thống đã được tạo ra; từ đó cũng xuất hiện những dấu hiệu cho thấy những thành phần có thể thực hiện code refactoring.

Xây dựng test

Những lợi ích trên sẽ chỉ được thấy rõ ràng khi nhóm phát triển áp dụng đúng TDD, và tất nhiên, giữ cho trái tim của TDD – test – được tốt. Ở đây, tôi chỉ đưa ra một số chỉ dẫn cơ bản.

Định nghĩa test trước. Trước khi viết code, hãy viết unit test; trước khi bắt đầu chức năng, hãy viết acceptance test. Rất nhiều lập trình viên chỉ viết test sau khi đã viết code hoặc hoàn thành chức năng cụ thể; cách làm này cũng tốt, ít nhất là còn hơn khi không viết test. Nhưng khi đó, test chỉ mang ý nghĩa kiểm tra (check / verification) và mất hẳn ý nghĩa tài liệu (document), và đương nhiên, đó không phải là tư tưởng của test driven hay TFD (Test-First Development).

Test phải nhỏ, và thực hiện cho từng phần nhỏ. Tốt nhất là tổ chức test theo testing pyramid với nhiều unit test và ít integration test, end-to-end test hơn. Nếu chúng ta muốn cô lập vấn đề, chúng ta buộc phải cô lập các test cho từng đoạn code, function, module... nhỏ. End-to-end test bị fail cho thấy một chức năng không chạy đúng dưới góc độ người dùng; nhưng để biết chính xác function nào đang được thực thi không đúng (trong hàng chục function phục vụ cho chức năng đó) thì chỉ unit test mới có thể giúp được.



Hình 5.14: Tháp testing - Xây dựng nhiều unit test

Test cần chạy nhanh. Nếu test chạy chậm, thời gian để nhận biết vấn đề lâu, và nhóm phát triển cũng không thấy hiệu quả thực sự. Việc xây dựng cấu trúc test với các test nhỏ cũng là một chỉ dẫn để test chạy nhanh.

Luôn ưu tiên sửa lỗi khi test fail. Bất cứ khi nào nhóm phát hiện ra có test case fail, việc fix bug phải có độ ưu tiên cao nhất. Việc này giúp duy trì hệ thống chạy ổn định. Chúng ta phát triển chức năng mới để làm gì khi hệ thống hiện tại đang có lỗi?



Sử dụng IDE và tạo Unit test cho User Story *Là giáo viên, tôi muốn đánh giá học lực của sinh viên dựa vào điểm số của họ*, theo các bước sau:

1. Đưa ra các trường hợp kiểm thử tương ứng với điểm số: 8: giỏi, 7: khá, -1: lỗi...
2. Định nghĩa phương thức đánh giá học lực với tham số đầu vào là điểm số. Ví dụ:

```
String getGrade(float grade) {  
}
```
3. Viết test gọi tới phương thức này với những test case kể trên.
Ví dụ:

```
void testGrade_Distinction {  
    Assert.equals("giỏi", getGrade( 8.0f ));  
}
```
4. Cài đặt phương thức getGrade và liên tục kiểm tra thông qua các test đã tạo, cho đến khi mọi test case được vượt qua.

LẬP TRÌNH CẶP

Lập trình cặp (pair programming) hay lập trình theo cặp là một kỹ thuật được nói đến nhiều trong phương pháp XP: *hai lập trình viên cùng làm việc trên một không gian, công cụ (máy tính, màn hình, bàn phím...) trong cùng một thời điểm, cùng nhau tạo ra một đoạn source code.*

Lập trình cặp thường được nói đến rất nhiều nhưng lại ít được sử dụng, giống như người ta nói về chiếc xe dẫn động 2 cầu vậy. Một chiếc xe dẫn động 2 cầu, truyền chuyển động đến 4 bánh, chắc chắn cung cấp sức kéo tốt hơn, an toàn hơn nhưng lại đắt hơn một chiếc xe dẫn động 1 cầu chỉ truyền chuyển động đến hai bánh trước hoặc sau. Ai cũng muốn sở hữu một chiếc xe dẫn động 2 cầu nhưng lại lo ngại về chi phí vì không biết nó tốt hơn bao nhiêu so với một chiếc xe dẫn động 1 cầu. Lập trình cặp cũng vậy, chúng ta dễ dàng đồng ý với nhau rằng 2 lập trình viên cùng làm việc để tạo ra một đoạn source code thì tốt hơn một người, ít nhất bởi những lý do sau:

- 2 lập trình viên có 2 nền tảng kinh nghiệm khác nhau, cung cấp 2 cách tiếp cận vấn đề khác nhau;
- vì vậy, họ cung cấp nhiều cách giải quyết vấn đề khác nhau;
- đoạn code được 2 lập trình viên viết ra, rà soát có thể hạn chế lỗi.

Nhưng quyết định thực hiện lập trình cặp trong thực tế lại là một câu chuyện khác. Không có một con số nào cho biết chính xác rằng nhiều cách giải quyết vấn đề, việc rà soát lỗi khi thực hiện lập trình cặp so với lập trình đơn sẽ mang lại lợi ích ra sao; đây vẫn là *ẩn số*. Trong khi, xét về chi phí trực tiếp, lập trình cặp sử dụng chi phí gấp đôi so với lập trình đơn; ai cũng dễ dàng tính ra được. Việc cân nhắc giữa việc tăng 100% chi phí với một ẩn số của lợi ích mang lại (có thể là 0%, 100%, 1000%)... khiến không nhiều tổ chức lựa chọn thực hiện lập trình cặp; họ luôn chờ đợi một con số chính xác để biết lập trình cặp có mang lại hiệu quả thực sự hay không. Và các tổ chức nghiên cứu về Agile lại gặp khó trong việc cung cấp con số này bởi không có đủ lượng dữ liệu từ số ít những tổ chức áp dụng. Bài toán con gà – quả trứng hình thành; và lập trình cặp trở thành vấn đề *ai cũng nói tới, không nhiều người hiểu, rất ít người làm dù đã ra đời hơn 2 thập kỷ* (năm 1992, trong báo cáo của Larry Constantine sau khi ghé thăm công ty sản xuất trình biên dịch *Hi lập trình viên ngồi trước một máy tính! Chắc chắn, chỉ một người thực sự viết code bằng bàn phím, lập trình viên kia theo dõi từ phía sau*). Vì vậy, rất ít tổ chức thực hiện triệt để lập trình cặp, hầu hết theo xu hướng *sử dụng xe dẫn động 2 cầu cho địa hình phức tạp*; tức là chỉ thực hiện lập trình cặp với những vấn đề phức tạp. Nhưng thế nào là vấn đề phức tạp để cần thực hiện lập trình cặp? Thật khó định nghĩa rõ ràng. Thông thường, khi gặp một vấn đề phức tạp, khó giải quyết bởi một lập trình viên, một số lập trình viên sẽ ngồi lại với nhau, thảo luận giải pháp rất chung chung và để một người viết code. Cách làm này có thực sự hiệu quả không?

Một số ít báo cáo dè dặt công bố kỹ thuật lập trình cặp giúp giảm 15% số bug. Nhưng theo tôi, lợi ích của lập trình cặp còn nhiều hơn vậy.

Trong kỹ thuật lập trình cặp, 2 lập trình viên thường đảm nhận 2 vai trò khác nhau, và thường được thay đổi thường xuyên:

- **Điều khiển (driver):** Người viết source code, hiện thực hoá ý tưởng, giải pháp thành những đoạn source code thực sự.
- **Định hướng (navigator):** Người theo dõi, đánh giá, giám sát nhằm đảm bảo từng dòng code được người kia viết ra không chỉ đi đúng hướng về giải pháp mà còn là *clean code*.

Lập trình cắp mang tới những lợi ích sau:

Chất lượng. Đoạn code được xem xét lại ngay trong khi được viết ra, nên dễ hiểu tại sao lượng bug được hạn chế.

Giải pháp, thiết kế tốt. Trước khi viết code, 2 lập trình viên cung cấp những góc nhìn, giải pháp khác nhau, trao đổi nhằm tìm ra giải pháp, thiết kế tốt nhất.

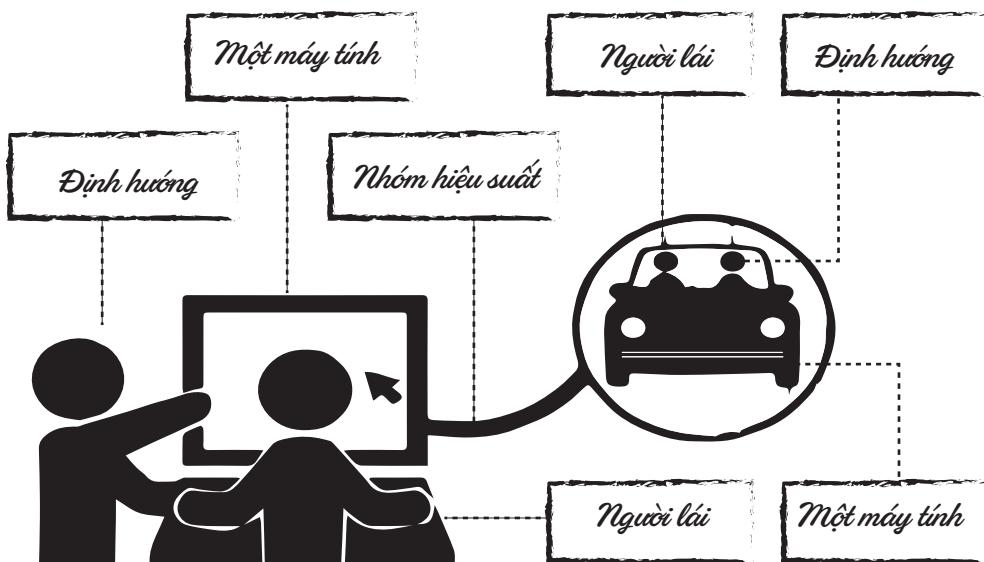
Học hỏi. Thông qua việc trao đổi giải pháp và hiện thực hoá giải pháp đến từng dòng code cùng nhau, 2 lập trình viên luôn học hỏi được từ nhau, bất kể họ ở trình độ ra sao:

Trình độ	Thấp	Cao
Thấp	<p>Lập trình cắp tỏ rõ ưu thế bởi lập trình viên ở trình độ thấp có khả năng tạo ra nhiều bug hơn.</p> <p>Được theo dõi bởi một lập trình viên khác giúp họ cẩn thận và tích luỹ nhiều kinh nghiệm.</p>	<p>Lập trình viên ở trình độ thấp học hỏi được từ cách giải quyết vấn đề, giải pháp tới cách hiện thực hoá thành đoạn code từ lập trình viên có trình độ cao hơn.</p> <p>Lập trình viên ở trình độ cao hơn học hỏi thêm từ cách nhìn mới mẻ, để tránh mắc phải bẫy kinh nghiệm. Họ cũng học được cách viết mã tốt hơn để một lập trình viên ở trình độ thấp hơn cũng hiểu được.</p>
Cao		<p>Càng ở trình độ cao, lập trình viên càng có những góc nhìn khác nhau nhằm cung cấp cách giải quyết vấn đề khác nhau.</p> <p>Lập trình cắp giúp họ tiến bộ hơn.</p>

Mặc dù trên lý thuyết, lập trình cắp mang lại lợi ích tuyệt vời. Nhưng thực tế, ngoài trở ngại về kinh tế, các tổ chức còn gặp phải trở ngại về con người khi những lập trình viên ở trình độ cao thường không có xu hướng “cắp” với lập trình viên ở trình độ thấp hơn mình, bởi họ cho rằng vừa viết code, vừa diễn giải cho một lập trình viên kém hơn thường tiêu tốn thời gian một cách không cần thiết. Nhưng dưới góc độ tổ chức, đây là cách nhanh nhất để họ có hai lập trình viên ở trình độ cao. Dưới góc độ cá nhân, lập trình viên ở trình độ cao thường mắc phải *bẫy kinh nghiệm* khiến họ giải quyết vấn đề theo lối mòn đã có; bằng việc “cắp” với lập trình viên ở trình độ thấp hơn, cách nhìn của họ mới mẻ hơn và cũng học được nhiều

hơn. Tôi khuyến nghị bạn tìm hiểu thêm về lập trình cặp với một bài viết rất tuyệt vời tại địa chỉ http://www.jamesshore.com/Agile-Book/pair_programming.html

Theo kinh nghiệm của tôi, nếu bạn chưa tin vào sức mạnh của lập trình cặp, hãy đến với các cuộc thi như *ACM ICPC (ACM International Collegiate Programming Contest)* dành cho những sinh viên Đại học. Trong 5 giờ, một đội 3 người sẽ phải lập trình để giải khoảng 10 bài toán chỉ với duy nhất một bộ máy tính. Và những đội xuất sắc nhất thường phải thành thạo lập trình cặp, khi một người lập trình, một người theo dõi để chắc chắn việc lập trình chính xác như những gì đã đề ra, đồng thời chuẩn bị các bộ test để kiểm thử source code. Bởi chỉ một sai sót có thể khiến họ tụt lại 20 phút so với đội khác và ảnh hưởng nhiều tới thứ hạng (mỗi lần nộp giải pháp không đúng, nhóm bị phạt 20 phút). Đó cũng là lý do những sinh viên Đại học đã trải qua những cuộc thi như ACM ICPC rất hiểu giá trị và thực hành tốt lập trình cặp, cũng như TDD.



Hình 5.15: Lập trình cặp

Bất cứ một kẻ khờ nào đều có thể viết code để máy tính hiểu được. Lập trình viên giỏi viết code để con người có thể hiểu. (Any fool can write code that a computer can understand. Good programmers write code that humans can understand.) Martin Fowler, 2008.

RETROSPECTIVE

Tôi cho rằng đây là *công cụ quan trọng hàng đầu* trong tập công cụ của phương pháp Agile. Một nhóm không bao giờ được coi là hiểu và thực hành Agile nếu bỏ qua công cụ này.

Retrospective là một trong 5 sự kiện được định nghĩa trong Scrum đã được giới thiệu tại Chương 3; nhưng Retrospective được sử dụng bởi hầu hết những nhóm thực hành Agile bất kể họ có thực hành Scrum hay không bởi sự tuyệt vời mà phương pháp này mang lại. Chính vì vậy tôi muốn dành thêm một phần trong chương này để nhấn mạnh về tầm quan trọng và một số kỹ thuật thực hành Retrospective phổ biến.

Thay vì đề ra một quy trình phát triển phần mềm chặt chẽ, Agile cung cấp phương pháp hỗ trợ các nhóm tìm ra quy trình cho chính mình dựa trên nguyên tắc cải tiến liên tục. Và Retrospective chính là bộ óc của việc cải tiến liên tục. Nhắc lại, Retrospective là hoạt động nhìn lại những gì đã xảy ra, phân tích nguyên nhân khách quan nhằm tìm ra những điểm cải tiến giúp lần thực hiện tiếp theo được tốt hơn; và mang lại hiệu quả cao hơn về mặt kinh tế:

Trao quyền cho các nhóm. Các nhóm cảm thấy tự tin vì được trao quyền và cảm nhận rõ tổ chức quan tâm tới những hành động cũng như kỳ vọng sự phát triển của nhóm.

- *Môi trường tích cực.* Tập trung vào việc tìm hiểu, học hỏi từ những gì đã xảy ra thay vì đơn thuần chỉ trích dựa trên kết quả.
- *Liên tục cải tiến.* Bằng cách tìm ra những vấn đề có thể cải tiến, nhóm luôn học hỏi, năng động và cho kết quả ngày càng tốt hơn.
- *Cam kết.* Do những hành động tiếp theo được cả nhóm thống nhất và đồng ý, sự cam kết thực hiện bởi nhóm và các thành viên cao hơn.

Bởi Retrospective là một công cụ tuyệt vời nhưng ít nhóm ở Việt Nam thực hiện tốt – một phần vì không biết cách thực hiện, một phần vì không thấy hiệu quả trực tiếp; tôi muốn giới thiệu một số phương pháp Retrospective hiệu quả.

Đặt câu hỏi

Đặt câu hỏi là phương pháp đơn giản nhất, hãy sử dụng phương pháp này cho những nhóm mới thực hiện Retrospective.

Có 2 câu hỏi hay được sử dụng là:

- Điều gì đã được thực hiện tốt?
- Điều gì đã có thể thực hiện tốt hơn?

Dựa trên việc tổng hợp câu trả lời cho 2 câu hỏi này, một câu hỏi khác nên được đặt ra là *Hành động tiếp theo là gì? (để cải tiến)*. Việc đặt câu hỏi mở sẽ khiến nhóm phát triển tự tìm ra giải pháp, cam kết thực hiện hành động.

Đã làm tốt	Có thể tốt hơn	Hành động
Xóa code Vẽ lại Workflow	Kiểm thử Đăng nhập Định nghĩa AC cho thanh toán	PO dành nhiều thời gian hơn

Hình 5.16: Retrospective với phương pháp đặt câu hỏi

ScrumMaster hay người điều hành cuộc họp tuyệt đối không được liệt kê những hành động tiếp theo và yêu cầu mọi người làm theo.

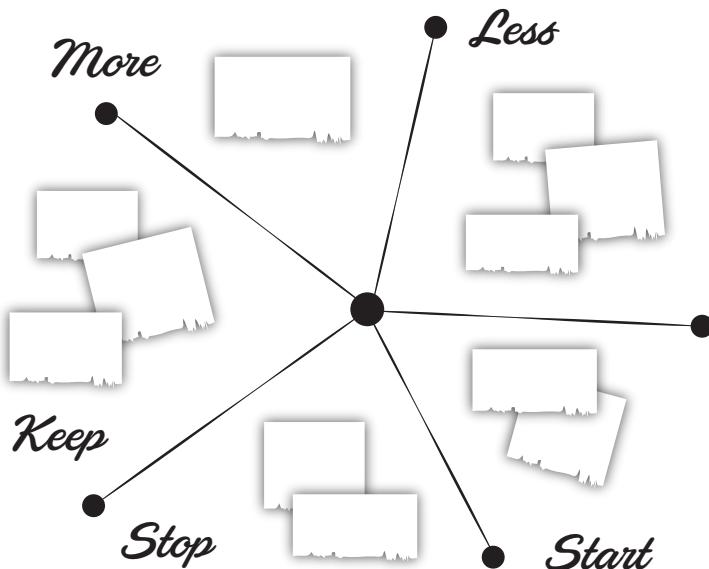
Starfish

Phương pháp Starfish cụ thể hơn phương pháp đặt câu hỏi. Phương pháp này phù hợp với những nhóm phát triển không đạt được sự ổn định trong một vài Iteration.

Phương pháp Starfish chia một vòng tròn thành 5 phần:

- *Stop.* Những hoạt động không mang lại giá trị cho nhóm hoặc cho khách hàng cần được loại bỏ.

- *Less.* Những hoạt động mang lại giá trị không tương xứng với nỗ lực nhưng không thể loại bỏ, cần được thực thi ít hơn.
- *Keep.* Thông thường là những hoạt động đang đạt giá trị tốt, cần được duy trì.
- *More.* Những hoạt động mang lại giá trị cao cần được tập trung nhiều nỗ lực hơn
- *Start.* Những hoạt động chưa được thực hiện, nhưng nhóm phát triển muốn thực hiện



Hình 5.17: Retrospective với phương pháp Starfish

Phương pháp này hay hơn phương pháp đặt câu hỏi ở điểm tập trung và rõ ràng hơn. Khi từng ý kiến được điền đầy đủ vào 5 phần này, những hoạt động tiếp theo cũng đồng thời xuất hiện. Khi chúng ta thấy nhóm phát triển có năng suất không ổn định trong một Iteration, thông thường là do nhóm không biết tập trung vào những hoạt động có giá trị; phương pháp này đặc biệt hiệu quả để nhận biết nhóm nên tập trung vào những gì.

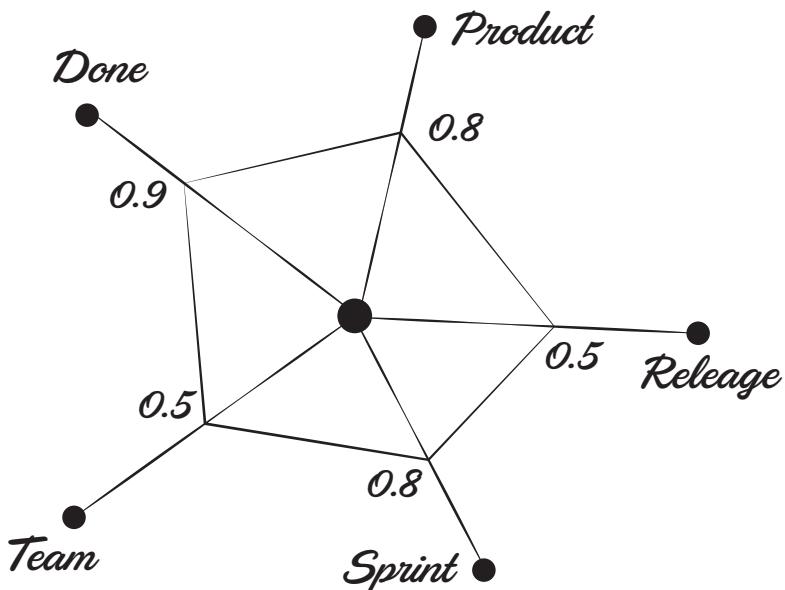
Đánh giá qua khảo sát

Một vấn đề thường được đặt ra là, nhóm phát triển muốn biết nhóm đã thực sự

tiến bộ như thế nào thông qua các chỉ số cụ thể. Phương pháp này dựa trên việc đánh giá thông qua khảo sát các thành viên trong nhóm phát triển, gồm những khía cạnh về chất lượng:

- Product Owner thực hiện
 - chuẩn bị User Story cho việc phát triển;
 - sắp xếp hiệu quả độ ưu tiên của User Story, release;
 - cộng tác với người quản lý sản phẩm và những nhân sự liên quan.
- *Iteration*: cách quản lý những hoạt động trong một Iteration
 - nhóm phát triển cộng tác hiệu quả trong việc lên kế hoạch thực hiện cho iteration;
 - nhóm phát triển hiểu rõ và cam kết thực hiện mục tiêu của iteration.
- Nhóm phát triển: tinh thần làm việc, cộng tác... trong nhóm
 - các thành viên trong nhóm tự tổ chức hiệu quả, tôn trọng và giúp đỡ những thành viên khác để nhóm đạt mục tiêu của Iteration;
 - các User Story được thực hiện qua nhiều lần lặp từ định nghĩa – thực hiện – kiểm thử.
- Kỹ thuật: mức độ áp dụng những best practice về công nghệ trong nhóm
 - thực hiện TDD, kiểm thử tự động;
 - có DoD;
 - thực hiện code refactoring.

Sau khi thu thập một loạt những thông tin, nhóm có thể thực hiện thống kê để có thông tin nhu:



Hình 5.18: Khảo sát chất lượng nhóm

Biểu đồ này cho thấy những khía cạnh nhóm đã đạt được với một điểm số cụ thể, qua đó cho thấy những điểm mạnh cần được duy trì và những điểm cần cải tiến.

Điểm mạnh của phương pháp này là độ cụ thể và số hoá được việc đo lường, thông qua đó cho thấy độ phát triển của nhóm. Lần lại lịch sử qua mỗi iteration, nhóm có thể thấy độ phát triển (hoặc đi xuống) thông qua những điểm số cụ thể. Đây là điều mà những phương pháp Retrospective khác khó thực hiện (bởi rất khó đo lường trên những phản hồi từ những câu hỏi mở). Tất nhiên, điểm yếu của phương pháp này là thời gian thực hiện; vì vậy, chúng ta không nên thực hành phương pháp này thường xuyên.

Những nguyên tắc cơ bản

Dù chúng ta lựa chọn phương pháp nào, vẫn có những nguyên tắc cơ bản sau cần được thực hiện để có Retrospective hiệu quả.

Thực hiện Retrospective đều đặn. Đối với Scrum, Retrospective được thực hiện sau mỗi Sprint; đối với những phương pháp khác, Retrospective không được định nghĩa, song nhóm nên lựa chọn thời gian cố định, đều đặn thực hiện. Nhóm sẽ không bao giờ tìm được quy trình cho chính mình nếu không được tổ chức dành cho một không gian, thời gian đều đặn để nhìn lại những điểm có thể cải tiến.

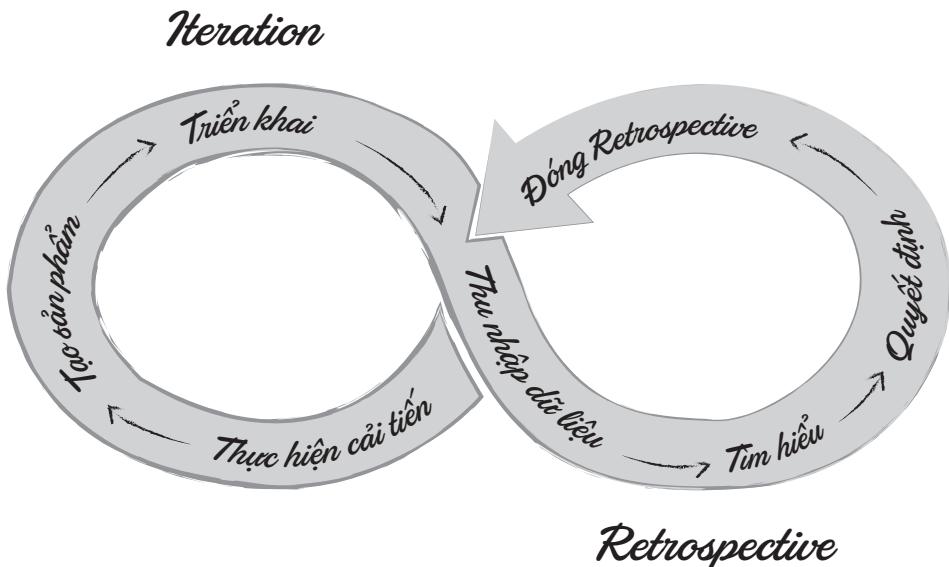
Không chỉ trích, giữ những cái đầu tinh táo. Với Scrum, Retrospective được thực hiện ngay sau Sprint Review nên sẽ chịu những ảnh hưởng nhất định. Với nhiều User Story thành công trong Sprint Review sẽ khiến nhóm hào hứng, ngược lại tâm lý của nhóm có thể thực sự tệ hại nếu Sprint Review thất bại. Việc này có thể dẫn đến những cái đầu không còn tinh táo và vô tình biến buổi Retrospective thành nơi để chỉ trích và đổ lỗi. Việc đổ lỗi, xét cho cùng chẳng đi đến đâu và nó nên bị loại bỏ.

Tập trung vào quy trình và cách làm, không tập trung vào cá nhân. Bởi tổ chức muốn nhóm phát triển và tìm ra quy trình phù hợp; và qua đó từng cá nhân phát triển.

Lựa chọn phương pháp đúng. Có hàng chục phương pháp thực hiện Retrospective, mỗi phương pháp đều có điểm mạnh và điểm yếu riêng; phương pháp Retrospective phù hợp với tình hình của nhóm sẽ mang lại hiệu quả cao.

Đặt câu hỏi tích cực. Câu hỏi *chúng ta đã làm gì kém?* thường mang ý nghĩa tiêu cực và tạo ra suy nghĩ phòng thủ với người trả lời (nghe giống như một yêu cầu giải trình); câu hỏi *chúng ta đã có thể làm gì tốt hơn?* có màu sắc tích cực hơn.

Phải có hành động tiếp theo. Dù thực hiện Retrospective theo phương pháp nào, nhóm phải đưa ra được danh sách những hành động tiếp theo, đây chính là những cải tiến sẽ được thực hiện.



Hình 5.19: Tác dụng của Retrospective

HIẾU ĐÚNG

Để đảm bảo chất lượng, source code được review bởi người có kinh nghiệm hơn.

Đây thường là cách những nhóm phát triển phần mềm truyền thống hay thực hiện, nhưng trong nhóm thực hành Agile thì không. Thật khó phủ nhận hiệu quả việc một lập trình viên có kinh nghiệm review source code của những lập trình viên ít kinh nghiệm hơn; 80% trường hợp tốt hơn hẳn. Song trong nhóm thực hành Agile, review source code không hẳn chỉ nhằm mục đích đảm bảo chất lượng của những source code được check-in vào repository; việc này còn mang tác dụng chia sẻ kiến thức, cung cấp những góc nhìn, giải pháp khác nhau, qua đó tạo ra collective ownership source code. Và về lâu dài, việc này thúc đẩy chất lượng source code được nâng cao.

Lập trình cặp luôn cho hiệu quả cao hơn.

Có rất nhiều nhóm thực hành Agile luôn thực hiện lập trình cặp, luôn luôn, bởi họ tin rằng lập trình cặp sẽ cho chất lượng cao hơn. Thật khó phủ nhận điều này; lập trình cặp chắc chắn luôn cho chất lượng cao hơn, song chúng ta luôn cần cân nhắc về hiệu quả kinh tế. Với những công việc không thể xảy ra sai sót (như tạo model,...) thì lập trình cặp không hiệu quả về mặt chi phí.

TDD bắt buộc phải viết test trước khi viết code.

Sẽ thật tuyệt vời nếu nhóm thực hành Agile thực hiện TDD, song giai đoạn đầu sẽ thực sự khó khăn. Theo tôi, không có vấn đề gì nếu nhóm thực hiện việc viết test sau khi đã viết code (trong giai đoạn đầu thực hành TDD). Vì ít nhất, nhóm cũng có thể đảm bảo việc regression test được thực hiện tốt. Nếu giới hạn lập trình viên thực hành TDD vào TFD, nhóm thực hành Agile chỉ có thể áp dụng trên những chức năng mới, không thể áp dụng vào legacy code, mà đó cũng là một phần rất quan trọng.

Retrospective hạn chế nói về con người.

Mục đích của Retrospective là tìm ra những điểm chưa hợp lý trong cách làm việc giữa những thành viên trong nhóm thực hành Agile. Scrum guide hướng trọng tâm tới việc nói về quy trình, cách làm; song không có nghĩa là vấn đề con người nên bị bỏ qua hoặc lảng tránh. Nếu nhóm gặp vấn đề về con người, hãy thẳng thắn nói về điều đó và giải quyết. Nhưng hãy nhớ rằng nói về con người không có nghĩa là tấn công cá nhân, nhóm thực hành Agile cần duy trì sự tinh táo để nói về con người một cách xây dựng và tích cực nhất.

TỔNG KẾT

Bảng là công cụ tối quan trọng với những nhóm thực hành Agile. Dù là bảng vật lý hay bảng điện tử, mức độ *chuyển động* và *sự tập trung* của nhóm vào bảng cho thấy một phần độ trưởng thành của nhóm thực hành Agile.

Continuous Integration về mặt source code được hiểu là việc *tích hợp liên tục* với nhau tạo để tránh việc *tích hợp trễ* gây ra lỗi tích hợp giữa các thành phần, module. *Continuous Integration* về mặt quy trình được hiểu là việc *đồng bộ công việc* giữa những thành viên trong nhóm nhằm đảm bảo mọi thành viên đều nắm được tình trạng công việc hiện tại.

Continuous Delivery được hiểu là khả năng *nhanh chóng* và *đều đặn triển khai* những thay đổi từ bug fix tới những chức năng mới tới môi trường được sử dụng bởi người dùng một cách an toàn và hiệu quả.

Trong dự án Agile, database không phải là một thành phần bất biến, việc thay đổi có thể được liên tục diễn ra. Để đảm bảo việc thay đổi được diễn ra an toàn và hiệu quả, database cũng nên được coi là một phần tài nguyên như source code, và cũng cần giải pháp versioning.

Bất kỳ nhóm thực hành Agile nào cũng cần một *branching strategy* cụ thể và phù hợp. Bởi thông thường, do cách thiết kế phần mềm kiểu Agile, rất nhiều xung đột có thể xảy ra, branching strategy và continuous integration chính là công cụ giải quyết những xung đột này. Branching strategy hiệu quả còn giúp việc release trở nên an toàn và đảm bảo chất lượng.

Code refactoring là hoạt động không thể thiếu, giúp nhóm thực hành Agile liên tục cải tiến chất lượng source code, hay chính là chất lượng phần mềm. Hãy cố gắng thực hiện code refactoring bất cứ khi nào có thể nhưng chỉ khi các test đã sẵn sàng.

Chất lượng source code có thể được đảm bảo theo nhiều cách. Ở mức cơ bản, là những quy chuẩn về ngôn ngữ, phong cách lập trình; ở mức cao hơn, là giải pháp cho một vấn đề cụ thể. Peer review là một cách làm tốt, ngoài việc nâng cao chất lượng source code, còn giúp những thành viên trong nhóm học hỏi lẫn nhau.

Test Driven Development là một kỹ thuật tuyệt vời của Agile nhằm *thu về một mối* vấn đề không đồng nhất giữa tài liệu, source code và testing. Test được coi là tài liệu, là thiết kế đảm bảo sự đúng đắn của source code từ mức function tới một chức năng cụ thể về mặt người dùng.

Lập trình cặp luôn là vấn đề hoài nghi và gây nhiều tranh cãi trong Agile. Nhóm thực hành Agile nên thực hành lập trình cặp bất cứ khi nào có thể (nếu đủ nguồn lực); nếu không, nên dành không gian cho việc lập trình cặp với những vấn đề phức tạp. Lập trình cặp ít khi cho hiệu quả nhìn thấy trực tiếp nhưng chất lượng phần mềm được cải thiện đáng kể.

Retrospective là điều tuyệt vời mà Agile mang lại về mặt quy trình, giúp nhóm thực hành Agile nhìn lại, biết những vấn đề đã thực hiện tốt và những vấn đề cần giải quyết nhằm tìm ra những hành động đúng đắn, cũng như tìm ra quy trình của riêng nhóm.

TỔ CHỨC LINH HOẠT

Những phương pháp Agile cơ bản như Scrum hay XP tập trung giải quyết vấn đề trong nhóm phát triển phần mềm. Sau một thập kỷ chứng minh được tính hiệu quả với những nhóm nhỏ, Agile đã tiến thêm một bước với làn sóng mở rộng tới quy mô của những dự án lớn hay trong toàn bộ tổ chức. Ở bất kỳ quy mô nào, Agile vẫn giữ nguyên chân giá trị về việc coi trọng con người, sự cộng tác nhằm nâng cao tính linh hoạt khiến giá trị kinh doanh được sinh ra nhanh nhất có thể. Đó là những gì chúng ta sẽ nghiên cứu trong 3 chương tiếp theo.



HTC

NHÓM
TỰ TỐ CHÚC
LIÊN CHÚC NĂNG

Nhóm tự tổ chức *liên chức năng* đã được tôi đề cập trong Chương 2 về Scrum nhưng chỉ dừng ở mức khái niệm. Có thể đây là lần đầu bạn nghe thấy thuật ngữ *về nhóm tự tổ chức liên chức năng*, song những người khai sinh ra Agile hoàn toàn không sáng tạo ra thuật ngữ hay mô hình này; họ chỉ áp dụng một trong nhiều phương pháp quản lý hiện đại vào việc phát triển phần mềm. Do đó, không có gì ngạc nhiên nếu *nhóm tự tổ chức liên chức năng* hoạt động hiệu quả trong nhiều lĩnh vực khác, ở quy mô dự án hay tổ chức.

Sự thành bại của việc áp dụng Agile trong dự án phát triển phần mềm hay tổ chức phụ thuộc rất lớn vào cách xây dựng cũng như quản lý và thực hành *nhóm tự tổ chức liên chức năng* một cách hiệu quả. Đó là lý do tôi muốn đi sâu hơn vào nội dung này trong Chương 6.

Về cơ bản, *nhóm tự tổ chức liên chức năng* khá dễ hiểu về mặt khái niệm, là *nhóm tự tổ chức* và *liên chức năng*, nhưng không thực sự đơn giản để áp dụng hiệu quả. Vì vậy, chúng ta cần phân tích nhiều hơn vào từng vấn đề cụ thể.

NHÓM LIÊN CHỨC NĂNG

Nhóm liên chức năng (cross functional) là nhóm tập hợp những cá nhân có đầy đủ kiến thức, kỹ năng cần thiết để cam kết và thực hiện một mục tiêu đề ra. Nhóm liên chức năng hoạt động như một đơn vị, những thành viên thường xuyên trao đổi, cộng tác và hỗ trợ lẫn nhau trong những hoạt động, công việc cần thiết nhằm đạt được mục tiêu chung.

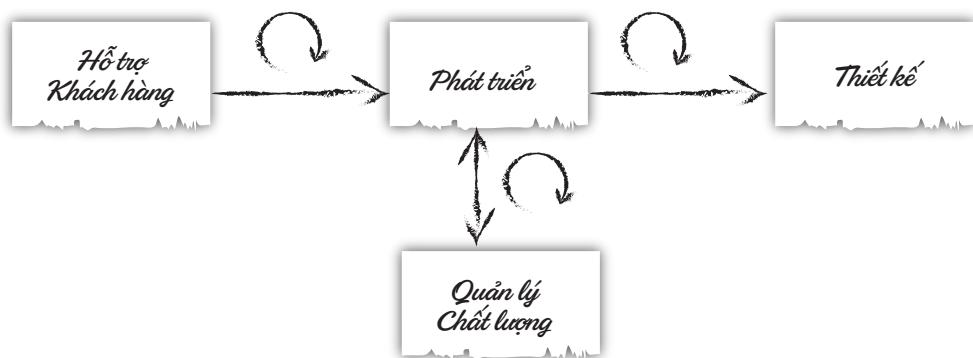
Ở quy mô tổ chức hay doanh nghiệp, *nhóm liên chức năng* thường là nhóm tập hợp những cá nhân từ những bộ phận chuyên trách. Hầu hết những doanh nghiệp ngày nay được tổ chức dưới dạng phân cấp theo chức năng chuyên môn; *nhóm liên chức năng* có thể được minh họa như sau:

Thiết kế	Phát triển	Quản lý Chất lượng	Hỗ trợ khách hàng
Chi Mai	Hùng Dũng	Yến Toàn	Tâm
Đạt	An Bình	Cường	Giang
	Nghĩa		Khanh

Hình 6.1: *Nhóm liên chức năng* được hình thành bởi những thành viên từ những bộ phận khác nhau

Trong đó một nhóm được hình thành với nhiều cá nhân từ những bộ phận có chức năng chuyên trách như An và Bình từ bộ phận *Phát triển*, Cường từ bộ phận *Quản lý chất lượng*, Đạt từ bộ phận *Thiết kế* và Giang từ bộ phận *Hỗ trợ khách hàng*. Họ có thể làm việc cùng nhau trong một hoặc nhiều dự án nhằm giải quyết những vấn đề có thể hỗ trợ khách hàng xuyên suốt từ việc tiếp nhận thông tin, phát triển và đảm bảo chất lượng của chức năng được chuyển giao tới khách hàng.

Tại sao cần nhóm liên chức năng? Nhóm liên chức năng được hình thành với giả định rằng một nhóm với tập hợp đủ kiến thức, kỹ năng cần thiết sẽ nhanh chóng đạt được mục tiêu thay vì từng cá nhân làm việc đơn lẻ theo cấu trúc bộ phận chuyên trách của tổ chức. Rất ít cá nhân có đủ kiến thức, kỹ năng cần thiết để thực hiện mọi công việc cấu thành nên giá trị kinh doanh. Ví dụ, để mang lại giá trị qua chức năng *quản lý danh sách sản phẩm* đòi hỏi nhiều kiến thức từ nghiệp vụ, thiết kế, lập trình, kiểm thử tới triển khai; cũng tương ứng với một tập kỹ năng lớn từ giao tiếp, phát triển phần mềm... tới quản trị hệ thống. Để biết hết những kiến thức, kỹ năng này là điều không thể với mỗi người. Từ đó nảy sinh ra cấu trúc của tổ chức dạng phân cấp theo bộ phận chuyên trách. Nhưng cấu trúc này không hỗ trợ việc *giao tiếp giữa những cá nhân ở những bộ phận khác nhau*, khiến họ chậm chạp trong việc phản ứng với những thay đổi. Điều gì xảy ra khi khách hàng cần sự hỗ trợ? Bộ phận *Hỗ trợ khách hàng* tiếp nhận yêu cầu, chuyển yêu cầu tới bộ phận *Phát triển*, bộ phận *Phát triển* chờ đợi một thiết kế từ bộ phận *Thiết kế*, trước khi chuyển giao thành quả tới bộ phận *Quản lý chất lượng*. Hàng loạt những vòng lặp trao đổi xảy ra như sau:



Hình 6.2: Những vòng lặp trao đổi trong nhóm truyền thống

Vấn đề gặp phải với mô hình này là việc giao tiếp xảy ra giữa *bộ phận – bộ phận* thay vì *cá nhân – cá nhân* dù công việc thực sự được thực hiện do từng cá nhân

đảm nhiệm. Thời gian cộng tác với những khoảng thời gian *chết* rất lớn. Nhưng vấn đề lớn nhất gặp phải theo mô hình này, là những cá nhân tham gia vào công việc không chia sẻ với nhau một *tầm nhìn chung* về *giá trị được tạo ra*. Bởi thiếu *tầm nhìn chung*, bộ phận *Thiết kế* có thể tạo ra một bản thiết kế giao diện rất đẹp nhưng tiêu tốn nhiều thời gian của bộ phận *Phát triển*, dẫn tới việc không dễ dàng để kiểm tra bởi bộ phận *Quản lý chất lượng*. Bởi vậy tập hợp những cá nhân này thành một nhóm, cùng chia sẻ *tầm nhìn chung*, cộng tác với nhau dưới dạng *network* giúp họ nhanh chóng đạt được mục tiêu đề ra.

Khi nào cần nhóm liên chức năng? Điểm mạnh của mô hình quản lý *phân cấp theo chức năng* cũng chính là điểm yếu của mô hình quản lý *nhóm liên chức năng*. Nhóm *liên chức năng* không thể quá lớn. Khi gồm quá nhiều thành viên với những kiến thức, kỹ năng, văn hoá khác nhau, nhóm không thể cùng chia sẻ với nhau một *tầm nhìn chung* cũng như *giá trị, văn hoá cụ thể*. Do đó, *nhóm liên chức năng* chỉ phù hợp với những bài toán cụ thể.

Với những bài toán đơn giản, nhóm *liên chức năng* không phát huy hiệu quả. Đó là những vấn đề có thể được từng cá nhân thực hiện riêng lẻ, ít sai sót và việc ghép nối những thành quả của từng công đoạn không xảy ra sai sót. Nói chung, đây là những bài toán cơ bản của tổ chức, đã có quy trình cụ thể và không cần nhiều thông tin trong mỗi phân đoạn. Ví dụ, trong một nhà hàng, bộ phận *Lễ tân* nhận yêu cầu đặt món từ khách hàng, gửi thông tin tới bộ phận *Bếp*; món ăn sau khi được chế biến xong được bộ phận *Chạy bàn* chuyển tới đúng địa chỉ. Ở đó họ không bao giờ cần những *nhóm liên chức năng*.

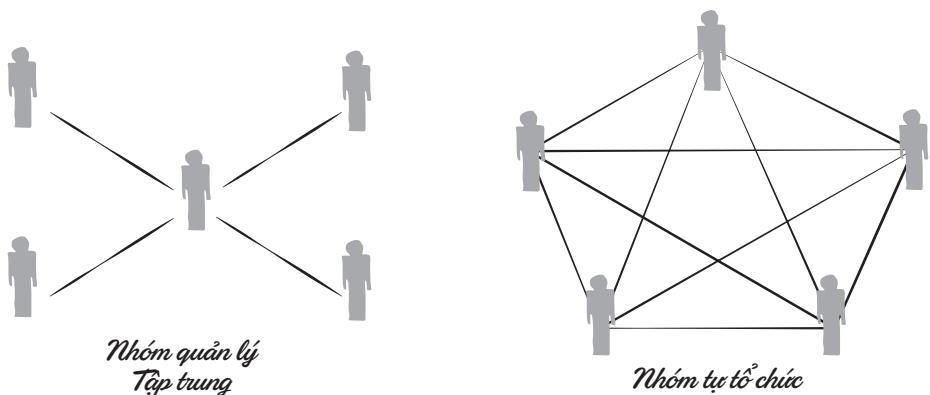
Với những bài toán cực kỳ phức tạp, nhóm *liên chức năng* không phát huy hiệu quả. Đó là những vấn đề cần chuyên môn ở mức sâu và rất lâu mới sản sinh ra giá trị chuyển giao. Ví dụ, trong ngành dược phẩm, việc nghiên cứu một hợp chất và thực hiện thí nghiệm để đưa ra một loại thuốc cụ thể sẽ cần tới hàng năm với xác suất thành công thấp; thì không có lý do gì để hình thành nhóm *liên chức năng* giữa những chuyên gia về thuốc và chuyên gia marketing.

Nhóm *liên chức năng* hoạt động tốt trong những vấn đề nằm ở giữa của hai tình huống trên. Đó là những bài toán phức tạp về việc không định hình rõ yêu cầu, giải pháp và cần đề cao sự cộng tác giữa những bộ phận có kiến thức, kỹ năng khác biệt. Ví dụ như phát triển phần mềm, với những dự án có yêu cầu, giải pháp kỹ thuật không rõ ràng.

NHÓM TỰ TỔ CHỨC

Nhóm tự tổ chức (self-organized team) là nhóm không tập trung quyền hạn vào một thành viên cụ thể; tự định nghĩa và thực hành cách thức thực hiện, cộng tác để đạt được mục tiêu đề ra; có thể tự đưa ra quyết định mà không phụ thuộc vào hệ thống bên ngoài nhóm.

Cách tổ chức truyền thống với một nhóm gồm những thành viên và người quản lý – là trung tâm, chịu trách nhiệm về quyết định, điều phối công việc nhằm hướng tới mục tiêu đề ra. Nhóm tự tổ chức chia sẻ trách nhiệm, cộng tác và quản lý công việc giữa những thành viên.



Hình 6.3: Nhóm quản lý tập trung và nhóm tự tổ chức

Tại sao cần nhóm tự tổ chức? Nhóm tự tổ chức được hình thành nhằm giải quyết vấn đề của việc quản lý tập trung được điều hướng bởi *comand and control* với người quản lý là trung tâm của mọi quyết định, đưa ra mệnh lệnh và quản lý việc thực thi. Mô hình quản lý tập trung gấp phải 3 vấn đề cơ bản. Thứ nhất, người quản lý nhanh chóng trở thành *nút thắt* trong mọi quyết định; đặc biệt trong môi trường biến động và cần liên tục có những quyết định cụ thể; khi đó, hiệu suất của nhóm phụ thuộc lớn vào hiệu suất của người quản lý. Thứ hai, trách nhiệm của những thành viên trong nhóm thấp; do quyết định được đưa ra bởi người quản lý, anh ta chính là người chịu trách nhiệm. Thứ ba, động lực của mỗi thành viên không cao; bởi họ không phải là người chịu trách nhiệm chính cho quyết định, công việc được thực thi. Trong khi đó, càng ngày, con người càng muốn bước lên những nấc thang cao hơn; và muốn được cống hiến với một mục tiêu cụ thể, thay vì làm việc như một chiếc máy.

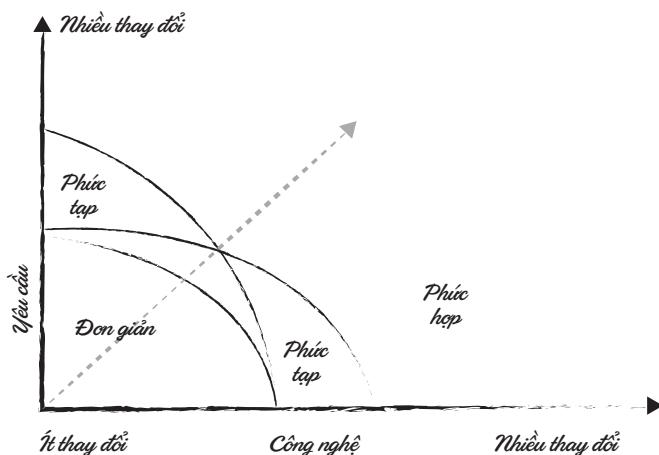
Khi nào cần nhóm tự tổ chức? Khi phương pháp quản lý *comand and control* không

phát huy hiệu quả. Đó thường là những môi trường, dự án biến động, luôn cần đưa ra nhiều quyết định để nhanh chóng phản hồi với những biến động, thay đổi. Đó cũng thường là những môi trường, dự án mà những thành viên trong nhóm muốn làm việc hướng tới *giá trị, ý nghĩa* của công việc, thay vì chỉ đơn giản là hoàn thành những công việc được xác định trước.

NHÓM TỰ TỔ CHỨC LIÊN CHỨC NĂNG

Như vậy, *nhóm tự tổ chức liên chức năng* (*self-organized cross-functional team*) là nhóm được hình thành bởi những cá nhân có đầy đủ kiến thức, kỹ năng cần thiết để hướng tới mục tiêu được đặt ra; thông qua việc thường xuyên trao đổi, cộng tác, hỗ trợ lẫn nhau, nhóm tự đưa ra quy trình, cách thức cũng như quyết định để đạt được mục tiêu chung và không chịu tác động của bất kỳ hệ thống nào ngoài nhóm.

Hầu hết những phương pháp nằm trong *chiếc ô Agile* đều hướng tới việc tổ chức nhóm *tự tổ chức liên chức năng* như một trong những yếu tố quan trọng để đi đến thành công.



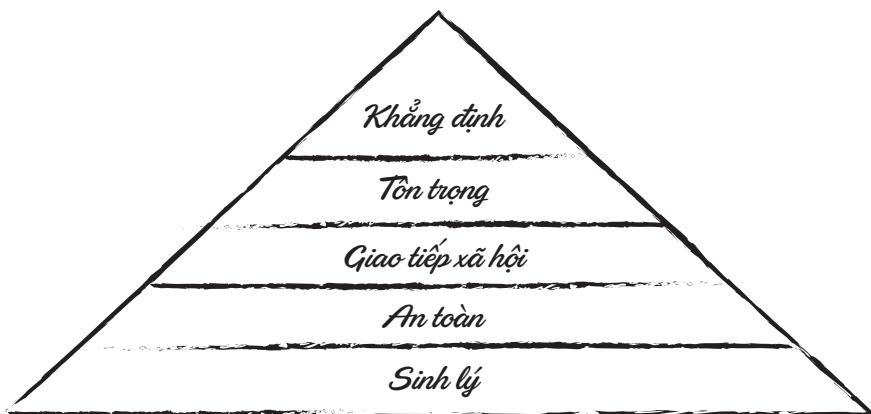
Hình 6.4: Những dự án phù hợp với nhóm tự tổ chức liên chức năng

Hình trên cho thấy những dự án phù hợp trong việc áp dụng Agile, đó là nơi những dự án phức hợp và phức tạp về mặt yêu cầu hoặc kỹ thuật. Thông thường, những dự án phát triển phần mềm đều có chung đặc tính này. Điều này lý giải tại sao Agile giờ đây được áp dụng trong gần như mọi dự án phát triển phần mềm.

Phát triển phần mềm cần nhiều kỹ năng và việc thành thục mỗi kỹ năng đòi hỏi rất nhiều thời gian. Nhưng những kỹ năng này không dễ dàng *ghép nối* với nhau;

việc thất bại của phương pháp phát triển phần mềm truyền thống là minh chứng rõ ràng nhất. Do đó, nhóm *liên chức năng* là cần thiết.

Khi yêu cầu hoặc công nghệ không rõ ràng, việc thường xuyên thay đổi và liên tục đưa ra quyết định là điều cần thiết. Những kỹ sư phần mềm cũng ngày càng bước lên những bậc cao hơn của tháp Maslow; nơi họ muốn khẳng định bản thân thông qua việc ra quyết định, chịu trách nhiệm với quyết định và công việc nhằm hướng tới việc tạo ra giá trị hơn là chỉ hoàn thành công việc một cách đơn thuần. Do đó, nhóm *tự tổ chức* là phù hợp.



Hình 6.5: Tháp Maslow

TÍNH CHẤT

Không phải mọi nhóm *tự tổ chức* *liên chức năng* đều hoạt động hiệu quả và tạo ra giá trị như mong đợi. Tổ chức không thể chỉ dựa trên sự phù hợp của mô hình này, tập hợp những cá nhân từ nhiều phòng chuyên môn khác nhau, thông báo rằng họ *tự tổ chức* cũng như mục tiêu của nhóm và chờ đợi sự thành công. Tổ chức cần xây dựng nhóm dựa trên những yếu tố cần có để đảm bảo một nhóm là *nhóm tự tổ chức* *liên chức năng*.

Dũng cảm. Đây là yếu tố quan trọng nhất của mỗi thành viên cũng như cả nhóm. Ra quyết định chưa bao giờ là việc dễ dàng; không phải bởi việc đưa ra quyết định; người đưa ra quyết định phải *chịu trách nhiệm* với kết quả được tạo ra bởi quyết định. Trách nhiệm luôn là nỗi sợ của việc ra quyết định; mỗi thành viên cũng như cả nhóm cần sự dũng cảm để vượt qua. Bởi *nhóm tự tổ chức* *liên chức năng* không chịu bất kỳ sự chi phối nào từ bên ngoài, nhóm cần dũng cảm đối mặt với thành công và thất bại của mình.

Cam kết. Nhóm *tự tổ chức* *liên chức năng* không hoạt động theo cơ chế *command and con-*

trol, và không chịu sự kiểm soát từ bên ngoài; chỉ có cam kết là yếu tố đảm bảo duy nhất. Một khi nhóm đã đồng ý với mục tiêu chung được đề ra, nhóm phải chứng minh được sự nỗ lực để cam kết với mục tiêu. Nhiều quyền hạn đồng nghĩa với nhiều trách nhiệm.

Cộng tác. Nhóm tự tổ chức liên chức năng được thành lập với mục tiêu loại bỏ những ràng buộc phức tạp để gắn kết và tăng cường việc cộng tác trực tiếp; qua đó nâng cao khả năng phản hồi với những thay đổi và nhanh chóng tạo ra giá trị. Cộng tác dựa trên giao tiếp bằng sự tôn trọng là kỹ năng không thể thiếu của mỗi thành viên.

SỰ PHÁT TRIỂN CỦA NHÓM

Dũng cảm, cam kết, cộng tác luôn là điều kiện cần để thành lập một nhóm tự tổ chức liên chức năng, song ngay cả khi nhóm được thành lập với những cá nhân ưu tú nhất, thoả mãn những điều kiện trên, nhóm tự tổ chức liên chức năng cũng không ngay lập tức đạt tới trạng thái hiệu quả nhất.

Bruce Tuckman đưa ra *mô hình* các trạng thái phát triển của nhóm (*stages of group development model*) gồm 4 giai đoạn là *forming – storming – norming – performing* được biết đến rộng rãi, và hầu như mọi nhóm tự tổ chức liên chức năng đều trải qua.



Mô hình các trạng thái phát triển của nhóm của Tuckman tập trung nghiên cứu cho nhóm liên chức năng và hoàn toàn không đề cập tới mô hình tự tổ chức; do đó cách giải quyết để nhóm bước tới những trạng thái tiếp theo của Tuckman đôi khi cần nhiều sự tác động từ bên ngoài, hơn là việc xử lý nội tại trong nhóm đúng với tinh thần tự tổ chức. Song mô hình Tuckman thường xuyên được sử dụng để diễn giải với những nhóm mới thực hành Agile bởi sự đơn giản và dễ hiểu.

Bạn có thể tìm hiểu thêm một số mô hình khác như của mô hình Kurt Lewin, Tubbs...

Forming – Hình thành

Mọi nhóm ngay sau khi thành lập đều ở trạng thái hình thành với một tập hợp những cá nhân đơn lẻ cùng sự khác biệt về kỹ năng, văn hoá. Những cá nhân sẽ chỉ thấy mình như những thành phần đơn lẻ, chưa có cảm giác về *nhóm*. Trong giai đoạn này, năng suất làm việc của nhóm thấp.

Trong giai đoạn này, mỗi thành viên trong nhóm đều đứng xa nhau để thăm dò, và thường có những câu hỏi chung:

- Sứ mệnh, mục tiêu chung của nhóm?
- Những phương pháp, quy chuẩn nào sẽ được áp dụng?
- Những hành vi nào được chấp nhận?
- Nhóm có hội tụ những thành viên đúng?
- Việc đánh giá cho nhóm và từng cá nhân?

Storming – Giông bão

Sau khi trả lời được những câu hỏi chung, những thành viên trong nhóm có cảm giác gần nhau hơn và bắt đầu thực hiện việc cộng tác để hướng tới mục tiêu chung, nhưng sự khác biệt về nền tảng, văn hoá bắt đầu làm nảy sinh xung đột. Năng suất làm việc của nhóm có xu hướng giảm hoặc không tăng nhanh.

Những vấn đề thường gặp phải trong giai đoạn này:

- Cách xử lý xung đột trong phương pháp thực hiện?
- Cách tạo sự đồng thuận và ra quyết định?

Norming – Hình thành quy tắc

Thông qua việc giải quyết hàng loạt những xung đột, nhóm bắt đầu tìm thấy tiếng nói chung trong những hoàn cảnh cụ thể và hình thành những quy tắc để làm việc cùng nhau cho mỗi cá nhân cũng như toàn nhóm. Nhóm bắt đầu có năng suất cao hơn và hiệu quả trong việc tạo ra giá trị. Cũng thông qua cách xử lý những xung đột trong quá khứ, nhóm bắt đầu nhận thức được tầm quan trọng của việc cải tiến cách thức làm việc nhằm nâng cao hiệu quả.

Khi đó, nhóm bắt đầu quan tâm tới những vấn đề lớn hơn:

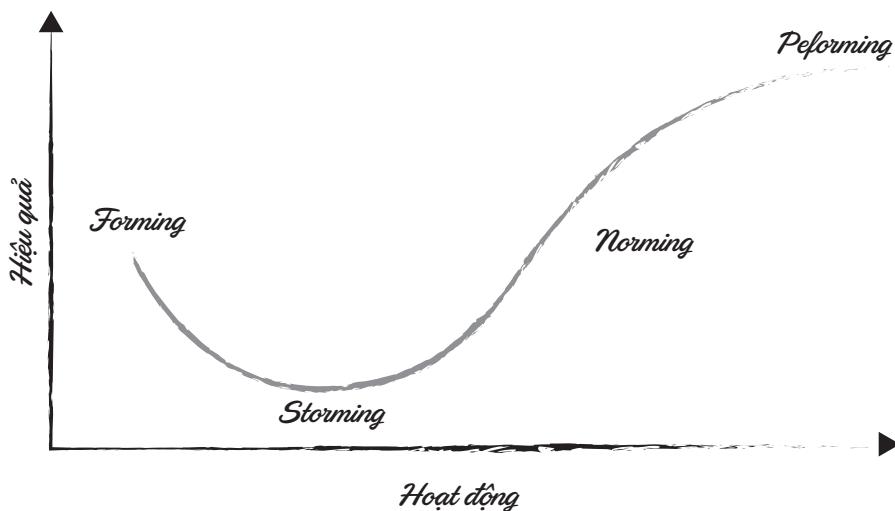
- Nhóm sẽ cùng nhau đạt được mục tiêu?
- Cách cải tiến hiệu suất và hiệu quả?

Performing – Hiệu suất

Nhóm đạt hiệu suất làm việc cao dựa trên sự ăn ý giữa những thành viên, việc giải quyết xung đột trở nên đơn giản và theo xu hướng tích cực. Đây là trạng thái nhóm cho hiệu suất và hiệu quả làm việc cao nhất.

Lúc này, nhóm sẽ quan tâm tới những vấn đề lớn hơn:

- Vai trò của nhóm với những thành tố bên ngoài? Tổ chức và đội ngũ quản lý có quan tâm và phản hồi tới nhóm và những khuyến nghị từ nhóm?
- Cách nhóm tiếp tục thực hiện những công việc cùng nhau ở hiệu suất và hiệu quả cao.



Hình 6.6: Các trạng thái phát triển của nhóm

Sẽ rất tuyệt vời nếu nhóm tự tổ chức liên chức năng nhanh chóng đạt tới trạng thái *Performing*, nhưng để đạt được trạng thái này, nhóm buộc phải trải qua những trạng thái khác, hoàn toàn không có con đường đi tắt tới sự hiệu quả khi làm việc nhóm. Mỗi khi nhóm bị thay đổi bởi mục tiêu, (thêm hoặc bớt) thành viên,... hãy

lưu ý rằng nhóm sẽ quay trở lại trạng thái đầu tiên *Forming* và buộc phải đi lại con đường cũ. Bởi khi những yếu tố này thay đổi, những xung đột sẽ bắt đầu được sản sinh và hạn chế hiệu suất làm việc chung của nhóm; ít hay nhiều phụ thuộc vào quy mô của sự thay đổi. Nếu nhóm thay đổi 3/5 thành viên, hoặc thực hiện dự án phức tạp, con đường tới *Performing* sẽ dài hơn với việc thay đổi 1/5 thành viên hoặc thực hiện một dự án tương tự.

VAI TRÒ CỦA NGƯỜI QUẢN LÝ VÀ TỔ CHỨC

Sẽ không có con đường khác để nhóm đạt được hiệu suất cao, nhưng nếu tổ chức và những người quản lý biết cách trợ giúp nhóm, con đường dẫn tới trạng thái *Performing* sẽ ngắn hơn. Chúng ta không thể tập hợp những thành viên từ những bộ phận khác nhau, hy vọng nhóm sẽ *tự tổ chức* và đem lại thành công; ngược lại, vai trò của tổ chức và những người quản lý là rất quan trọng, ít nhất ở ba điểm: xác định trạng thái hiện tại của nhóm, hành động để giải quyết những vấn đề để tiến tới trạng thái tiếp theo, xuyên suốt đó là quá trình huấn luyện, trợ giúp.

Những vấn đề được nêu trong từng trạng thái nhóm của Tuckman cũng đưa ra chỉ dẫn tới những hành động phù hợp của tổ chức nhằm giúp nhóm đạt tới và duy trì trạng thái *Performing*.

Forming

Trong giai đoạn này, những hành động của tổ chức rất quan trọng. Mọi thành viên trong nhóm có chung cảm giác về sự đơn lẻ, và luôn hoài nghi rằng nhóm là tập hợp của những thành viên phù hợp cũng như phân vân về những quy tắc, hành vi được chấp nhận trong nhóm bởi sự không rõ ràng về mục tiêu.

Tổ chức hoặc những nhà quản lý phải *ngay lập tức* có những hoạt động truyền thông hiệu quả. *Sứ mệnh và mục tiêu chung* của nhóm là điều đầu tiên cần làm rõ và thống nhất nhằm đảm bảo chỉ có một cách hiểu duy nhất về những gì nhóm cần hướng tới. Tổ chức và những nhà quản lý cũng cần chỉ rõ con đường mà nhóm sẽ cùng nhau đi, qua những trạng thái nào, để nhóm hiểu rằng việc xung đột trong tương lai là chuyện hoàn toàn dễ hiểu. Về cơ bản, để hình thành *nhóm tự tổ chức* cần *liên kết năng lực*, tổ chức và những nhà quản lý sẽ cố gắng không can thiệp vào những quyết định của nhóm về cách thức hoạt động, cộng tác; song để nhóm nhanh chóng vượt qua giai đoạn này, tổ chức cố gắng hướng dẫn, huấn luyện hoặc chuyển giao cho nhóm một phương thức hoạt động chung như một định hướng ban đầu, có thể rất sơ khởi nhưng là nền tảng để nhóm làm việc cùng nhau. Ví dụ như Scrum hay Kanban; khi nhóm được hình thành, tổ chức cần đào tạo cho nhóm về cách vận hành của Scrum hay Kanban gồm nền tảng giá

trị, những vai trò, những sự kiện, cách thức vận hành... Việc này giúp giải quyết mối hoài nghi về phương pháp, quy trình nào được áp dụng cũng như tập hành vi được chấp thuận trong nhóm.

Trước khi yêu cầu nhóm cam kết thực hiện mục tiêu chung, tổ chức cần thực hiện *cam kết trao quyền* cũng như *phạm vi quyền hạn của nhóm*. Điều này giúp nhóm cảm nhận được không gian hoạt động cũng như nâng cao sự tự tin và trách nhiệm chung. Về cơ bản, *nhóm tự tổ chức liên chức năng* chỉ có thể cam kết tới mục tiêu nếu tổ chức minh bạch và cam kết việc trao quyền cũng như những nguồn lực có thể hỗ trợ nhóm. Bằng sự tự trọng cũng như hiểu biết của mình, sẽ không có nhóm nào cam kết việc *hoàn thành sản phẩm và tìm kiếm 100 khách hàng trong 3 tháng* nếu tổ chức không minh bạch cam kết về ngân sách, sự hỗ trợ của những phòng ban liên quan.

Những hoạt động xã hội, xây dựng nhóm trong giai đoạn này cũng cần thiết. Trong giai đoạn đầu, mọi thành viên trong nhóm đều e dè vì hoàn toàn không có hiểu biết về những thành viên khác, cũng như vai trò của mình trong nhóm. Những hoạt động xây dựng nhóm giúp những thành viên hiểu về nhau, những nét tính cách, thói quen cũng như những quan niệm về những hành vi được chấp nhận. *Nhóm liên chức năng* vừa là thử thách vừa là lợi thế, chính từ sự khác nhau trong *chức năng*. Những cá nhân với chuyên môn khác nhau thường có những đặc trưng văn hoá, hành xử khác nhau, khiến "khoảng cách" của họ xa nhau hơn; nhưng chính sự khác biệt đó lại là nét tươi mới mà mỗi cá nhân có thể tìm thấy ở những thành viên khác. Lập trình viên, những người có xu hướng tư duy logic, lại có thể tìm thấy những kiến thức thú vị về tâm lý con người, đôi khi rất phi logic từ thành viên thực hiện marketing. Việc tổ chức tận dụng lợi thế này qua những hoạt động xây dựng nhóm rất quan trọng, giúp mọi thành viên trong nhóm xích lại gần nhau.

Storming

Xung đột chính là nền tảng của sự phát triển, *nhóm tự tổ chức liên chức năng* cũng không đứng ngoài quy luật này.

Trong giai đoạn Storming, điều tối quan trọng là tổ chức có cách hành xử đúng với những xung đột xảy ra. Thông thường, xung đột xảy ra do những thành viên trong nhóm có sự khác biệt trong cách tiếp cận giải quyết vấn đề, một phần đến từ kinh nghiệm và góc nhìn về chuyên môn khác nhau. Mỗi khi xung đột xảy ra, ngoài việc cố gắng bảo vệ quan điểm, góc nhìn, giải pháp của mình, mỗi cá nhân trong nhóm còn "thăm dò" phản ứng của những thành viên khác trong nhóm cũng như của tổ chức và những người lãnh đạo. Bởi theo kinh nghiệm trong quá

khứ, họ biết rằng việc để xảy ra xung đột có thể là *con dao hai lưỡi*, tổ chức có thể đánh giá họ tốt hoặc trừng phạt họ vì đã tạo ra xung đột. Lúc này, vai trò của tổ chức và những người quản lý rất quan trọng cả trong truyền thông và hành động.

Về truyền thông, tổ chức và những người quản lý cần khẳng định *những xung đột tích cực được khuyến khích xảy ra*; cùng với đó là việc định nghĩa rõ ràng những gì được coi là *xung đột tích cực*. Việc này giúp nhóm nhận biết rõ ràng về những gì mình làm, những xung đột có thể tạo ra, và điều quan trọng là, cảm thấy an toàn – chính là động lực để không ngăn cản việc tạo ra xung đột tích cực. Tất nhiên, việc tranh luận sử dụng chân tay hay mạt sát cá nhân luôn tồi tệ – là *những xung đột tiêu cực*; mọi tổ chức không bao giờ muốn điều này xảy ra.

Về hành động, tổ chức và những người quản lý có 2 công việc chính. Thứ nhất, thực hiện những gì đã cam kết về mặt truyền thông: khuyến khích những xung đột tích cực, cũng như đảm bảo tôn trọng sự tự tổ chức trong nhóm. Việc tổ chức hoặc những người quản lý trực tiếp đứng ra giải quyết xung đột không nên xảy ra, đó là cách can thiệp từ một hệ thống bên ngoài nhóm, khiến nhóm không thấy được tôn trọng về sự tự tổ chức; và có thể gây ra sự tiêu cực trong việc cam kết. Can thiệp của tổ chức và những người quản lý sẽ nhanh chóng giải quyết được xung đột ngay tức thì nhưng sẽ không có hiệu quả về mặt lâu dài; bởi việc này hình thành nên một kinh nghiệm xấu rằng xung đột không được khuyến khích bởi tổ chức. Hơn nữa, việc tự giải quyết xung đột trong nội tại của nhóm chính là con đường bước tới trạng thái tiếp theo – Norming. Thứ hai, tổ chức và những người quản lý cần trang bị cho nhóm *cách thức giải quyết xung đột*. Để có thể tự mình giải quyết xung đột một cách hiệu quả, nhóm cần có đủ kiến thức, kỹ thuật cần thiết – được đào tạo, huấn luyện và trợ giúp bởi tổ chức và những nhà quản lý: khi nào sử dụng *bỏ phiếu, lướt quyết định, brainstorming...* Tổ chức chỉ nên hướng dẫn, nhóm tự tổ chức liên chức năng có trách nhiệm tự thực hành. Nói chung, tổ chức và những người quản lý chỉ nên can thiệp nếu nhận biết đó là những xung đột tiêu cực và cần nhanh chóng ngăn chặn; nhưng điều này ít khi xảy ra.

Norming

Ở trạng thái này, nhóm tự tổ chức liên chức năng hoạt động trơn tru và có hiệu suất tốt do đã định hình được quy tắc ứng xử khi có xung đột xảy ra; nhóm quan tâm tới việc tạo ra giá trị cũng như việc cải tiến để có hiệu quả cao hơn.

Lúc này, về cơ bản, tổ chức và những người quản lý ít phải lo lắng về mặt quy trình, song vẫn có 2 công việc cần làm ở mức độ cao hơn. Thứ nhất, đào tạo, hướng dẫn, trợ giúp nhóm về cách cộng tác. Ví dụ, làm thế nào để tổ chức

buổi Retrospective hiệu quả, làm thế nào để chuyển những ý tưởng thành hành động thực sự... Thứ hai, trợ giúp và khuyến khích nhóm trong việc ra quyết định nhằm mang lại giá trị. Trong 2 giai đoạn đầu, giá trị lớn nhất nhóm tạo ra là quy trình, cách thức giải quyết xung đột để làm việc gắn bó cùng nhau nhằm hướng tới mục tiêu chung. Trong 2 giai đoạn sau, giá trị lớn nhất nhóm tạo ra chính là giá trị kinh doanh; khi nhóm càng lúc càng hướng tới mục tiêu một cách rõ ràng.

Performing

Đây là giai đoạn *nhóm tự tổ chức liên chức năng* đạt hiệu suất cao nhất, nhưng không có nghĩa là tổ chức và những người quản lý nên bỏ mặc nhóm; bởi đây là lúc nhóm có những nhu cầu cao, tương ứng với trình độ, hiệu suất và giá trị của nhóm.

Lúc này, nhu cầu lớn nhất của nhóm là được *coi trọng về giá trị tạo ra*. Nhóm có nhu cầu được nhìn thấy những giá trị đã và đang mang lại cho tổ chức hay người dùng cụ thể; nhóm rất vui vẻ và hân diện nếu những công việc đã chuyển giao mang lại lợi ích cho tổ chức hay người dùng; ngược lại, nhóm cảm thấy buồn nếu công sức đổ ra không phát huy hiệu quả. Việc lớn nhất tổ chức và những người quản lý cần làm là chia sẻ thông tin, tạo liên kết nhóm với toàn tổ chức, khuyến khích những ý tưởng và quản lý mục tiêu.

Việc chia sẻ thông tin và gắn kết nhóm với toàn tổ chức là việc rất quan trọng, giúp nhóm nhận ra giá trị mang lại, chính là động lực phát triển.

Tổ chức và những người quản lý cần khích lệ những ý tưởng được đưa ra từ nhóm bởi đó chính là nguồn cảm hứng để nhóm tiếp tục duy trì ở trạng thái Performing.

Tổ chức cần thực hiện việc quản lý mục tiêu tốt hơn. Trạng thái Performing rất tuyệt song có thể gây ra sự bùng nổ tiêu cực. Chính bởi nhu cầu duy trì trạng thái bằng việc liên tục sản sinh ý tưởng nhằm mang lại giá trị, mục tiêu chung có thể bị vi phạm hoặc chệch hướng.

	<i>Quản lý dẫn dắt nhóm</i>	<i>Nhóm tự quản lý</i>	
<i>Thiết lập định hướng chung</i>			
<i>Thiết kế nhóm</i>			
<i>Quản lý công việc và tiến độ</i>			
<i>Thực hiện công việc</i>			

**Trách nhiệm
quản lý**

**Trách nhiệm
của nhóm**

Hình 6.7: Trách nhiệm của người quản lý và nhóm

Quyền hạn càng lớn, trách nhiệm càng cao. Hầu hết những phương pháp nằm trong chiếc ô Agile đều cố gắng đẩy quyền hạn của nhóm lên mức cao nhất có thể, là *tự tổ chức*; đồng nghĩa với việc yêu cầu nhóm có trách nhiệm cao về công việc hoàn thành cũng như giá trị tạo ra. Phương pháp này đi ngược hoàn toàn với phương pháp phát triển phần mềm truyền thống khi người quản lý tiếp nhận gần như toàn bộ mọi công việc cũng như trách nhiệm trong nhóm. Nhưng phát triển phần mềm là công việc trí óc phức tạp, và những cá nhân ở trình độ đó đều luôn mong muốn nhận nhiều trách nhiệm hơn cũng như khát khao tạo ra giá trị cho tổ chức hay cuộc sống. Cách tiếp cận bằng việc chia nhỏ việc phát triển phần mềm thành nhiều giai đoạn độc lập như cách sản xuất bất cứ một mặt hàng công nghiệp nào đã trở nên lỗi thời. Và Agile cung cấp một phương pháp tiếp cận khác thông qua *nhóm tự tổ chức liên chức năng*.

HIỂU ĐÚNG

Nhóm liên chức năng là nhóm mọi thành viên làm nhiều công việc khác nhau.

Đây là hiểu nhầm rất phổ biến, dễ bắt gặp ở những người lần đầu tiếp xúc hoặc chỉ đọc qua Scrum guide, cho rằng nhóm liên chức năng gồm những thành viên có thể vừa lập trình, vừa kiểm thử và thiết kế... Thực tế không như vậy, không các nhân nào có đủ kiến thức và kỹ năng để thực hiện mọi công việc; đó là đòi hỏi phi thực tế và khiến chất lượng không đảm bảo. Mỗi cá nhân trong nhóm liên chức năng vẫn sở hữu kiến thức, kỹ năng riêng biệt và thực hiện chính công việc trong chuyên môn của mình. Những thành viên khác có thể trợ giúp trong những điểm *ghép nối* như lập trình viên có thể hỗ trợ tester trong việc thực hiện một số *free test case*.

Nhóm tự tổ chức liên chức năng cần những thành viên ở trình độ đồng đều.

Cùng với hiểu nhầm trên, rất nhiều người hoài nghi rằng “*Agile chỉ thành công trong những nhóm có chất lượng từng cá nhân cao (vì có thể làm được nhiều việc ở trình độ cao)*” và e dè trong việc thực hành Agile. Bất kỳ phương pháp phát triển phần mềm nào đều có sác xuất thành công cao hơn với một nhóm có trình độ cao và đồng đều; Agile cũng vậy, dù Agile vẫn cho khả năng thành công với những nhóm khác. Theo tôi, nhóm *tự tổ chức liên chức năng* chỉ cần những thành viên có sự dũng cảm, cam kết và quan trọng nhất là kỹ năng giao tiếp. Trình độ của từng thành viên đồng đều giúp nhóm nhanh chóng đi tới trạng thái *Performing*. Nếu không, con đường sẽ dài hơn; song thật may mắn là phương pháp *tự tổ chức* sẽ giúp những thành viên phát triển rất nhanh.

Nhóm tự tổ chức không cần người quản lý.

Về cơ bản là đúng. Nhóm *tự tổ chức* không cần người quản lý truyền thống – những người quản lý cách thức làm việc, việc thực thi từng công việc... trong nhóm; bởi mọi thành viên đều chia sẻ trách nhiệm này. Sẽ không có sự *quản lý trực tiếp* và vai trò của người quản lý tập trung vào *quản lý mục tiêu*: huấn luyện, định hướng, trợ giúp... nhóm đi đúng hướng nhằm đạt được mục tiêu chung.

TỔNG KẾT

Nhóm *liên chức năng* là nhóm tập hợp những cá nhân có đầy đủ kiến thức, kỹ năng cần thiết để cam kết và thực hiện một mục tiêu đề ra. Nhóm *liên chức năng* hoạt động như một đơn vị, những thành viên thường xuyên trao đổi, cộng tác và hỗ trợ lẫn nhau trong những hoạt động, công việc cần thiết nhằm đạt được mục tiêu chung.

Nhóm *tự tổ chức* là nhóm không tập trung quyền hạn vào một thành viên cụ thể; tự định nghĩa và thực hành cách thức thực hiện, công tác để đạt được mục tiêu đề ra; có thể tự đưa ra quyết định mà không phụ thuộc vào *hệ thống bên ngoài* nhóm.

Nhóm *tự tổ chức liên chức năng* là nhóm được hình thành bởi những cá nhân có đầy đủ kiến thức, kỹ năng cần thiết để hướng tới mục tiêu được đặt ra; thông qua việc thường xuyên trao đổi, cộng tác, hỗ trợ lẫn nhau, nhóm tự đưa ra quy trình, cách thức cũng như quyết định để đạt được mục tiêu chung và không chịu tác động của bất kỳ *hệ thống nào* ngoài nhóm.

Nhóm *tự tổ chức liên chức năng* luôn yêu cầu sự *dũng cảm, cam kết và kỹ năng cộng tác*. Nhóm *tự tổ chức liên chức năng* luôn trải qua những trạng thái *Forming, Storming, Norming* và *Performing* với những đặc trưng khác nhau. Vai trò của tổ chức và những người quản lý dành cho nhóm *tự tổ chức liên chức năng* vẫn rất lớn, giúp nhóm nhanh chóng đạt được và duy trì trạng thái *Performing*.

07

MỎ RỘNG
AGILE

Tất cả những phương pháp theo tư tưởng Agile đều sử dụng đơn vị *nhóm* và tập trung phát triển những lý thuyết, kỹ thuật để giúp nhóm Agile thực hành hiệu quả nhằm tạo ra giá trị. Để duy trì tính hiệu quả cũng như đảm bảo *Tuyên ngôn Agile* trong việc cộng tác, những phương pháp này đều khuyến nghị nhóm có kích thước nhỏ, dưới 10-15 thành viên. Kanban là phương pháp duy nhất không đề cập tới kích thước nhóm (và đảm bảo hiệu quả ít bị suy giảm khi kích thước nhóm tăng nhanh); nhưng về bản chất, Kanban không thực sự theo tư tưởng Agile. Scrum hoạt động tốt với nhóm có dưới 10 thành viên nhưng rất nhanh chóng bộc lộ vấn đề với quy mô nhóm lớn hơn, dù chỉ với 15 hay 20 thành viên.

Nhưng quy mô sản phẩm, quy mô việc phát triển phần mềm cũng như quy mô tổ chức luôn có nhu cầu mở rộng. Vậy nên, đã có một thời gian, Agile đứng trước sự hoài nghi về sự phù hợp với những tổ chức vừa và lớn. Nhưng thật may mắn, bằng những học thuyết và kỹ thuật phù hợp đã đưa Agile sang một làn sóng mới, *làn sóng mở rộng Agile (scaling Agile)* trên cả hai phương diện *phát triển sản phẩm* và *cấu trúc, vận hành tổ chức*.

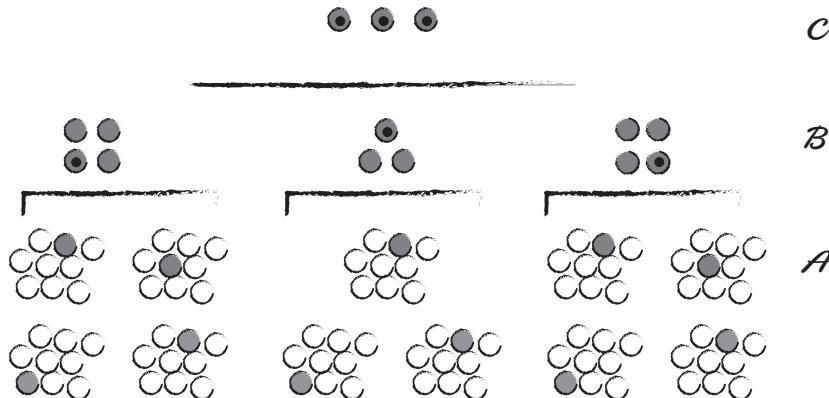
Trên phương diện phát triển sản phẩm, tổ chức có nhu cầu phát triển sản phẩm có quy mô lớn trong thời gian ngắn. Mặc dù mọi phương pháp Agile đều coi trọng việc chuyển giao những phần có giá trị lớn trước, theo nguyên lý 80/20; song điều gì xảy ra nếu 20% vẫn là một khối lượng công việc lớn với một nhóm Scrum 10 thành viên? Nhóm có thể mất 1-2 năm để hoàn thành và đánh mất cơ hội thị trường. Nhu cầu của tổ chức về một phương pháp với 100 nhân lực để hoàn thành sản phẩm trong 6 tháng là thực sự chính đáng.

Trên phương diện cấu trúc, vận hành tổ chức, khi Agile đã chứng minh được sự thành công trong việc phát triển phần mềm, tổ chức có nhu cầu tích hợp sâu hơn những tư tưởng, kỹ thuật Agile trong những hoạt động khác. Điều này đặc biệt tốt với những tổ chức công nghệ với mọi thứ cần được xoay quanh sản phẩm được tạo ra (như nhân sự, marketing...) và nảy sinh nhu cầu phản ứng nhanh với những thay đổi. Với những doanh nghiệp, tổ chức khác, việc áp dụng Agile cũng mang đến những hiệu quả trong việc hỗ trợ việc ứng biến với những thay đổi và cổ vũ việc học tập.

Thật ra làn sóng mở rộng Agile không mới trên thế giới, nhưng ở Việt Nam thì làn sóng này mới bắt đầu. Lý do đơn giản là chúng ta vẫn đang tiếp cận làn sóng đầu tiên về thực hành Agile trong những nhóm nhỏ. Song sẽ tốt hơn nếu tôi điểm qua những phương pháp này, để bạn biết cần tìm tới đâu để tham khảo khi đối mặt với bài toán mở rộng Agile.

SCRUM OF SCRUMS

Scrum of Scrums là một phương pháp mở rộng Scrum rất tự nhiên. Thay vì mở rộng nhóm Scrum thành một nhóm lớn hơn; tổ chức duy trì *nhóm Scrum vật lý* như một thành phần đơn vị và tạo ra sự liên kết giữa những thành phần này thông qua một *nhóm Scrum logic*.



Hình 7.1: *Scrum of Scrums*

Hình minh họa cho thấy quy mô dự án với 243 nhân sự. Thay vì mở rộng nhóm Scrum thành 243 thành viên (scale out); những nhân sự được tổ chức thành nhiều nhóm Scrum với 9 thành viên mỗi nhóm. Mỗi nhóm Scrum A có 1 thành viên tham gia vào nhóm Scrum ở mức cao hơn để kết nối, đồng bộ, cộng tác công việc giữa các nhóm và hình thành *Scrum of Scrums A* là *Scrum B*; mỗi thành viên trong nhóm Scrum B có 1 thành viên tham gia vào nhóm Scrum C ở mức cao hơn để kết nối, đồng bộ, cộng tác công việc giữa các nhóm.

Những nhóm Scrum A chính là một nhóm Scrum tiêu chuẩn với những thành viên luôn làm việc cùng nhau và gắn kết với nhau; tôi gọi là *nhóm Scrum vật lý*.

Những nhóm Scrum B, Scrum C bao gồm những thành viên không cố định; tôi gọi là *nhóm Scrum logic*.

Nhóm Scrum vật lý được tổ chức là một nhóm Scrum tiêu chuẩn, tuân theo những vai trò, sự kiện, artifact được định nghĩa trong Scrum guide. *Nhóm Scrum logic* thì khác và cần một chút lưu ý.

Thứ nhất, *nhóm Scrum logic* chỉ xác định số thành viên, không xác định thành viên cụ thể. Bởi trong nhóm Scrum tiêu chuẩn, mọi thành viên trong nhóm đều biết điều gì đang xảy ra trong nhóm, những gì đang được thực hiện, những khó khăn gặp phải và tiến độ hiện tại. Do đó, bất kỳ thành viên nào trong *nhóm Scrum vật lý* đều có thể tham gia *nhóm Scrum logic* và cung cấp những thông tin như nhau. Nhóm Scrum A thường sẽ không chỉ định An hay Bình sẽ là người tham gia nhóm Scrum B; bởi bất kỳ ai cũng có thể giam gia với hiệu quả như nhau; chế độ quay vòng là một ví dụ. Ngược lại, việc này cũng giúp mọi thành viên trong nhóm có trách nhiệm trong việc quản lý chung công việc của nhóm Scrum A. Tương tự như vậy với nhóm Scrum B.

Thứ hai, *nhóm Scrum logic* không có nhiều lý do để thực hiện mọi vai trò, sự kiện như nhóm Scrum tiêu chuẩn. Nhóm sẽ chỉ thực hiện nhanh Sprint Planning cũng như Daily Scrum để đồng bộ công việc với nhau, cũng như tìm ra những công việc đang gặp khó khăn khiến lộ trình có thể bị thay đổi. Việc xử lý những khó khăn gặp phải nói chung vẫn là trách nhiệm của *nhóm Scrum vật lý*.

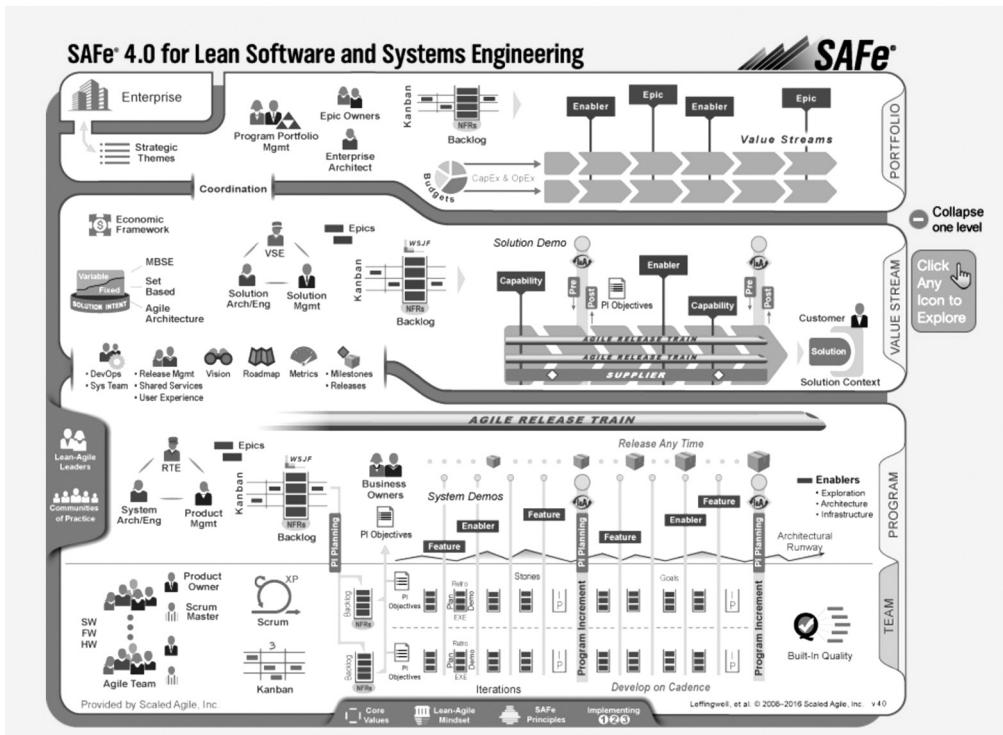
Để *Scrum of Scrums* hoạt động hiệu quả, việc giảm sự ràng buộc giữa các nhóm rất quan trọng. *Scrum of Scrums* không giới hạn việc cộng tác, trao đổi trực tiếp giữa những thành viên khác nhóm; song nếu việc này xảy ra với tần suất cao, hiệu suất cùng khả năng hướng tới Sprint Goal của nhóm Scrum giảm sút do có quá nhiều ràng buộc. Điều này nói thì dễ, song trong một hệ thống lớn cùng một phần cutting edge phức tạp thì không đơn giản. Khi đó vai trò của việc thiết kế cũng như phân rã chức năng trở nên quan trọng.

Để tăng sự độc lập, những nhóm Scrum A được thiết kế là *feature team*, làm việc trên một sản phẩm hoặc một nhóm chức năng cụ thể. Tôi không rõ nhóm thực hiện Firebase của Google được tổ chức theo phương pháp nào, song dù chia sẻ một tầm nhìn chung về sản phẩm, Firebase Database, Firebase Testlab... vẫn có sự phân mảnh lớn trong cách cấu hình, sử dụng (console hoặc API...) bởi họ được tổ chức theo *feature team*. Về mặt kiến trúc bạn có thể không hài lòng bởi việc tích hợp giữa những thành phần này không thực sự tự nhiên và nhất quán; song dưới góc độ phát triển sản phẩm, việc này cho hiệu quả cao bởi Google chỉ mất 12 tháng để cho ra đời một tập lớn dịch vụ hữu ích.

Scrum of Scrums cũng vậy, đây là cách mở rộng rất tự nhiên để tổ chức việc phát triển sản phẩm với quy mô lớn theo phương pháp *scale up* thay vì *scale out*. Những phương pháp khác ở dưới hầu như cũng chia sẻ chung tầm nhìn này, nhưng có sự trưởng thành cao hơn (và đương nhiên, phức tạp hơn).

SCALED AGILE FRAMEWORK – SAFe

Bạn có thể thấy, *Scrum of Scrums* khá tự nhiên và dễ hiểu; song tính chất tự nhiên và dễ hiểu của *Scrum of Scrums* lại mang đến sự ngờ vực. Lý do chính là hiểu biết và thói quen đã ăn sâu vào lịch sử phát triển phần mềm; với những dự án cần tới hàng trăm nhân sự, những tổ chức luôn cần một phương pháp hay công cụ quản lý phức tạp; và họ tin rằng sự phức tạp là điều hợp lý với dự án quy mô lớn, và đồng thời, tin tưởng rằng sự phức tạp chính là chìa khoá thành công. Vậy nên, thời kỳ đầu, không ai tin rằng *Scrum of Scrums* sẽ giải quyết được bài toán; chỉ vì nó quá đơn giản. Bạn còn nhớ khoảnh khắc chiếc iPhone đầu tiên được giới thiệu không? Không ai tin rằng một chiếc điện thoại có thể làm được mọi thứ chỉ với một nút Home. Sự đơn giản luôn bị hoài nghi.



Hình 7.2: SAFe

Và Scaled Agile Inc. đã thiết kế ra một framework phức tạp hơn giúp mọi người đập tan lo ngại; đó là SAFe – Scaled Agile Framework.

SAFe hướng trọng tâm vào sự cộng tác, điều chỉnh và chuyển giao với nhiều nhóm thực hành Agile trong một tổ chức lớn, thường phù hợp với 8 nhóm thực hành Agile với khoảng 100 nhân sự. SAFe được giới thiệu lần đầu với phiên bản 1.0 vào năm 2011. Phiên bản mới nhất, 4.0, được đưa ra trong năm 2016, với tên gọi *SAFe 4.0 for Lean Software and Systems Engineering* hỗ trợ việc mở rộng tới quy mô hàng ngàn nhân sự.

Tư tưởng chính của SAFe là phân chia thành nhiều mức (level) khác nhau, bao gồm: *Team, Program, Portfolio, Value Stream*; cộng tác với nhau thông qua *Agile Release Train*.

Team

Ở mức thấp nhất vẫn là nhóm thực hành Agile. Không như *Scrum of Scrums* bao gồm nhiều nhóm thực hành Scrum, team trong SAFe có nhiều phương pháp thực hành khác nhau như XP, Lean... nhưng vẫn là nhóm liên chức năng, đảm bảo nhóm có đủ kỹ năng để hoàn thành một công việc cụ thể. Các nhóm cũng không hẳn được thiết kế là feature team; SAFe chấp nhận những nhóm đơn chức năng như nhóm phụ trách kiến trúc, nhóm testing... Những nhóm này cũng không hoạt động độc lập, họ cộng tác trên Backlog ở mức cao hơn, gọi là *Program Backlog* cùng công việc được đồng bộ giữa các nhóm thông qua *Agile Release Train*.

Program

Ở mức cao hơn, những (5 – 8) nhóm thực hành Agile cùng cộng tác với nhau thông qua *Program*. Họ cùng chia sẻ với nhau Program Backlog và những Iteration có khoảng thời gian cố định, ví dụ 2 tuần. Sau mỗi Iteration, mỗi nhóm đã hoàn thành những phần tăng trưởng cụ thể. Sau một số (3 – 5) Iteration, những phần tăng trưởng được của từng nhóm tạo thành *Program Increment*, là phần tăng trưởng của Program, được xác định trước đó thông qua *Innovation and Planning Iteration*. Tổ chức cũng duy trì việc demo những phần tăng trưởng ở level Program sau mỗi số Iteration xác định, gọi là *System Demo*.

SAFe mô tả việc tích hợp, đồng bộ phần tăng trưởng giữa các nhóm ở mức Program được thực hiện sau một số Iteration, nhưng không giới hạn việc chuyển giao chỉ được thực hiện tại những thời điểm *System Demo*. Việc chuyển giao tốt nhất nên được thực hiện bất cứ khi nào thông qua hệ thống *Continuous Delivery*.

Portfolio

Mức tiếp theo là Porfolio, bao gồm nhiều *Strategic Theme*. *Theme* được tối ưu để cập nhật trong chương 3 về Scrum, là tập hợp những User Story mang lại một giá trị và mục tiêu cụ thể. Những theme hoàn toàn có thể tồn tại ở mức Team hoặc Program; Portfolio gồm *Strategic Theme* – là những theme ở mức cao, mang ý nghĩa điều

hướng tới những giá trị, mục tiêu chung của sản phẩm hơn là những giá trị từ chức năng cụ thể.

Value Stream

Ở phiên bản 3.0, SAFe chỉ dừng lại ở Portfolio với sự mở rộng tới 8 – 10 Team gồm 100 nhân sự. Phiên bản 4.0 bổ sung thêm mức *Value Stream*, được thiết kế ở mức tổ chức, doanh nghiệp lớn. Value Stream bao gồm nhiều thành phần, giá trị cốt lõi, chiến lược để chuyển giao giá trị tới người dùng...

Tôi chưa có cơ hội thực sự thực hành SAFe nên không thể đánh giá hiệu quả, nhưng bản thân có một chút hoài nghi trong cách tiếp cận, cũng như sự sâu sắc của SAFe, nhất là cách *Scaled Agile Inc.* tiếp cận việc mở rộng và thay đổi qua từng phiên bản.

LESS

Một framework khác để mở rộng Scrum là *LeSS – Large-Scale Scrum* được xây dựng



Bạn sẽ thấy SAFe chi tiết và phức tạp hơn so với Scrum of Scrums. Scrum of Scrums chỉ đơn thuần xác định việc đồng bộ ở mức cao hơn, và mọi mức cao hơn đều có cách tiếp cận giống nhau. SAFe chỉ phân chia thành 4 mức và có tên gọi, cách tiếp cận cụ thể cho từng mức với tư tưởng chung là mức cao hơn thực hiện việc đồng bộ, điều chỉnh trong khoảng thời gian dài hơn, và quản lý những công việc ở mức ít cụ thể hơn. Thật ra nếu bạn đã quen với những phương pháp quản lý dự án phần mềm truyền thống, thì bạn sẽ thấy SAFe rất dễ hiểu; bởi SAFe có xu hướng tiếp cận kiểu top-down hơn là bottom-up theo tư tưởng Agile thông thường. Chính vì vậy, có rất nhiều tranh cãi xung quanh SAFe; một số ưa thích, cổ vũ và coi đây là phương pháp tuyệt vời để mở rộng Agile trong tổ chức lớn; song cha đẻ của Scrum, Ken Schwaber thì không nghĩ như vậy. Ken cho rằng SAFe được tạo ra bởi những chàng trai theo tư tưởng RUP (Rational Unified Process), những người nghĩ rằng chỉ một phương pháp được xác định rõ ràng có thể giải quyết bài toán phát triển phần mềm của mọi tổ chức – điều mà RUP đã được chứng minh sự thất bại sau một thời gian dài.

và cổ vũ bởi hai chuyên gia hàng đầu về Agile là Bas Vodde và Craig Larman, hoàn chỉnh lần đầu tiên vào năm 2013, với nguyên lý chủ đạo *Large-Scale Scrum is Scrum (mở rộng Scrum là Scrum)*.

LeSS được mô tả như sau:

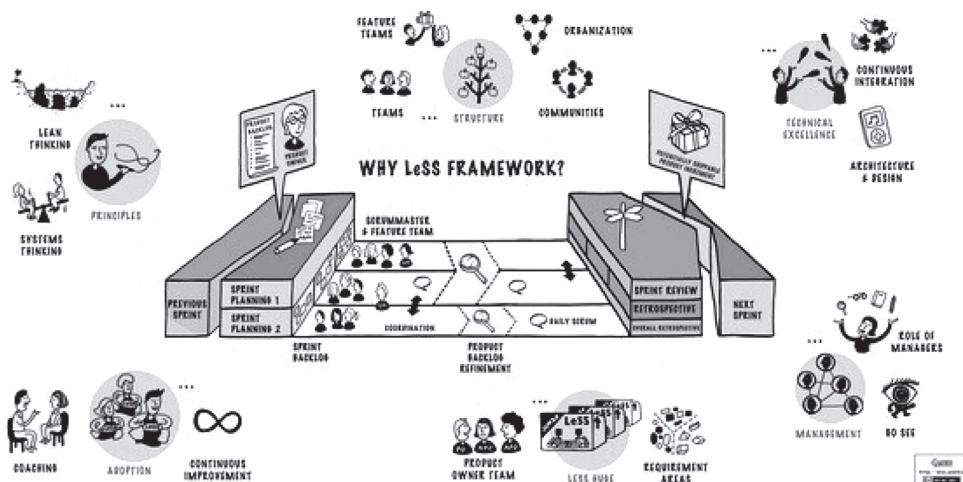
Cấu trúc: LeSS được cấu trúc gồm nhiều nhóm phát triển, đều là nhóm tự quản liên chức năng và làm việc với nhau trong một thời gian dài – chính là nhóm Scrum. ScrumMaster vẫn giữ trách nhiệm và làm những công việc được mô tả chính xác trong Scrum guide, nhưng hoạt động trong không gian rộng hơn, 1 – 3 nhóm phát triển. Product Owner cũng không hoạt động trong phạm vi một nhóm mà phụ trách chung trong toàn sản phẩm.

Sản phẩm: LeSS giữ nguyên cách thực hiện của Scrum, nên chỉ có duy nhất một Product Owner sở hữu và quản lý một Product Backlog và trợ giúp những nhóm Scrum làm việc với khách hàng và chủ đầu tư như trong Scrum. Mỗi nhóm Scrum sẽ thực hiện phần tăng trưởng cụ thể nhưng đều hướng tới mục tiêu chung là đặt giá trị cung cấp cho khách hàng là trung tâm. Tất cả các nhóm Scrum đều thống nhất một DoD chung nhưng có thể mở rộng và cụ thể hóa DoD cho nhóm của mình.

Sprint: LeSS thực hiện theo đúng Sprint – trái tim của Scrum. Tất cả các nhóm Scrum đều thực hiện chung một Iteration, có chung thời điểm bắt đầu và kết thúc Sprint. Công việc phức tạp nhất là Sprint Planning bởi theo Scrum cả nhóm Scrum thực hiện việc lập kế hoạch song nhóm LeSS gồm hàng trăm nhân sự là quá lớn, việc này không khả thi. LeSS chia việc thực hiện Sprint Planning thành hai phần: *Sprint Planning One* được thực hiện với nhóm LeSS nhằm lựa chọn ra những item nào sẽ được thực hiện trong từng nhóm Scrum; *Sprint Planning Two* được thực hiện trên từng nhóm Scrum và những item được lựa chọn theo cách truyền thống. Sau đó, Sprint tiếp diễn theo cách thông thường với từng nhóm thông qua Daily Scrum, Continuous Integration... cùng việc cộng tác chéo giữa những thành viên khác nhóm Scrum nếu cần. Sẽ chỉ có một buổi Sprint Review duy nhất cho toàn nhóm LeSS khi kết thúc Sprint; nhưng mỗi nhóm Scrum có thể có Sprint Retrospective cho riêng mình, trước khi nhóm LeSS thực hiện Retrospective cho thông qua những hoạt động Open Space.

LeSS hoạt động gần như chính xác theo cách Scrum được định nghĩa, chỉ giải quyết bài toán hàng chục nhân sự bằng cách cấu trúc thành những nhóm Scrum; phân tách những hoạt động như Sprint Planning, Sprint Retrospective thành hai mức khác nhau, cho nhóm Scrum và nhóm LeSS. Song cách mở rộng này không thể hoạt động với dự án lớn. Gần đây, LeSS cung cấp một phiên bản

mới, LeSS Huge là sự mở rộng của LeSS cho dự án gồm hàng trăm nhân sự, gồm nhiều *Requirement Area* như một cách giúp những nhóm Scrum hiểu được yêu cầu và hoạt động hiệu quả hơn.



Không giống như SAFe xây dựng việc mở rộng Scrum thông qua nhiều mức được tổ chức chặt chẽ (ngay ở mức Team cũng không thực hiện theo đúng Scrum), LeSS cố gắng giữ Scrum ở mức nguyên bản, chỉ giải quyết những vấn đề với số lượng lớn nhân sự tham gia dự án. Chính vì vậy, những người hâm mộ Scrum thường thích LeSS và hay nói rằng “*SAFe is safe? LeSS is more*” (một cách chơi chữ, “*SAFe có an toàn? LeSS có nhiều hơn*”). Theo tôi, cách chơi chữ trong việc đặt tên 2 framework này cũng cho chúng ta biết xu hướng tiếp cận của tác giả: SAFe hướng tới sự *an toàn* thông qua việc kiểm soát bởi việc quản lý phân cấp; LeSS hướng tới sự đơn giản, ít hơn trong việc tổ chức. Nói như vậy không có nghĩa là LeSS tốt hơn SAFe, tất cả đều là cách mở rộng Scrum – một *management framework*, chỉ có cách tiếp cận khác nhau, và khác nhau về mặt truyền thông.

NEXUS

Một phương pháp mở rộng Agile khác được giới thiệu trong những năm gần đây là Nexus, do chính tác giả của Scrum, Ken Schwaber đưa ra, với tư tưởng giữ việc mở rộng Agile trở nên đơn giản và tinh gọn như bản chất của Agile.

Nexus được hiểu là một thành phần phát triển phần mềm, gồm nhiều nhóm thực hành Scrum, thực hiện theo những chỉ dẫn Scrum và bổ sung thêm một số thành phần sau.

Nexus bổ sung thêm một vai trò là *Nexus Integration Team* chịu trách nhiệm kết nối, huấn luyện việc thực hành Nexus và Scrum. Khác với Scrum of Scrums, *Nexus Integration Team* là một nhóm cố định, giải quyết những vấn đề liên nhóm về mặt thực hành, kỹ thuật cũng như quy trình được sử dụng.

Nexus bổ sung thêm *Nexus Sprint Backlog* là một artifact mới, chính là Product Backlog duy nhất cho cả nhóm Nexus. Khi những nhóm Scrum vận hành theo cùng một Iteration, Nexus giới thiệu *Integrated Increment* là phần tăng trưởng được tạo thành bởi cả nhóm Nexus, được hình thành sau khi nhóm Nexus hoàn thành *Nexus Sprint Goal* – mục tiêu chung trong một Sprint của nhóm Nexus. Để đánh giá việc hoàn thành phần tăng trưởng, một *Common Definition of Done* cũng được đưa ra, và cụ thể hóa cho từng nhóm Scrum.

Để đồng bộ nhóm Nexus, một số sự kiện được đưa ra bao gồm *Refinement*, *Nexus Daily Scrum*, *Nexus Sprint Review* và *Nexus Sprint Retrospective*.

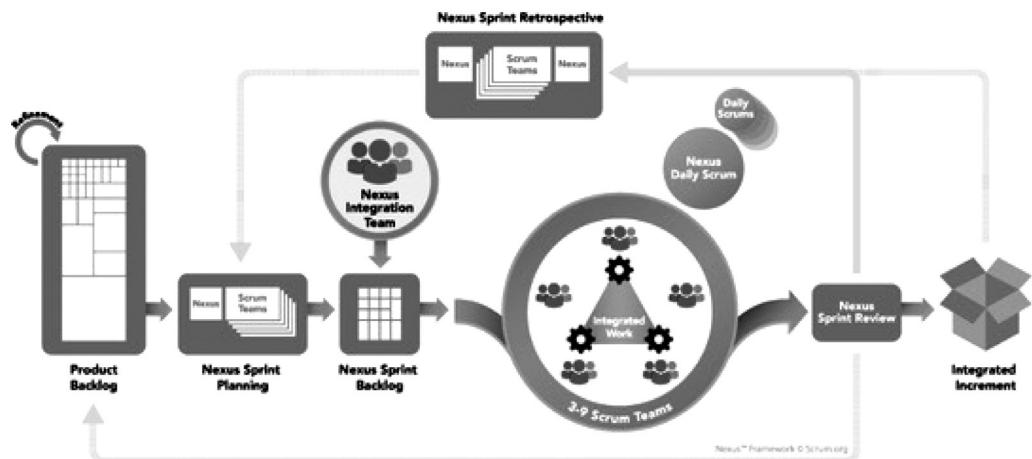
Nexus Framework Process



Hình 7.3: Nexus framework process

Về cơ bản, Nexus không (hoặc chưa) chạy theo việc mở rộng Agile ở quy mô lớn như SAFe 4.0 hay LeSS Huge, Nexus mở rộng Agile ở quy mô dưới 8 nhóm thực

hành Scrum, tương tự SAFe và LeSS. Tư tưởng chung của Nexus là tạo thêm một thành phần nữa để kết nối và tích hợp giữa những nhóm thực hành Scrum, biểu hiện ở artifact với *Nexus Sprint Backlog*, *Nexus Sprint Goal* hay sự kiện với *Nexus Daily Scrum*, *Nexus Sprint Review*... được thực hành bởi một nhóm cố định là *Nexus Integration Team* (khác với *Scrum of Scrums*).

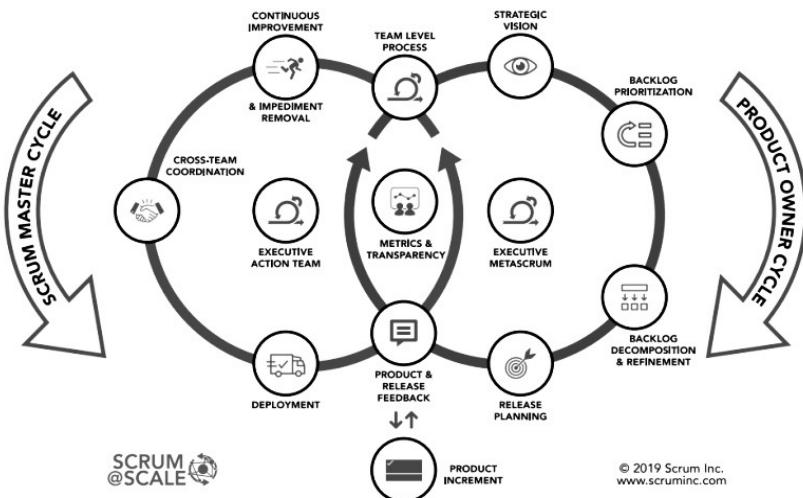


Hình 7.4: Nexus framework

SCRUM@SCALE

Một tác giả nữa của Scrum là Jeff Sutherland trong những năm gần đây giới thiệu một phương pháp mở rộng Scrum khác là Scrum@Scale với việc mở rộng tổ chức tới quy mô không giới hạn.

Scrum@Scale thiết lập một hướng dẫn rõ ràng về 2 cycle: Product Owner tập trung giải quyết và đồng bộ về *what* - những sản phẩm tạo ra giá trị; Scrum Master tập trung giải quyết và động bộ về *how* - cách thực hành Scrum để sản sinh giá trị nhanh nhất.



Hình 7.5: Scrum@Scale framework

Điểm khác biệt của Scrum@Scale là mỗi nhóm Scrum được coi như một tế bào trong tổ chức với sự kết nối thông tin nhanh nhất có thể: một vấn đề trong nhóm Scrum phát sinh sau buổi họp Daily Scrum có thể được các C level nhận biết và giải quyết chỉ sau một vài cuộc họp ở mức cao hơn. Đây chính là cơ sở để Jeff đưa ra khái niệm mở rộng Agile trong tổ chức có quy mô không giới hạn. Điều này quá tuyệt vời về mặt lý thuyết; song cần kiểm chứng trong thực tế vì dù sao Scrum@Scale cũng còn quá mới mẻ.

HIỂU ĐÚNG

Chúng ta không thể mở rộng Agile vì vi phạm Tuyên ngôn Agile về Cá nhân và sự tương tác hơn là quy trình và công cụ.

Một lý luận có vẻ rất xác đáng, bởi khi mở rộng Agile mọi phương pháp cần bổ sung thêm quy trình và công cụ. Thật ra chúng ta không có cách nào khác để đảm bảo sự cộng tác nếu không thêm vào quy trình và công cụ với những nhóm lớn. Song hãy nhớ rằng, Tuyên ngôn Agile kết nối hai vấn đề bởi chữ “*hơn là*” kèm một ghi chú chung “*Mặc dù các điều bên phải vẫn còn giá trị, nhưng chúng tôi đánh giá cao hơn các mục ở bên trái.*,” có nghĩa là việc mở rộng Agile sẽ vẫn đảm bảo Tuyên ngôn Agile nếu chúng ta vẫn thực sự đề cao cá nhân và sự tương tác hơn là quy trình và công cụ. Scrum of Scrums là một ví dụ, có rất ít những ràng buộc quy trình phức tạp phát sinh, và việc cộng tác giữa các nhóm vẫn được đề cao thông qua *Daily Scrum of Scrums*.

SAFe là phương pháp không tốt để mở rộng Agile.

Tôi tin bạn sẽ nghĩ tới điều này nếu nhìn vào những chỉ trích của chính cha đẻ của Scrum, Ken Schwaber, dành cho SAFe, và thậm chí ông còn cố gắng đưa ra một phương pháp tiếp cận khác là Nexus. Cá nhân tôi thích cách tiếp cận của Nexus hơn, song có hai điều không thể phủ nhận; một là, Scrum cũng chỉ là một phương pháp nằm trong *chiếc ô Agile*, và chúng ta đang nói về việc mở rộng Agile; hai là, số những doanh nghiệp áp dụng thành công SAFe tăng nhanh qua từng năm. Điều đó cho thấy SAFe vẫn đang tỏ rõ sự hiệu quả của mình.

TỔNG KẾT

Scrum of Scrums là một phương pháp mở rộng rất tự nhiên của Scrum. Thay vì mở rộng nhóm Scrum thành một nhóm lớn hơn; tổ chức duy trì nhóm Scrum vật lý như một thành phần đơn vị và tạo ra sự liên kết giữa những thành phần này thông qua một nhóm *Scrum logic*.

SAFe hướng trọng tâm vào sự cộng tác, điều chỉnh và chuyển giao với nhiều nhóm thực hành Agile trong một tổ chức lớn, thường phù hợp với 8 nhóm thực hành Agile với khoảng 100 nhân sự. Tư tưởng chính của SAFe là phân chia thành nhiều mức (level) khác nhau, bao gồm: *Team, Program, Portfolio, Value Stream*; cộng tác với nhau thông qua *Agile Release Train*.

LeSS hoạt động gần như chính xác theo cách Scrum được định nghĩa, chỉ giải quyết bài toán hàng chục nhân sự bằng cách cấu trúc thành những nhóm Scrum; phân tách những hoạt động như *Sprint Planning, Sprint Retrospective* thành hai mức khác nhau, cho nhóm Scrum và nhóm LeSS.

Tư tưởng chung của *Nexus* là tạo thêm một thành phần nữa để kết nối và tích hợp giữa những nhóm thực hành Scrum, biểu hiện ở *Artifact* với *Nexus Sprint Backlog, Nexus Sprint Goal* hay sự kiện với *Nexus Daily Scrum, Nexus Sprint Review...* được thực hành bởi một nhóm cố định là *Nexus Integration Team*.

Scrum@Scale tập trung giải quyết bài toán thông suốt thông tin trong tổ chức dựa trên giả định rằng quyết định đúng đắn được đưa ra nhanh và chính xác khi thông tin thông suốt và rõ ràng.

08

TỐ CHÚC
AGILE

Trong những phần nội dung trước, chúng ta đã thấy cách thực hành Agile trong việc phát triển phần mềm, từ quy mô nhóm nhỏ với Agile tới quy mô lớn với mở rộng Agile. Song sức mạnh và hiệu quả của Agile còn mạnh mẽ hơn nữa nếu những tư tưởng Agile được lan tỏa và thực hành tốt trong toàn bộ tổ chức.

Trong chương này, tôi đề cập tới việc hình thành tổ chức Agile theo hai cách tiếp cận: *xây dựng tổ chức Agile* và *chuyển đổi tổ chức sang Agile*.

XÂY DỰNG TỔ CHỨC AGILE

Khi nói về *xây dựng tổ chức Agile*, tôi muốn hướng tới việc tiến hành tại những tổ chức vừa và nhỏ, nơi cấu trúc tổ chức, công việc, quy trình... chưa được định hình đầy đủ và rõ ràng; đó thường là startup, những doanh nghiệp nhỏ và vừa. Bởi với một tổ chức lớn và có độ trưởng thành nhất định, câu chuyện sẽ là *chuyển đổi tổ chức sang Agile*, và phức tạp hơn.

Với startup, giờ đây, việc xây dựng tổ chức Agile đã trở thành công thức để tránh những thất bại. Thành công luôn bao gồm nhiều yếu tố, song không hoặc thất bại trong việc xây dựng tổ chức Agile, gần như là cách nhanh nhất đưa startup đến bên bờ vực. *The Lean startup*, cuốn sách của Eric Ries nằm trong danh sách gối đầu giường của mọi nhà khởi nghiệp.

Xây dựng một tổ chức Agile từ trang giấy trắng thường không gặp nhiều khó khăn như việc chuyển đổi tổ chức, song vẫn có hàng tá những công việc phải làm. Theo tôi, đây là những công việc quan trọng nhất.

Cấu trúc tổ chức xoay quanh khách hàng và sản phẩm

Đâu là lợi thế lớn nhất của tổ chức nhỏ và vừa so với những tổ chức lớn? Sự linh hoạt trong cấu trúc tổ chức để nhanh chóng phản hồi với sự thay đổi và sản sinh giá trị. Những tổ chức vừa và lớn, thường được cấu trúc thành những bộ phận chuyên trách với chuyên môn cụ thể nhằm tối đa hóa sự kiểm soát phục vụ công việc quản lý.

Cấu trúc tổ chức đặt *khách hàng làm trung tâm* (*customer centric organization*) không phải là điều mới mẻ, bởi hầu hết tổ chức đều nhận ra rằng tối đa hóa giá trị mang lại cho khách hàng chính là mục tiêu và là công cụ hiệu quả nhất mang lại doanh thu; đặt khách hàng và giá trị mang lại cho khách hàng làm trung tâm chính là lợi thế cạnh tranh cốt lõi. Song điều cần làm là phủ bóng triết lý

đó trong toàn bộ tổ chức, cũng như cấu trúc tổ chức cho phù hợp nhằm đạt được hiệu quả mong muốn. May mắn là, việc này không quá khó với những tổ chức nhỏ và vừa; điều quan trọng là họ nhận thức được công việc cần phải làm. Thông thường, những bộ phận bán hàng, marketing, hỗ trợ... thường sẽ "gần" khách hàng hơn, họ chính là những người hiểu rõ nhu cầu cũng như cách thức khiến khách hàng thoả mãn bằng những giá trị được tạo ra. Điều tổ chức cần làm là kéo khách hàng gần hơn nữa, với đội phát triển và vận hành; và sử dụng giá trị mang lại cho khách hàng là thước đo. Điều này giúp cho tổ chức có *tiếng nói chung*, cùng nhìn về một mục tiêu chung thay vì những mục tiêu đơn lẻ như KPI cụ thể như số khách hàng của bộ phận bán hàng, số bug tìm thấy của bộ phần kiểm thử, số dòng source code viết được của bộ phận phát triển... KPI cụ thể cho từng bộ phận, từng công việc không sai; song quá để tâm vào KPI có thể dẫn tới việc lạc hướng trong mục tiêu chung. Trong nguồn lực hạn chế, việc tạo ra một chức năng được sử dụng bởi 1% người dùng, nói chung, lãng phí hơn so với tập trung nguồn lực đảm bảo chất lượng của những chức năng khác.

Cấu trúc tổ chức đặt khách hàng làm trung tâm là cách thức thiết lập hành vi tổ chức để mọi hành động, công việc được điều hướng bởi giá trị mang lại cho khách hàng. Song, cấu trúc này đòi hỏi tổ chức phải có *khách hàng*. Đó có thể là những đối tác, người dùng trả tiền cho sản phẩm, dịch vụ hoặc những người đại diện cho tiếng nói của họ. Điều gì xảy ra nếu trong giai đoạn đầu, tổ chức không có *khách hàng* – những người thực sự sử dụng hoặc trả tiền cho sản phẩm? Trên thế giới có hàng trăm tình huống như vậy xảy ra mỗi ngày, tại những startup, những tổ chức bắt đầu xây dựng với một ý tưởng sản phẩm, không phải từ một đơn đặt hàng. Lúc này, cấu trúc tổ chức gần như được định hình đặt sản phẩm làm trung tâm (*product centric organization*) theo cách rất tự nhiên. Nhưng một tổ chức thực sự Agile cần nhanh chóng nhận ra việc cần thiết phải chuyển đổi từ tổ chức đặt sản phẩm làm trung tâm thành tổ chức đặt khách hàng làm trung tâm. Điều này đòi hỏi tổ chức nhanh chóng đưa ra một *sản phẩm khả thi tối thiểu* – *MVP (Minimum Viable Product)* – nhằm kiểm nghiệm những ý tưởng trước khi bước tiếp. MVP cũng giúp tổ chức nhận tiếp cận khách hàng và mang lại dữ liệu cho việc phát triển tiếp theo. Thông qua MVP, tổ chức cũng có thể lựa chọn việc đi tiếp với việc định hình cấu trúc *sản phẩm làm trung tâm* hay *khách hàng làm trung tâm*.



Hình 8.1: Tổ chức đặt khách hàng làm trung tâm và tổ chức đặt sản phẩm làm trung tâm

Thực hành Agile

Thực hành Agile trong tổ chức nhỏ và vừa thường không gặp nhiều khó khăn, bởi hầu hết những tổ chức này đều được xây dựng từ một nhóm nhỏ và mở rộng dần quy mô khi trưởng thành. Vấn đề là tổ chức cần thực hành đúng Agile, càng sớm càng tốt.

Câu chuyện của những tổ chức nhỏ và vừa là họ thường không thực hành đúng Agile, và nguyên nhân thường do người đứng đầu. Với quy mô nhỏ, người đứng đầu thường có xu hướng quản lý *command and control* dưới bàn tay của mình; họ cố gắng kiểm soát mọi thứ, điều này hoàn toàn khả thi ở quy mô nhỏ với 5 hay 10 nhân sự. Và mọi chuyện bắt đầu rối tung khi quy mô tổ chức tăng lên. Khi đó, mọi sự bắt đầu vượt ra khỏi tầm kiểm soát trong khi nhân viên đã quen với việc thực hiện những chỉ thị cụ thể do bị triệt tiêu khả năng tự tổ chức; từ đó hệ thống quản lý phân cấp bắt đầu được áp dụng và ngày càng phức tạp thêm. Thông thường, những tổ chức nhỏ, startup thường cổ vũ cho phong trào *người hùng* trong giai đoạn đầu – những người đưa ra mọi quyết định, quản lý mọi công việc trong nhóm hoặc tổ chức. Tôi không phản đối phong cách này, vì thực sự mang lại hiệu quả cao và phù hợp; con người muốn đi nhanh, cần đi một mình; tổ chức muốn đi nhanh, cần đi theo *người hùng*. Song để đi dài, con người cần đi cùng nhau; tổ chức muốn phát triển và trường tồn, cần cổ vũ cho việc cộng tác giữa các cá nhân và cổ vũ phong trào làm việc nhóm. Tôi nhận thấy vấn đề thường gặp với những tổ chức nhỏ là khi họ muốn phát triển nóng, phong trào người hùng được áp dụng triệt để; và khi quy mô tổ chức tăng lên, những người hùng này sẽ được đề bạt dần lên những vị trí cao hơn như một phần thưởng; việc quản

lý phân cấp được định hình; và phong trào người hùng tiếp tục được cổ vũ ở quy mô rộng và sâu hơn; cùng lúc đó là hệ thống *command and control* phủ bóng lên toàn bộ tổ chức ở quy mô mới. Đó thường là khó khăn, và cạm bẫy của những tổ chức nhỏ; đến khi quy mô tổ chức lớn hơn, họ phải giải quyết bài toán *chuyển đổi tổ chức*, thường phức tạp và tốn kém hơn.

Giải pháp hiệu quả nhất, là tận dụng lợi thế của quy mô nhỏ, thực hành đúng Agile sớm nhất có thể.

Nhóm tự tổ chức là yếu tố cần quan tâm nhất. Agile được xây dựng trên nền tảng nhóm *liên chức năng tự tổ chức*, với những tổ chức nhỏ, yếu tố *liên chức năng* thường được hình thành khá tự nhiên, song *tự tổ chức* thì không. Tổ chức cần kiên trì khi trao quyền tự chủ cho nhóm, phong trào người hùng vẫn có thể được cổ vũ theo một cách khác; từng cá nhân là người hùng trong việc xây dựng nhóm, xây dựng tổ chức, không tập trung vào một cá nhân cụ thể - người đưa ra mọi quyết định.

Phương pháp đúng là yếu tố quan trọng nhất. Agile về cơ bản không khó hiểu song không dễ thực hành đúng và thành thực, và khó để chuyển hóa thực sự thành tư tưởng trong tổ chức. Định hình cách làm đúng ngay từ ban đầu có ý nghĩa và hiệu quả lớn. Đơn giản nhất từ việc đảm bảo mọi cuộc họp Daily Scrum được thực hiện dưới 15 phút; duy trì sự kiện Retrospective được thực hiện trong không khí và tinh thần xây dựng, cũng như cam kết thực hiện những hành động sau đó. Phức tạp hơn là việc cam kết trao quyền cho nhóm, cam kết trao quyền quản lý Product Backlog duy nhất cho Product Owner. Những thực hành này nghe qua khá dễ hiểu, song theo quan sát của tôi, rất nhiều tổ chức không thực hành đúng. Những người quản lý thường tham gia vào cuộc họp hàng ngày, đưa ra hàng loạt những chỉ đạo, thảo luận khiến cuộc trao đổi kéo dài hàng giờ; và ngày mai, họ lại đưa ra một ý tưởng khác. Những nhà quản lý cũng có xu hướng can thiệp thô bạo vào việc yêu cầu Product Owner thực hiện một chức năng không có trong kế hoạch vào giữa Sprint và đổ lỗi cho nhóm khi không có một Sprint thành công. Những ví dụ tôi đưa ra, tất nhiên, vi phạm những nguyên tắc trong Scrum, không hẳn trong Agile. Song ý tưởng chính là, nếu tổ chức lựa chọn Scrum cho nhóm, hãy thực hiện đúng Scrum; ngược lại, hãy chọn một phương pháp khác và thực hiện đúng nó.

Sai lầm lớn nhất của những tổ chức nhỏ mới thực hành Agile, là họ hiểu rằng Agile bản thân phải rất linh hoạt và cố gắng thay đổi để phù hợp với cách làm cũng như cấu trúc hiện tại của tổ chức. Nhưng việc làm đúng là: *thay đổi hành vi tổ chức để thực hành đúng Agile*. Thay đổi cách áp dụng và thực hành Agile chỉ được phép thực hiện khi tổ chức có đầy đủ bằng chứng rằng nên thay đổi thông qua dữ liệu lịch sử của việc thực hành đúng Agile.



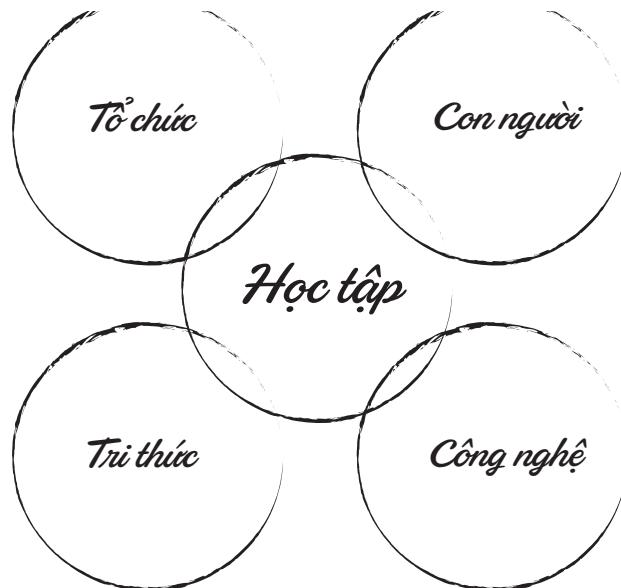
Hãy xây dựng vị trí Agile Coach hoặc ScrumMaster sớm nhất có thể. Phần nhiều tổ chức không quan tâm tới vị trí này trong giai đoạn đầu bởi họ thấy Agile hay Scrum khá dễ hiểu và thực hành. Các tổ chức thường tuyển dụng vị trí này khi quy mô tổ chức tăng lên và họ nhận thấy hàng loạt những khó khăn phát sinh. Song nếu chi phí không phải là vấn đề, tôi khuyến nghị tổ chức nên xây dựng vị trí này, hoặc tìm đến những chuyên gia tư vấn về Agile sớm nhất có thể. Việc thực hành đúng Agile ngay từ ban đầu giúp việc mở rộng tổ chức Agile sau này dễ hơn nhiều.

Xây dựng tổ chức học tập

Xây dựng tổ chức học tập là điều quan trọng bậc nhất trong quá trình xây dựng tổ chức Agile, bất kể quy mô tổ chức, dù nhỏ hay lớn. *Tổ chức học tập là tổ chức có đủ kỹ năng để tạo ra, thu nhận và chuyển giao kiến thức trong tổ chức, từ đó thay đổi hành vi để phù hợp với những kiến thức và quyết định mới.*

Agile được xây dựng dựa trên lý thuyết thực nghiệm, với giả định rằng không có một công thức chung phù hợp cho mọi bài toán hay mọi tổ chức (và có ai ngờ ngờ giả định này không đúng?). Agile, vì thế, không cung cấp lời giải bài toán cho tổ chức, thay vào đó cung cấp phương pháp luận thực nghiệm và quyết định dựa trên dữ liệu, trải nghiệm quá khứ cùng thông tin hiện tại, và liên tục đưa ra quyết định phù hợp với hiện tại. Do đó, việc tạo ra kiến thức và sử dụng kiến thức để thay đổi hành vi trong tổ chức là việc đặc biệt quan trọng.

Việc xây dựng tổ chức học tập trước hết phải được thực hiện thông qua cam kết xây dựng tổ chức, cổ vũ và tạo không gian cho việc học tập của cá nhân, của nhóm và của cả tổ chức. Điều này thật ra không khó thực hiện, rất nhiều tổ chức cung cấp những khoá học trực tuyến và ngoại tuyến cho nhân viên, chi trả cho việc tham gia hội thảo... Google và nhiều doanh nghiệp cam kết dành 20% thời gian làm việc được trả lương để các nhân viên thực hiện side project. Trong một tổ chức có tài chính và chi phí hạn chế, việc học tập vẫn có thể được cổ vũ và thực hiện hiệu quả: những khoá học trực tuyến, ebook... miễn phí, hệ thống open-source, blog của những chuyên gia công nghệ... vẫn luôn có sẵn. Học tập cùng chuyên gia luôn thú vị và là cách hiệu quả, song nếu tình hình tài chính không cho phép, tổ chức vẫn có thể thực hiện việc *cùng nhau học tập* với những kênh thông tin miễn phí.



Hình 8.2: Xây dựng tổ chức học tập

Cỗ vũ và thực hành việc học tập tạo ra tri thức trong tổ chức không quá khó, nhưng chỉ là bước đầu tiên. Bước tiếp theo khó khăn hơn rất nhiều, *thay đổi hành vi* tổ chức dựa trên những tri thức được tạo ra. Bởi việc thực hành học tập nhưng không áp dụng vào thực tế là sự lãng phí và đói khi gây tác dụng ngược. Tôi đã thấy nhiều tổ chức dù cỗ vũ cho việc học tập, đọc sách, tham gia hội thảo... của nhân viên, song họ không chào đón những thay đổi từ những tri thức mới; và lâu dần, những nhân viên cảm thấy việc học tập là vô ích và họ bắt đầu dừng việc học tập. Ngại thay đổi là vấn đề chung của nhiều tổ chức, đặc biệt với những tổ chức lớn có quy trình được định hình rõ ràng; lý do duy nhất là *chi phí và sự an toàn*. Mỗi khi thay đổi, quy trình mới cần được định nghĩa lại gây tốn kém và những nhà quản lý có cảm giác bị mất kiểm soát. Và đây chính là lợi thế của những tổ chức nhỏ, khi họ có thể nhanh chóng thay đổi hành vi của mình, với chi phí nhỏ và từng bước. Agile cỗ vũ cho việc nhanh chóng thay đổi hành vi dựa trên những tri thức đã có bằng nhiều phương pháp và kỹ thuật cụ thể. Đó là Retrospective trong Scrum, nơi nhóm cùng kiểm nghiệm lại những gì đã xảy ra, cùng nhau sản sinh ra tri thức và áp dụng qua việc điều chỉnh quy trình trong Sprint tiếp theo. Đó là việc liên tục thử nghiệm, đánh giá và thay đổi tới từng phần nhỏ trong Lean.

Thật tuyệt vời Agile cung cấp phương pháp luận để thực hiện việc thay đổi hành vi tổ chức một cách an toàn và hiệu quả. Một tổ chức học tập đúng nghĩa cần cỗ vũ cho việc thay đổi hành vi qua những kiến thức mới, qua những cải tiến. Ở quy mô

nhóm, nếu nhóm phát triển nhận thấy việc thực hành kiểm thử sau khi chức năng đã hoàn thành phát sinh nhiều lỗi, nhóm nên được cổ vũ thực hành TDD; nếu nhóm phát triển nhận thấy việc triển khai tốn quá nhiều thời gian, họ nên được cổ vũ để dành thời gian cài đặt và triển khai hệ thống Continuous Deployment. Ở quy mô tổ chức, nếu doanh nghiệp nhận thấy việc trao đổi giữa các thành viên làm việc từ xa gặp nhiều khó khăn, doanh nghiệp có thể áp dụng những kênh giao tiếp nhóm mới, hoặc tổ chức việc onsite, bootcamp... để nhân viên làm việc cùng nhau. Điều quan trọng là những người lãnh đạo tổ chức phải luôn cổ vũ cho tinh thần *thử nghiệm, thất bại nhanh (fail fast)* bởi kết quả từ những thất bại chính là những bài học và kiến thức tổ chức thu nhận được. *Thất bại nhanh* với những thử nghiệm nhỏ không chỉ giúp tổ chức tránh được việc sụp đổ mà khiến tổ chức tiếp tục đi lên mạnh mẽ hơn qua những kiến thức thu được. Tổ chức không cổ vũ cho việc thử nghiệm luôn có khả năng dẫn tới *thất bại chậm (fail slow)* là nguyên do tổ chức sụp đổ hoàn toàn và không thể vực dậy do họ chưa có kinh nghiệm đối diện với thất bại.



Xây dựng tổ chức học tập luôn là một vòng lặp theo tinh thần Lean, học tập – thực hành – đo lường – đánh giá: học tập kiến thức mới, áp dụng để thay đổi hành vi tổ chức, đo lường kết quả thực hiện, đánh giá những giá trị mang lại, từ đó sản sinh ra những kiến thức mới, và tiếp tục được áp dụng để thay đổi hành vi tổ chức...

CHUYỂN ĐỔI TỔ CHỨC

Chuyển đổi tổ chức chưa bao giờ là câu chuyện dễ dàng. Chuyển đổi tổ chức truyền thống sang tổ chức Agile còn khó khăn hơn thế. Tôi thực sự nghiêm túc khi nói vậy. Bởi việc chuyển đổi một tổ chức truyền thống sang tổ chức Agile, chúng ta không chỉ đơn thuần thay đổi quy trình, cách làm... hay cấu trúc của tổ chức; đó là sự thay đổi trong hệ tư tưởng.

Những mô hình quản lý và kinh doanh truyền thống thường được xây dựng dựa trên cách tiếp cận *top-down* hoặc *bottom-up*; và gần như mọi tổ chức truyền thống đều xây dựng cách thức quản lý dựa trên một trong hai tư tưởng này. Một tổ chức với bộ phận giám đốc lãnh đạo những quản lý cấp cao và cấp trung, những người trực tiếp phân bổ công việc và giám sát việc thực hiện với những nhân viên cấp dưới. Công việc của từng nhân viên được điều hướng bởi mệnh lệnh của cấp trên hoặc yêu cầu từ cấp dưới.

Song tổ chức Agile có cách tiếp cận khác, *outside-in*. Mọi công việc được thực hiện trong tổ chức được điều hướng bởi giá trị mang lại cho khách hàng. Khách hàng chính là ông chủ, đánh giá hiệu quả mọi công việc được hiện, không phải người quản lý.

Và khi một tổ chức quyết định chuyển đổi từ mô thức quản lý truyền thống sang môi trường Agile, những vấn đề chính sau cần được cân nhắc kỹ lưỡng.

Push system hay pull system?

Đây là vấn đề cốt lõi cần được giải quyết của việc chuyển đổi sang môi trường Agile.

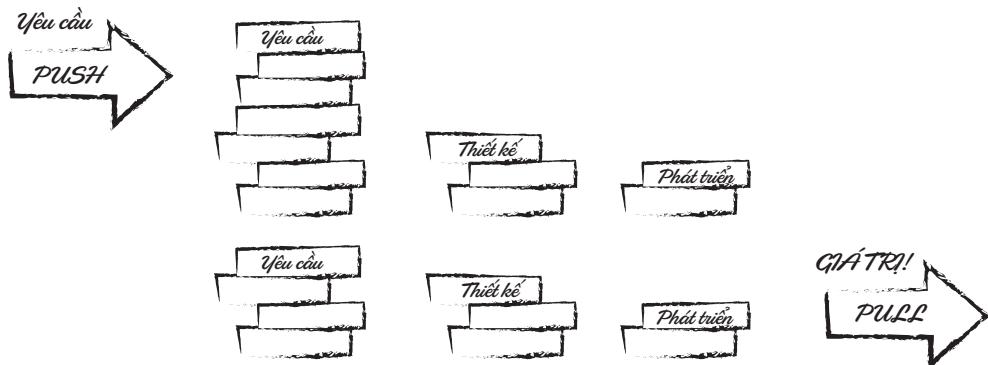
Push system (hệ thống đẩy) là mô thức quản lý phổ biến hiện nay, sử dụng một quy trình được định nghĩa trước cùng các vai trò và mô tả công việc tương ứng. Mọi công việc được chỉ định tới một thành viên cụ thể, quy định từng mức trách nhiệm và khả năng quyết định. Trong một hệ thống phân cấp, mọi công việc đều:

- cần phối hợp và báo cáo cho người quản lý trực tiếp;
- thực hiện bởi người ở mức thấp hơn trong phân cấp quản lý.

Qua việc định nghĩa rõ ràng sự phân cấp trong mô hình tổ chức, mọi công việc đều được định nghĩa cụ thể về mục tiêu, thời điểm hoàn thành, tiến độ... giúp người quản lý thực hiện việc *push* công việc xuống những nhân viên như *command* và *control* được những gì đang xảy ra; chất lượng công việc phụ thuộc lớn vào tài năng của những nhà quản lý. *Push system* dựa trên giả định rằng những nhân sự không có động lực làm việc và luôn cần được giao việc và giám sát để hoàn thành.

Cuối thế kỷ 20, Peter Drucker lần đầu nói về *thời đại của những lao động tri thức*, những công việc trở nên phức tạp hơn, và cần nhiều kỹ năng để hoàn thành hơn. Hệ quả là sự bất hợp lý trong việc phân chia và giao từng công việc cụ thể tới một cá nhân; mô hình làm việc nhóm với những thành viên có đầy đủ kỹ năng trở nên thăng thế. Và bước phát triển tiếp là nhóm tự quản.

Pull system (hệ thống kéo) là mô thức quản lý được điều hướng bởi giá trị và kết quả; những công việc được xây dựng thành một danh sách cần hoàn thành theo một thứ tự nhất định và không được giao cho một thành viên cụ thể; những thành viên trong nhóm sẽ chủ động thực hiện *pull* công việc và thực hiện. *Pull system* được xây dựng trên giả định rằng những nhân sự luôn có động lực làm việc và không cần nhiều sự quản lý.



Hình 8.3: Push system và pull system

Hiểu được sự khác biệt giữa *push system* và *pull system* là vấn đề lớn nhất với những tổ chức muốn chuyển đổi sang môi trường Agile. Điều cốt lõi nằm ở:

Cách thức định nghĩa quy trình. Trong *push system*, quy trình được định nghĩa trước, và thông thường được định nghĩa bởi những người không trực tiếp thực hiện công việc cụ thể, và đặt trọng tâm vào giá trị quản lý. Bộ phận quản lý quy trình định nghĩa ra cách thức thực hiện một chiến lược marketing, những bước thực hiện, tài liệu trao đổi giữa các phòng ban từ việc thiết lập chi phí đến thiết kế hình ảnh, nội dung... song chính họ lại có rất ít kiến thức về cách tạo ra một chiến lược marketing hiệu quả. Quy trình được định nghĩa chủ yếu hướng tới việc giúp những nhà quản lý có cái nhìn toàn cảnh và kiểm soát công việc thông qua chỉ định công việc tới từng cá nhân cụ thể; cho mọi dự án. Nhưng sự *cứng nhắc* của những quy trình được định nghĩa sẵn làm hạn chế sự sáng tạo, tạo ra rào cản làm chậm quá trình tạo ra giá trị thực sự. *Pull system* và Agile hướng tới *thực nghiệm*. Thông qua việc thực hành và đo đạc hiệu quả, nhóm tự tổ chức biết cách thực hiện nào là phù hợp cho nhóm trong từng hoàn cảnh cụ thể, và hình thành quy trình riêng cho nhóm; từ đó loại bỏ rào cản và tối ưu giá trị tạo ra.

Trách nhiệm hoàn thành công việc. Điều khiến mọi người phân vân là trách nhiệm cá nhân trong *pull system*: *nếu một công việc không được chỉ định cho một cá nhân cụ thể, điều gì khiến công việc đó được hoàn thành?* Đây là một câu hỏi hợp lý; song suy nghĩ theo một cách khác, nếu mọi công việc được chỉ định theo cá nhân cùng KPI tương ứng, những cá nhân này có thể vì tập trung nỗ lực hoàn thành một phần việc cụ thể mà bỏ qua việc cộng tác trong bức tranh toàn cảnh và tạo ra giá trị lớn hơn. Một tester được giao nhiệm vụ cụ thể sẽ cố gắng tìm ra thật nhiều bug thay vì cộng tác với developer trong nhóm nhằm tìm cách hạn chế chúng. *Pull*

system và Agile hướng trọng tâm vào xây dựng nhóm có động lực làm việc – tập trung vào con người. Nhưng đó là cách làm phù hợp với thị trường lao động tri thức ngày nay khi nhu cầu người lao động đi dần lên phía đỉnh của tháp Maslow.

Chất lượng công việc. Công việc được nâng cao khi người thực hiện hiểu được giá trị và có cái nhìn thống thể. *Push system* thường chỉ tập trung vào từng công việc cụ thể và bỏ qua góc nhìn về toàn cảnh hệ thống và giá trị, mức độ quan trọng cụ thể của một thành phần công việc tạo ra giá trị chung. *Pull system* cung cấp góc nhìn toàn cảnh, những giá trị đóng góp của từng công việc cụ thể tạo thành giá trị chung, hỗ trợ việc ra quyết định chính xác hơn. Cách vận hành kiểu thực nghiệm của *pull system* cũng giúp nhóm tìm ra quy trình cho chính mình, giúp loại bỏ trở ngại, tăng năng suất và chất lượng công việc.

Về cơ bản, *push system* và *pull system* khác nhau về hệ quy chiếu đo lường và đánh giá. *Push system* đo lường và đánh giá số lượng và khối lượng những công việc được hoàn thành; song cả số lượng và khối lượng công việc được hoàn thành không đại diện cho giá trị được tạo ra. Ngược lại, *pull system* đo lường và đánh giá dựa trên giá trị được tạo ra.

Doing Agile hay being Agile?

Đây là câu hỏi quan trọng, xác định mục tiêu, đích đến cụ thể của việc chuyển đổi sang môi trường Agile: *thực hành Agile (doing Agile) hay thực sự Agile (being Agile)?*

Thực hành Agile luôn là bước đầu tiên nhưng rất quan trọng để hình thành thói quen thông qua những thực hành, từ Daily Scrum của Scrum hay TDD của XP. Cũng thông qua việc thực hành Agile, mô thức quản lý của tổ chức bắt đầu có sự dịch chuyển từ *push system* sang *pull system*. Khi thực hành Scrum, tổ chức bắt buộc phải chuyển đổi từ phương pháp phát triển phần mềm truyền thống dưới sự điều hướng của người quản lý dự án, chỉ định từng công việc tới cá nhân cụ thể trong *push system*, sang phương pháp mới, được điều hướng bởi việc sắp xếp công việc theo độ ưu tiên về giá trị kinh doanh và được thực hiện bởi quy trình được hình thành bởi nhóm tự quản liên chức năng trong *pull system*.

Thực hành theo chỉ dẫn về bản chất là sự sao chép hành động và mô thức quản lý, giúp tăng hiệu quả kinh doanh, song việc đó dừng ở mức *cải thiện*, và thông thường, không tạo ra sự đột biến. Nhưng nếu việc thực hành thuần thực khiến thay đổi tư duy, cách tiếp cận của tổ chức, thành *thực sự agile*, những giá trị đột biến, bởi toàn bộ tổ chức sẽ được điều hướng bởi giá trị tạo ra và không bị lãng phí nguồn lực vào công việc quản lý và vận hành.



Hình 8.4: Thực hành Agile và thực sự Agile

Chuyển dịch từng bước hay bước ngoặt?

Câu hỏi cuối cùng trong việc chuyển đổi sang môi trường Agile là tổ chức lựa chọn phương pháp thực hiện chuyển đổi, *từng bước*, hay *bước ngoặt*?

Đa số tổ chức lựa chọn cách tiếp cận *từng bước*. Ban đầu việc thực hành Agile được thực hiện giới hạn trong một vài thử nghiệm tại phòng lab hay studio, trong một vài dự án với một nhóm được thành lập bởi những cá nhân *tiêu biểu*. Khi tổ chức cảm thấy tự tin với việc thực hành của nhóm, đặc biệt cảm thấy tự tin rằng nhóm đã “học đủ” qua những lần thất bại hay giải quyết xung đột, tổ chức sẽ quyết định việc thực hiện với quy mô lớn hơn. Những thành viên trong nhóm sẽ trở thành hạt nhân trong những nhóm mới, người đại diện cho việc truyền bá và lan toả tư tưởng và thực hành Agile trong tổ chức.

Nhưng cũng có cách tiếp cận khác, là thực hiện *bước ngoặt*, đồng loạt thay đổi hoàn toàn cấu trúc và cách vận hành của tổ chức. Salesforce là ví dụ tiêu biểu cho câu chuyện không tưởng như vậy, chuyển đổi sang tổ chức Agile gồm hàng ngàn nhân sự *chỉ sau một đêm*. Nói như vậy không có nghĩa rằng chỉ cần ban lãnh đạo Salesforce ra quyết định và mọi việc diễn ra một cách hoàn hảo. Salesforce đã chi hàng chục triệu đô-la cho việc chuyển đổi, bao gồm những khoá đào tạo cho nhân viên trước đó, cùng việc thay đổi cơ sở vật chất, hệ thống cho phù hợp với môi trường mới. Cũng phải nói thêm rằng, trước khi chuyển đổi, Salesforce đã mang trong mình *chất Agile*, việc chuyển đổi như một cách hệ thống hoá những gì họ đã làm. Tuy nhiên, trên thế giới cũng không thiếu những tổ chức thực hiện cách chuyển đổi *big bang* như vậy.

Phần lớn tổ chức mặc định lựa chọn việc chuyển dịch *từng bước* và cho rằng đó là giải pháp an toàn hơn, song tôi không nghĩ vậy.

Thực sự agile là câu chuyện phức tạp khi chuyển đổi, đặc biệt xung đột với những phương thức quản lý truyền thống về vai trò và công việc. Và điều rắc rối nhất đến từ những người có tiếng nói quyết định việc chuyển đổi – *những nhà quản lý*. Về cơ bản, *pull system* cung cấp một phương thức giảm tải rất nhiều khi người quản lý cấp trung không phải tập trung quá nhiều nỗ lực vào việc phân rã, giao việc, và kiểm soát quá trình thực hiện những công việc cụ thể, thường chiếm tới 80% thời gian. Và rắc rối phát sinh. Thứ nhất, những nhà quản lý có cảm giác họ mất đi công cụ kiểm soát, là quy trình và quyền hạn. Những nhà quản lý truyền thống thường theo mô hình quản lý X, với giả định rằng nhân viên không bao giờ muốn làm việc, nên họ cần quyền hạn để yêu cầu và quy trình để kiểm soát việc thực hiện. Thứ hai, khi chuyển đổi sang môi trường *thực sự agile*, những nhà quản lý này nhận vị trí và công việc gì?

Những nhà tư vấn việc chuyển đổi sang tổ chức Agile thường đề cập đến những điều tuyệt vời dành cho những nhà quản lý, rằng họ sẽ *sung sướng* vì không phải thực hiện công việc quản lý vặt khi nhóm tự tổ chức và nhận trách nhiệm. Tôi cho rằng cách tiếp cận này ít mang lại hiệu quả. Bạn không thể bảo những nhà quản lý gen X cảm thấy thoải mái và sung sướng vì không được làm công việc kiểm soát. Nếu chúng ta giả định rằng những lập trình viên cảm thấy thoải mái và sung sướng khi tự họ quản lý công việc, kiểm soát quá trình theo phương pháp luận Agile, thì cũng nên tôn trọng rằng những nhà quản lý gen X coi việc quản lý và kiểm soát giống như hơi thở của họ vậy.

Thực hiện việc *chuyển dịch* là thay đổi nhà quản lý gen X thành nhà quản lý gen Y, những người giả định rằng nhân viên luôn muốn làm việc và thực hiện tốt công việc của mình, và họ không lạm dụng *cây gậy và củ cà rốt*. Một tổ chức với đa số nhà quản lý gen Y hoặc những nhà quản lý gen X khó thay đổi, theo tôi, là hai tổ chức phù hợp để thực hiện *big bang*, chuyển đổi *bước ngoặt* sang tổ chức Agile.

Dù sao thì *chuyển dịch từng bước* vẫn gần với tư tưởng Agile hơn, nhưng chuyển đổi sang tổ chức Agile là câu chuyện theo đặc trưng riêng của từng tổ chức và không có mẫu chung. Chỉ chính tổ chức mới có câu trả lời phù hợp.



Tôi đề cập nhiều tới những nhà quản lý không đồng nghĩa với việc phủ nhận tầm quan trọng của nhân viên trong việc chuyển đổi. Đây là thành tố rất quan trọng. Song với đa số tổ chức, đây là thành phần năng động và dễ chuyển đổi; cái họ thiếu là việc đào tạo, hướng dẫn, định hướng... đúng. Bài toán này có thể giải quyết bằng những nguồn lực trong tổ chức, thuê ngoài hoặc tư vấn...

Những nhà quản lý cấp cao, đa số ủng hộ việc chuyển đổi, bởi họ là người nhìn thấy rõ nhất chi phí quản lý và cái giá của việc nâng cao giá trị kinh doanh. Song mọi quyết định của họ, cần sự hỗ trợ *thật tâm từ tư tưởng* của những nhà quản lý cấp trung.

HIẾU ĐÚNG

Being Agile là mục tiêu.

Mục tiêu tối thượng của mọi tổ chức và doanh nghiệp chưa bao giờ thay đổi. Đó là giá trị được tạo ra cho xã hội, cho tổ chức, cho các ông chủ... Mọi phương pháp xây dựng, quản lý trong tổ chức đều là công cụ nhằm tối đa hoá giá trị được tạo ra tức thì hoặc lâu dài. Xây dựng môi trường Agile trong tổ chức cũng vậy, *being Agile* không phải là mục tiêu, đó là công cụ giúp tổ chức tối đa hoá giá trị được tạo ra dựa trên việc phản hồi linh hoạt với những thay đổi trên nhiều yếu tố thị trường, khách hàng, nhân sự... Do đó, việc xây dựng hay chuyển đổi tổ chức không nên đặt việc *doing Agile* hay *being Agile* là mục tiêu của quá trình xây dựng hoặc chuyển đổi, thực chất đó là *lựa chọn công cụ phù hợp* với mục tiêu của tổ chức. Song sự linh hoạt của tổ chức sẽ được tối đa hoá khi tổ chức đó *thực sự Agile – being Agile*.

Agile không phù hợp với những doanh nghiệp outsourcing.

Đây là một trong những thắc mắc và trở ngại khi những doanh nghiệp Việt Nam tiếp cận với Agile; đa phần đều mặc nhiên nhận định Agile không phù hợp với những dự án outsourcing khi giá cả, quy mô hoặc thời gian cố định thường là mẫu chung của những dự án outsourcing. Và họ thẳng tay loại Agile ra khỏi danh sách phương pháp luận. Ở đây, doanh nghiệp cần phân định rạch ròi giữa hai phần. Thứ nhất, là những dự án nhận được từ khách hàng. Ngày càng nhiều khách hàng nhận ra rằng, mô hình outsourcing truyền thống khiến sản phẩm của họ không đi tới đâu cả, và họ áp dụng Agile cho đối tác của mình; không thực hành Agile chính là bất lợi của doanh nghiệp trong việc tiếp cận những khách hàng tốt. Thứ hai, ngay cả khi không thực hiện dự án theo Agile, phần còn lại của tổ chức cũng có thể xây dựng theo môi trường thực sự Agile; sự linh hoạt trong việc tiếp cận khách hàng, nhân sự, và cung cấp dịch vụ outsourcing giá trị cao tới khách hàng không phụ thuộc tất cả vào việc dự án có thực hành Agile hay không.

TỔNG KẾT

Xây dựng tổ chức Agile không thực sự là điều khó khăn, nhưng thực hiện sớm nhất có thể sẽ giúp tổ chức tiết kiệm công sức và chi phí chuyển đổi. Cấu trúc tổ chức xoay quanh khách hàng và sản phẩm hơn là quy tắc quản lý và kiểm soát là điều quan trọng nhất. Khi đã lựa chọn Agile, hãy thực hiện đúng ngay từ ban đầu, đánh giá trước khi đưa ra những thay đổi; khả năng thực hiện đúng Agile chính là lợi thế lớn nhất của những tổ chức có quy mô nhỏ và vừa. Hãy cổ vũ cho việc thay đổi hành vi của tổ chức thông qua học tập; thử nghiệm sớm là việc quan trọng để duy trì khả năng bền vững của tổ chức.

Chuyển đổi tổ chức sang môi trường Agile là thử thách lớn hơn rất nhiều, và không có công thức thành công chung. *Thực sự Agile* không phải là mục tiêu của tổ chức, đó là công cụ để sản sinh giá trị cho khách hàng, cho tổ chức; song *thực sự Agile* thường là công cụ cho ra giá trị bước ngoặt so với *thực hành Agile*. Các tổ chức thường tiếp cận việc chuyển đổi thông qua chuyển dịch từng bước; song chuyển dịch bước ngoặt cũng là một lựa chọn khả thi.

CÁ NHÂN LĨNH HOẠT

Cảm ơn bạn đã đọc đến phần này của cuốn sách!

Mặc dù Agile thực sự đã chứng minh được hiệu quả về mặt phương pháp luận cũng như thực hành trong môi trường làm việc nhóm và các tổ chức công nghệ (và dần mở rộng tầm ảnh hưởng sang những lĩnh vực khác), mỗi cá nhân, thật tiếc, thường không phải là người quyết định việc Agile được áp dụng (và áp dụng thành công) trong tổ chức của mình. Agile, cũng như bất cứ phương pháp vận hành tổ chức nào khác, đều được quyết định bởi người lãnh đạo – và số lượng này, thường rất nhỏ.

Những phần trước trong cuốn sách giúp mang đến những góc nhìn toàn cảnh về việc áp dụng Agile trong tổ chức nhằm cổ vũ cho phong trào Agile tại Việt Nam. Nhưng khi tổ chức còn chưa thực sự chuyển mình sang Agile, thật may mắn là mỗi cá nhân vẫn có thể sử dụng Agile cho công việc của riêng mình, trong từng hoạt động của cuộc sống cá nhân.

Agile chưa bao giờ được đề cập tới như một phương pháp luận dành cho cá nhân; song trong suốt những năm thực hành Agile trong công việc, tôi nhận ra những tư tưởng Agile thực sự mang lại hiệu quả trong cuộc sống và phù hợp với cá nhân trong thời đại ngày nay – những người lao động theo định hướng *giá trị* thay vì *khối lượng* như những cách thức truyền thống. Và dưới góc độ của lập trình viên, việc cá nhân áp dụng thành công những tư tưởng Agile trong cuộc sống cũng sẽ mang lại hiệu quả tuyệt vời khi họ làm việc trong nhóm thực hành Agile; bởi khi đó, mọi triết lý với họ thuần thực như hơi thở.

QL
CN

QUẢN LÝ
CÁ NHÂN

Thực ra những phương pháp trợ giúp phát triển kỹ năng cá nhân, tập trung vào quản lý công việc, đã được phát triển từ rất sớm, và hàng năm vẫn có hàng chục đầu sách được xuất bản với số lượng lớn. Và thật may mắn, ngày càng nhiều những cuốn sách với tính thực hành cao, phần nhiều trong đó nói về...

QUẢN LÝ THỜI GIAN...

Không có gì đáng tranh luận khi *quản lý thời gian* luôn là kỹ năng hàng đầu được nhắc đến trong những phương pháp quản lý nhằm tăng hiệu suất công việc cá nhân. Bởi thời gian là bất biến; bởi thời gian là vàng bạc. Dù bạn là một người lao động bình thường hay ông chủ của một tập đoàn lớn; dù bạn là một công dân trong thế giới của 1 hay 7 tỷ người; bạn vẫn chỉ có 24 giờ trong một ngày. Chiếm lĩnh thời gian tốt hơn giúp bạn dễ thành công hơn.

Nguyên lý đứng sau việc quản lý thời gian là: *Với cố định 24 giờ trong một ngày, sắp đặt công việc vào thời điểm hợp lý nhằm tối đa hoá số lượng công việc được thực hiện và giảm thiểu thời gian “chết” trong tổng thời gian hữu hạn là tiền đề của thành công.*

Ý tưởng chính của những phương pháp quản lý thời gian truyền thống là thiết lập một danh sách công việc cần thực hiện trong ngày, đặt chúng vào những khoảng thời gian phù hợp, và cố gắng tối giản những khoảng thời gian “chết” – những khoảng thời gian không có công việc.

<i>Thứ Hai</i>	<i>08h30</i>	<i>Thức dậy</i>
	<i>09h00</i>	<i>Ăn sáng</i>
	<i>09h30</i>	<i>Tối văn phòng</i>
	<i>10h00</i>	<i>Hợp giao ban</i>
	<i>11h00</i>	<i>Tổng hợp báo cáo</i>
	<i>12h00</i>	<i>Ăn trưa</i>
	<i>14h00</i>	<i>Hợp với đối tác</i>
	<i>15h00</i>	<i>Trình bày kế hoạch tuần</i>
	<i>16h30</i>	<i>Phỏng vấn</i>
	<i>18h00</i>	<i>Đi ăn với đối tác</i>
	<i>22h00</i>	<i>Đọc sách</i>

Hình 9.1: Quản lý thời gian

Ngày nay, với sự trợ giúp của công nghệ, đặc biệt là smartphone mang đến cho chúng ta sự tiện lợi đặc biệt trong việc quản lý thời gian *trong lòng bàn tay*. Nếu

Calendar không phải là một ứng dụng được bạn sử dụng thường xuyên trên smartphone, bạn chắc chắn không được coi là một *công dân kiểu mẫu*.

... LÀ CHƯA ĐỦ

Nhưng nếu bạn chỉ thường xuyên sử dụng Calendar trên smartphone, bạn vẫn chỉ được coi là *công dân kiểu mẫu*, không hơn; bởi quản lý thời gian chỉ là một trong những kỹ năng tối thiểu nằm trong tập kỹ năng quản lý công việc hiện nay.

Điểm mấu chốt của vấn đề, *dù thời gian của chúng ta là hữu hạn, số lượng và khối lượng công việc chúng ta cần làm không ngừng tăng lên*. Và hậu quả tất yếu xảy ra là, bất kể kỹ năng quản lý thời gian của bạn tốt đến đâu, vẫn còn hàng loạt những công việc khác đang sẵn sàng chờ bạn sắp lịch và thực hiện. Cũng giống như trong lĩnh vực phát triển phần mềm, làm tốt việc quản lý thời gian thường là chưa đủ; khi chúng ta đã phát triển kỹ năng quản lý thời gian đến một mức độ nhất định, hiệu quả mang lại thường không cao với những cải tiến cục bộ. Vậy nên, chúng ta cần tiếp cận vấn đề theo một cách nhìn khác.

Có lẽ tôi cần nhấn mạnh rằng, với bất kỳ cách tiếp cận quản lý công việc nào, thì chỉ có *thời gian là yếu tố duy nhất cố định*. Thông tin chúng ta nhận được ngày càng nhiều lên, lượng việc chúng ta cần thực hiện mỗi ngày nhanh chóng thay đổi, nguồn lực của chúng ta cũng nhanh chóng thay đổi theo; *chỉ có thời gian là không đổi*. Tối đa hóa thời gian bằng cách sắp xếp và bám theo một lịch làm việc dày đặc bỗng chốc trở nên lỗi thời. Điều gì xảy ra nếu khách hàng của bạn đột ngột viếng thăm vào 2 giờ chiều? Bạn sẽ từ chối họ vì đó là khoảng thời gian bạn đã lập kế hoạch để lập trình hay sẽ chấp nhận phá vỡ kế hoạch? Điều gì xảy ra nếu bạn bỗng cảm thấy chán nản và mệt mỏi vào 2 giờ chiều khi còn một loạt những công việc đã được lên lịch? Quản lý thời gian theo phương pháp truyền thống bỗng chốc trở nên điên rồ. Bởi vậy, chúng ta cần sự linh hoạt hơn trong cách quản lý công việc thay vì bám sát một kế hoạch hay lịch trình sẵn có theo phương pháp quản lý thời gian truyền thống.

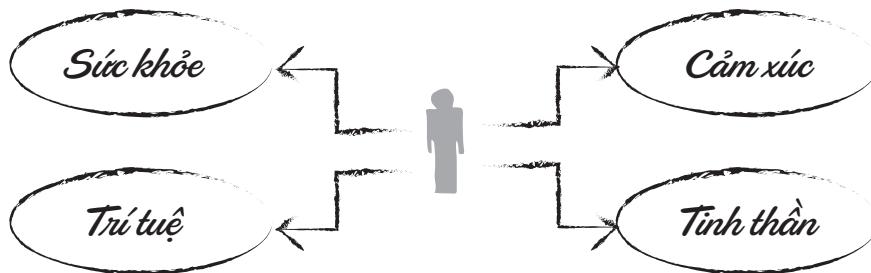
Phương pháp quản lý thời gian truyền thống, từ *tối đa lượng công việc thực hiện trong thời gian cố định*, cần được chuyển thành *tối đa giá trị công việc thực hiện trong khoảng thời gian và nguồn lực hữu hạn*.

Và giờ đây chúng ta cần nhiều hơn là chỉ kỹ năng quản lý thời gian.

QUẢN LÝ NĂNG LƯỢNG

Thời gian không biến đổi, nhưng năng lượng của con người lại không phải như vậy.

Đã bao giờ bạn nhận ra mình có một ngày xử lý hàng trăm công việc thành công một cách suôn sẻ, và một ngày khác lại không thực hiện được công việc nào? Mỗi ngày chúng ta đều có 8 giờ làm việc, một giờ buổi sáng hay một giờ buổi chiều đều là 60 phút và giống nhau; nhưng tại sao lượng công việc được thực hiện tại 2 giờ vào 2 thời điểm khác nhau lại khác nhau đến vậy? Đáng tiếc, số lượng và đặc biệt là chất lượng các công việc lại không phải là một hằng số phụ thuộc tuyến tính vào thời gian; chúng phụ thuộc nhiều hơn vào *năng lượng* của bản thân. *Năng lượng* của mỗi người được hiểu là khả năng thực hiện công việc của họ tại một thời điểm bất kỳ, là tổng hợp của các yếu tố: sức khoẻ, cảm xúc, trí tuệ và tinh thần. Những yếu tố này được Tony Schwartz đưa ra trong một bài viết khá nổi tiếng trên Harvard Business Review *Quản lý năng lượng của bạn, không phải thời gian*.



Hình 9.2: Các thành tố của năng lượng

Giờ đây, kỹ năng không chỉ dừng lại ở việc sắp đặt những công việc vào từng thời điểm cụ thể thông qua việc quản lý thời gian; chúng ta còn cần quản lý và duy trì nguồn năng lượng bản thân ở mức cao nhất, khiến công việc hiệu quả hơn. Bao gồm:

Thể trạng: *Năng lượng vật lý*. Dù chúng ta có lao động theo bất cứ hình thức nào, chân tay hay trí óc, thể trạng là điều quan trọng nhất. Chúng ta không thể làm gì nếu gặp vấn đề về sức khoẻ, ngay cả khi đó chỉ là một cuộc gọi điện thoại để trao đổi với đồng nghiệp. Việc giữ cho thể trạng, hay nói cách khác là sức khoẻ, được duy trì ở trạng thái tốt là điều đặc biệt quan trọng. Làm việc liên tục trong một khoảng thời gian, dù bằng hình thức nào, cũng đốt cháy hàng ngàn calo. Và không gì tốt hơn bằng những khoảng nghỉ, để hồi phục các cơ bắp và duy trì nguồn năng lượng vật lý. Nhưng dù thế nào, nguồn năng lượng này cũng sẽ cạn kiệt vào cuối ngày; và việc tái tạo nguồn năng lượng này cũng quan trọng không kém. Chế độ ăn uống và chất lượng giấc ngủ giờ đây được nhắc đến nhiều hơn bao giờ hết.

Cảm xúc: *Chất lượng của năng lượng*. Thể trạng, thật ra mới đáp ứng được phần

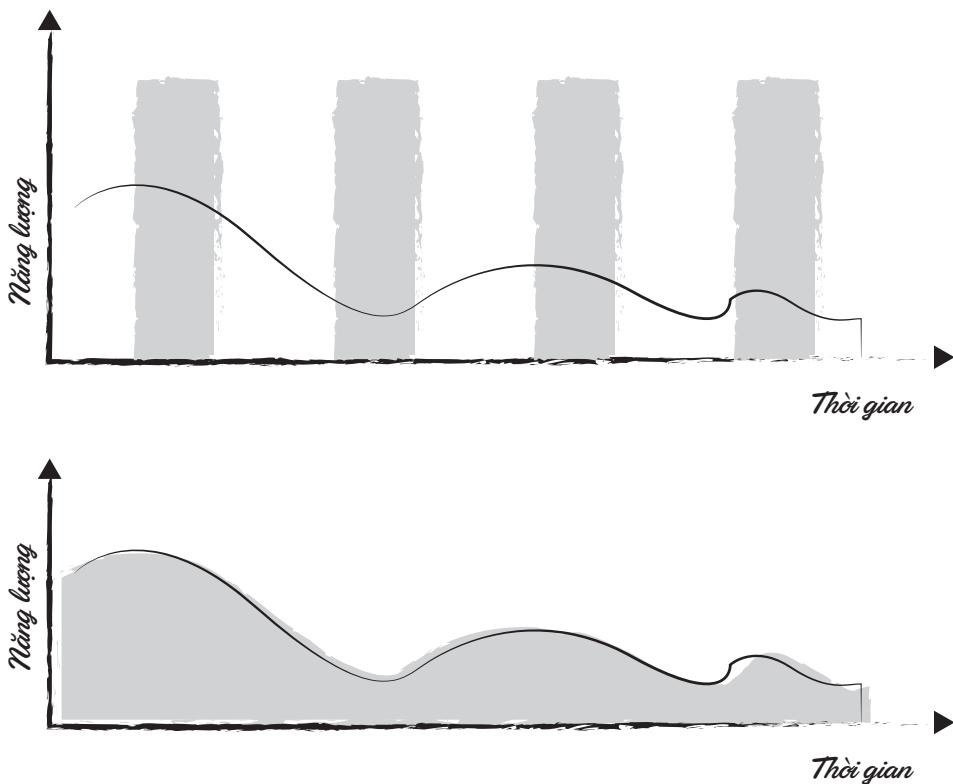
số lượng của năng lượng, và thực tế, cũng giống như thời gian, chất lượng công việc được thực hiện lại không phải là sự phụ thuộc tuyến tính theo số lượng năng lượng; chúng bị chi phối nhiều hơn bởi chất lượng của năng lượng – cảm xúc. Rất nhiều những nghiên cứu đều cho thấy rằng, con người làm việc hiệu quả hơn khi họ sở hữu những cảm xúc tích cực như vui vẻ, háo hức; một số ít đạt hiệu quả làm việc cao khi sở hữu cảm xúc tiêu cực như chịu sức ép, tức giận (chàng khổng lồ xanh Hulk là một ví dụ). Việc định hướng bản thân tới cảm xúc phù hợp nhằm tối đa hoá chất lượng của những năng lượng hiện có cũng quan trọng không kém việc sản sinh ra những năng lượng này qua một thể trạng tốt. Hàng chục đầu sách trợ giúp tăng hiệu suất công việc ra đời gần đây đều cho thấy rằng, việc loại bỏ áp lực và nâng cao cảm xúc, chỉ số EQ quan trọng hơn nhiều làm việc chăm chỉ; chính là phương pháp chúng ta tạo ra nhiều năng lượng chất lượng.

Trí tuệ: Tập trung năng lượng. Số lượng lớn năng lượng với chất lượng cao chỉ là tiền đề cho những công việc được hoàn thành; bởi dù sao, những năng lượng nội tại cần được chuyển hoá một cách hợp lý thành những công việc cụ thể. Và không gì tốt hơn sự tập trung. Khoa học đã chứng minh, chỉ một số ít những bộ não siêu việt có thể thực hiện được những công việc song song; hầu hết chúng ta chỉ có thể thực hiện tốt một công việc tại một thời điểm cụ thể. Thực hiện những công việc song song đồng nghĩa với việc chúng ta phải luôn dành thời gian chuyển đổi giữa những công việc; và thời gian này, lớn một cách đáng kinh ngạc, có thể lên tới 40% tổng thời gian. Cách khôn ngoan nhất, là dồn toàn bộ năng lượng để tập trung xử lý duy nhất một công việc, hoàn thành nó trước khi xử lý công việc tiếp theo. Phương pháp Podomoro dù không mới nhưng rất hiệu quả trong việc quản lý sự tập trung cũng như nâng cao chất lượng và số lượng năng lượng thông qua việc tái tạo chúng qua từng khoảng thời gian làm việc và nghỉ ngơi hợp lý.

Tinh thần: Ý nghĩa của năng lượng. Mọi việc được hoàn thành tốt hơn khi chúng ta muốn chúng được hoàn thành. Một phần của năng lượng ẩn rất sâu trong mỗi con người nhưng là gốc rễ của mọi vấn đề, là tinh thần. Bạn yêu thích công việc đó? Bạn muốn hoàn thành công việc đó? Nếu mọi câu trả lời là có, xin chúc mừng, bạn sẽ biết cách sử dụng nguồn năng lượng dồi dào với chất lượng cao mà mình có. Đấy là lý do những người thành công, những người luôn làm việc với hiệu suất cao, luôn nói rằng hãy làm công việc bạn yêu thích.

Tôi không muốn đi sâu hơn về chủ đề này, vì đây là một chủ đề đặc biệt và cần nhiều hơn một cuốn sách để lý giải tại sao những yếu tố này lại tồn tại song song, cấu thành nên năng lượng (và năng lực) làm việc của mỗi cá nhân. Điều tôi muốn nhấn mạnh ở đây là, chúng ta cần nhận ra rằng, dù trong bất cứ hoàn cảnh nào, con người đều không phải là một chiếc máy và để duy trì hiệu suất làm việc cao,

những yếu tố này luôn cần được duy trì cân bằng một cách khoa học. Đây chính là vấn đề của phương pháp quản lý thời gian truyền thống, khi giả định rằng *con người như một cỗ máy, duy trì nguồn năng lượng ổn định, tuyến tính theo thời gian*. Bạn có thể để ý tới *diện tích hiệu quả* dưới đây.



Hình 9.3: Diện tích hiệu quả

Khi chúng ta cùng lúc giải quyết hàng loạt công việc đồng thời, nguồn năng lượng sẽ nhanh chóng đi xuống. Khi chúng ta tập trung giải quyết từng công việc quan trọng, nguồn năng lượng được duy trì trong thời gian dài, giúp *diện tích hiệu quả* tăng lên đáng kể.

Và những phương pháp tiếp theo trong những chương sau, là hệ quả cách áp dụng tư tưởng Agile để quản lý những thành tố này, nhằm mang lại hiệu quả ở mức cao nhất.

HIỂU ĐÚNG

Quản lý thời gian không cần thiết trong thời đại này.

Không, quản lý thời gian vẫn rất cần thiết. Những gì tôi phủ nhận ở trên là *phương pháp quản lý thời gian truyền thống* bằng việc *tối đa hoá số lượng công việc được thực hiện*. Phương pháp quản lý thời gian vẫn rất quan trọng, nhưng là chưa đủ; chúng ta cần thêm phương pháp *quản lý năng lượng* trong tập kỹ năng.

TỔNG KẾT

Thời gian là yếu tố cố định duy nhất trong thế giới ngày nay. Bất kể chúng ta có kỹ năng quản lý thời gian tốt đến đâu, vẫn là chưa đủ. Chúng ta cần nhìn nhận về *giá trị công việc* nhiều hơn là số lượng công việc được làm trong khoảng thời gian hữu hạn. *Quản lý năng lượng* là cách tốt nhất để thực hiện những công việc tập trung vào giá trị.

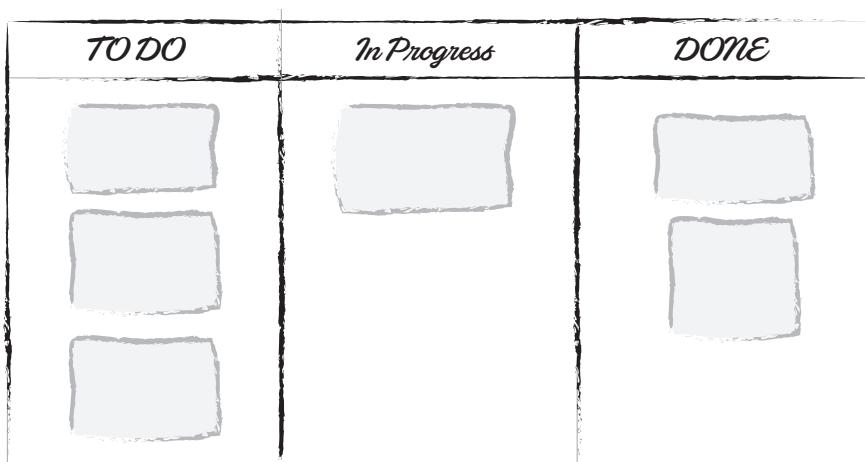
10

TRỢ GIÚP CỦA
KANBAN

Những năm cuối của thế kỷ 20 và thập niên đầu thế kỷ 21 là sự bùng nổ những công cụ trợ giúp cá nhân trong quản lý công việc dựa trên tiến trình thời gian. Có tới hàng chục công cụ cho phép mỗi cá nhân gói gọn lịch làm việc của mình chỉ trong một chiếc điện thoại và nhanh chóng lên lịch chỉ với một cú chạm tay. Nhưng rồi người ta nhanh chóng nhận thấy việc chỉ bám sát tiến trình thời gian là không đủ bởi trong một thế giới quá biến động, hiệu quả công việc nhiều khi không phải là phép tịnh tiến theo thời gian. Khi con người sử dụng tràn ngập những công cụ hỗ trợ, chúng ta có khả năng thực hiện rất nhiều công việc cùng lúc, đó cũng là khi xuất hiện những cạm bẫy hiệu quả sụt giảm nghiêm trọng. Và rồi chúng ta lại có nhu cầu quay về với những phương pháp căn bản nhất đã bị lãng quên: *nâng cao hiệu quả thông qua bám sát tiến trình trạng thái công việc thay vì chỉ bám sát tiến trình thời gian.*

Và không một công cụ nào tốt hơn Kanban trong việc này.

Ý tưởng *Kanban cá nhân (Personal Kanban)* hoàn toàn tương tự như sử dụng Kanban trong mỗi tổ chức bởi sự đơn giản và hiệu quả. Mỗi cá nhân sử dụng một bảng Kanban được chia thành các cột tương ứng với mỗi trạng thái của công việc. Mỗi công việc được ghi trên một tấm thẻ, xuất phát từ cột *TO-DO (cần làm)* và kết thúc ở cột *DONE (hoàn thành)*.



Hình 10.1: Bảng Kanban cá nhân

Thật đơn giản đúng không? Nhưng chính sự đơn giản này mang lại hiệu quả không ngờ nếu chúng ta thực hành tốt phương pháp Kanban cá nhân với hiểu biết sâu sắc về những nguyên lý đứng sau.

NHỮNG NGUYÊN LÝ CỦA KANBAN CÁ NHÂN

Kanban cá nhân bao gồm một tập những nguyên lý cơ bản nhằm nâng cao hiệu quả quản lý công việc cá nhân.

Trực quan hóa là nguyên lý nổi bật nhất. Chúng ta chỉ mất khoảng 15 giây để nhận biết được tình hình cá nhân vào bất kỳ thời điểm nào trong ngày. Khi chúng ta quay lại sau bữa trưa và nhận thấy chỉ có 2 công việc được thực hiện cùng 10 công việc đang tồn đọng; chúng ta biết mình cần tăng tốc hoặc mua thêm một chút đồ ăn dự phòng cho chiều tối. Ngược lại, khi chỉ còn 1 công việc tại cột *TO-DO*, hãy nghĩ đến việc chọn một nơi thú vị để kết thúc ngày làm việc tuyệt vời; hoặc sẵn sàng để thêm những công việc mới nếu chúng ta vẫn đang tràn đầy năng lượng. Trực quan hóa còn giúp chúng ta phản ứng kịp thời với những thay đổi bất chợt. Chúng ta có thể đã hình dung ra một ngày nhàn hạ chỉ với một cuộc họp vào buổi sáng khi bước ra khỏi nhà; song kết quả của buổi họp là một loạt những công việc cần được thực hiện ngay trong ngày; bảng Kanban cá nhân lúc này sẽ phản ánh chính xác những gì chúng ta phải thực hiện với một loạt những công việc được đổ đầy cột *TO-DO*.

Tập trung là nguyên lý quan trọng nhất. Hãy bắt đầu với từng công việc trong cột *TO-DO*, cố gắng hoàn thành để đưa công việc tới trạng thái cuối cùng, *DONE*, trước khi bắt đầu một công việc khác. Nguyên lý này gọi là *WIP (Work In Progress - giới hạn công việc đang thực hiện)* để tránh việc xao nhãng và chuyển đổi giữa các công việc đang được thực hiện nhằm đạt hiệu suất cao nhất.

Ưu tiên tạo ra hiệu quả. Luồng công việc tạo ra kiểm soát. Trạng thái công việc tạo ra động lực. Những công việc trong danh sách *TO-DO* được sắp xếp theo độ ưu tiên tạo ra hiệu quả. Từng công việc trải qua một luồng trạng thái từ *TO-DO* tới *DONE* cho chúng ta kiểm soát hoàn toàn tiến trình công việc. Bằng việc đẩy dần từng công việc về *DONE*, chúng ta càng có động lực mạnh mẽ để dọn dẹp hết danh sách *TO-DO*, giống như một người sắp về đích vậy. Và ngay cả khi chúng ta cảm thấy cạn kiệt năng lượng sau khi thực hiện 5/10 công việc; hãy yên tâm, đó là 5 công việc mang lại hiệu quả cao nhất, đôi khi chính là 80% hiệu quả của cả 10 công việc, và chúng ta có thể yên tâm nghỉ ngơi.

ÁP DỤNG NHỮNG NGUYÊN LÝ

Hãy bắt đầu nào, chúng ta cần những gì?

Trực quan hóa. Hãy tạo cho mình một bảng Kanban cá nhân với những trạng thái

thích hợp. Hãy chắc chắn rằng, bảng Kanban của chúng ta có 2 cột cố định, *TO-DO* là cột đầu tiên và *DONE* là cột cuối cùng. Xem giữa *TO-DO* và *DONE* là một hoặc một loạt những cột khác thể hiện trạng thái công việc; được sắp xếp từ trái sang phải thể hiện mức độ tiến triển của công việc. Một số người chỉ sử dụng một cột duy nhất là *In Progress* (đang thực hiện), số khác thích có nhiều cột thể hiện mức độ chi tiết hơn như: *In Conversation* (đang thảo luận), *Decision Made* (đã quyết định), *In Progress* (đang thực hiện) như:

<i>TODO</i>	<i>In Conversation</i>	<i>Decision Made</i>	<i>In Progress</i>	<i>DONE</i>

Hình 10.2: Bảng Kanban cá nhân chi tiết

Đơn giản hay chi tiết, ít cột hay nhiều cột? Thông thường, những người mới thực hành Kanban cá nhân thường hay định nghĩa nhiều cột thể hiện những trạng thái chi tiết hơn; và sử dụng ít cột hơn khi đã thực hành thuần thục. Việc sử dụng nhiều cột khi mới thực hành Kanban cá nhân cũng rất hữu ích vì giúp chúng ta tập trung sự chú ý vào bảng Kanban nhiều hơn do thường xuyên cập nhật trạng thái công việc. Tuy nhiên, chúng ta chỉ nên sử dụng tối đa là 5 cột (bao gồm 2 cột *TO-DO* và *DONE* cố định); sử dụng quá nhiều cột khiến chúng ta không nhanh chóng chụp được trạng thái công việc trong 15 giây và có thể gây ra sự xao nhãng. Đây không phải là điều Kanban cá nhân khuyến khích.

Ngày nay có rất nhiều công cụ cho phép bạn tạo ra bảng Kanban và sử dụng qua điện thoại thông minh hoặc trên máy tính như Trello, Lino, Kanbanize,... Tuy vậy, tôi khuyến khích bạn sử dụng bảng Kanban vật lý trong giai đoạn đầu thực hành. Tất nhiên, việc này không tiện dụng nhưng bạn hoàn toàn có thể linh hoạt hơn bằng việc sử dụng một phần bức tường để tạo ra bảng Kanban tại nơi làm việc và tại nhà. Mục đích cũng là để chúng ta không bị phân tán sự tập trung hoặc lảng

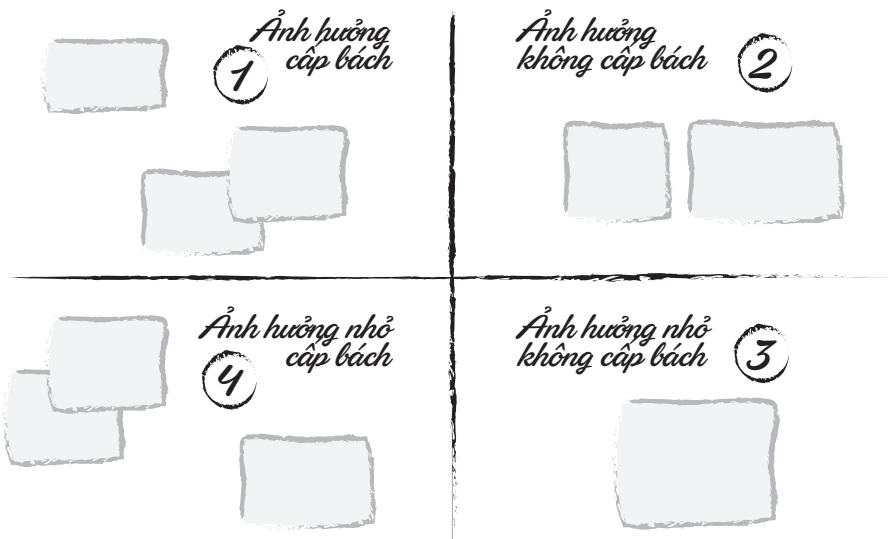
quên bảng điện tử. Một người thường di chuyển và làm việc nhiều nơi trong ngày như tôi không thích hợp với bảng Kanban vật lý; trong giai đoạn đầu sử dụng Trello, tôi dành ra 1 (trong 2) màn hình tôi có ở nơi làm việc chỉ để hiện thị Trello. Bạn có thể áp dụng phương pháp đó, bởi trong giai đoạn mới thực hành Kanban cá nhân, sử dụng bảng điện tử trên điện thoại thông minh thường mang lại hiệu quả rất tệ bởi 2 lý do: *chúng ta không cập nhật tình hình công việc hiện tại vào điện thoại và đánh mất hoàn toàn tính trực quan hóa*; và, *chúng ta rất dễ xao nhãng mỗi khi sử dụng điện thoại thông minh để cập nhật công việc bởi một vài hành động “tranh thủ” vào Facebook, đọc tin...* Theo tôi, bạn chỉ nên sử dụng bảng Kanban điện tử nếu bạn là người hay di chuyển và hãy chắc chắn rằng mình biết cách sử dụng đúng đắn.

Phân loại và xác định độ ưu tiên. Để Kanban cá nhân hoạt động hiệu quả, hãy đảm bảo rằng, bất kỳ công việc nào được đưa vào danh sách TO-DO cũng thỏa mãn:

- có định nghĩa công việc và mức độ hoàn thành (*DoD: Definition of Done*) thật rõ ràng;
- được đặt đúng vị trí sau khi được phân loại và xác định độ ưu tiên.

Thường thì chúng ta bị cuốn vào vòng xoáy ngập trong công việc vì không phân loại và xác định được mức độ ưu tiên giữa các công việc. *“Trời ơi, nhiều việc quá, làm gì bây giờ?”* là một lời than phổ biến; và thật đáng buồn, để trả lời cho câu hỏi *làm gì bây giờ* chúng ta chọn làm những việc “dễ” trước chỉ vì đấy là câu trả lời dễ dàng và nhanh chóng. Và chúng ta đi dọn dẹp nhà cửa, trong lúc đó nghĩ về bản báo cáo một cách hời hợt và trình bày nó bằng PowerPoint khi đã quá mệt vào cuối ngày. Đấy chính xác là những gì nhiều người đang làm, thật tệ hại.

Trong hàng trăm công việc chúng ta có hàng ngày, tất cả đều có thể được phân loại và xác định độ ưu tiên dựa trên *mức độ ảnh hưởng* và *mức độ cấp bách*. Mức độ ảnh hưởng của công việc chính là hệ quả của việc thực hiện công việc đó; nói cách khác, là ưu tiên về sự quan trọng của công việc. Mức độ cấp bách của công việc là sự cần thiết phải thực hiện công việc trong tương lai gần hoặc xa; nói cách khác, là ưu tiên về thời gian thực hiện công việc. Hãy cùng viết tất cả công việc bạn đang có, mỗi việc trên 1 mảnh giấy và sắp xếp chúng vào các góc phần tư như sau:



Hình 10.3: Sắp xếp công việc theo những góc phần tư

Góc phần tư thứ nhất: *ảnh hưởng lớn và cấp bách*. Đây là những công việc quan trọng và cần được thực hiện ngay như gặp gỡ khách hàng tiềm năng nhất, hoàn thành bản báo cáo cho buổi họp ngày mai...

Góc phần tư thứ hai: *ảnh hưởng lớn nhưng không cấp bách*. Đây là những công việc quan trọng, có ảnh hưởng lâu dài nhưng có thể trì hoãn việc thực hiện như đọc sách, phát triển kỹ năng thuyết trình...

Góc phần tư thứ ba: *ảnh hưởng nhỏ và không cấp bách*. Mỗi ngày chúng ta đều tiêu tốn một phần thời gian lớn cho những công việc này, đáng tiếc phần nhiều là do thói quen không khoa học như đọc báo, lướt Facebook, lan man trên những blog tâm sự...

Góc phần tư thứ tư: *ảnh hưởng nhỏ và cấp bách*. Đây là góc phần tư hài hước nhất bởi đa phần chúng ta bị nhầm lẫn và đánh đồng giữa việc cấp bách và quan trọng như nghe điện thoại, trả lời email, xác định tối nay đi đâu chơi vào buổi sáng... Nói chung thì chẳng có mấy việc có ảnh hưởng nhỏ mà lại cấp bách cả, trừ khi chúng ta có một con chuột chết trong nhà cần ném đi ngay.

Những công việc nằm trong góc phần tư thứ nhất, hãy di chuyển chúng lên đầu danh sách *TO-DO*. Những công việc nằm trong góc phần tư thứ ba, hãy di chuyển

chúng xuống cuối danh sách TO-DO. Xen giữa hai tập công việc này là những công việc thuộc góc phần tư thứ tư và góc phần tư thứ hai; nhưng bạn cần chắc chắn rằng mình xác định đúng những công việc thuộc góc phần tư thứ tư bởi như tôi đã nói, có rất ít công việc có ảnh hưởng nhỏ mà lại cấp bách, đa phần chúng nên nằm ở góc phần tư thứ ba. Đó chính là danh sách công việc được sắp xếp theo độ ưu tiên và chúng ta sẽ thực hiện theo thứ tự này. Nếu bạn vẫn muốn có một thứ tự chính xác hơn, hãy sử dụng kỹ thuật sau:

- *Đánh số mức độ ảnh hưởng:* công việc có mức độ ảnh hưởng nhỏ nhất được đánh số 1, những công việc có mức độ ảnh hưởng lớn hơn được đánh số lớn hơn: 2, 3, 4,... gọi là E.
- *Đánh số mức độ cấp bách:* công việc không thể trì hoãn được đánh số 1, những công việc có thể trì hoãn trong 1 giờ, 2 giờ, 1 ngày, 1 tháng... được đánh số lớn hơn: 2, 3, 10, 100... gọi là D.
- *Xác định mức độ ưu tiên của một công việc thông qua chỉ số:* $P = E / D$. Công việc có chỉ số P lớn nhất là công việc có độ ưu tiên lớn nhất. Sắp xếp danh sách công việc theo chỉ số P giảm dần chính là danh sách công việc nên được thực hiện trong danh sách TO-DO.

Với những người mới thực hành xác định độ ưu tiên của công việc, tôi đặc biệt lưu ý việc xác định đúng những công việc nằm ở góc phần tư thứ tư. Nói chung đây thường chỉ là những công việc có lead-time lớn nhưng thời gian bạn tham gia lại rất nhỏ. Đây thường là những công việc uỷ quyền sẽ được chúng ta tìm hiểu trong chương tiếp theo. Ví dụ, gọi điện cho nhân viên để trao đổi và nhắc họ về bản báo cáo cuối ngày; công việc này tốt nhất nên được thực hiện vào buổi sáng để uỷ quyền và đọc lại bản báo cáo vào buổi trưa trước khi bắt tay chỉnh sửa vào cuối giờ chiều.

Một lưu ý khác là, độ ưu tiên của các công việc không phải là một hằng số, chúng luôn biến động theo tình hình thực tế. *Dọn dẹp ổ cứng* có thể là một công việc nằm trong góc phần tư thứ ba nhưng ngay bây giờ, khi bạn muốn cài đặt thêm chương trình và dung lượng trống không còn đủ, nó lập tức đi sang góc phần tư thứ nhất.



Ngay bây giờ, bạn có thể thực hành với những công việc bạn sẽ thực hiện vào ngày mai:

1. Liệt kê các công việc bạn sẽ làm vào ngày mai, mỗi việc trên một mẩu giấy dán.
2. Sắp xếp chúng vào 4 góc phần tư như trên.
3. Thực hiện việc đánh chỉ số *mức độ ảnh hưởng (E)* và *mức độ cấp bách (D)* và tính toán độ ưu tiên thông qua công thức $P = E / D$ như trên.
4. Sắp xếp lại những công việc này theo chỉ số P giảm dần.

KHÔNG THAM LAM

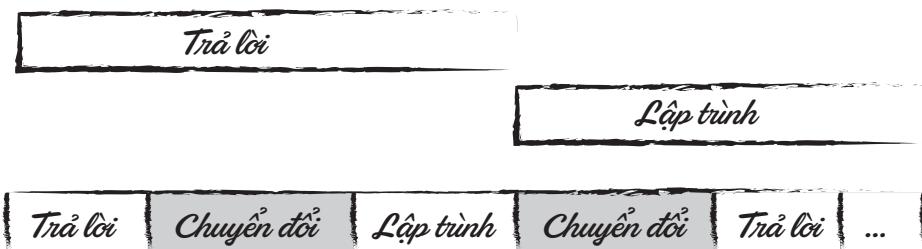
Một trong những cạm bẫy chúng ta thường gặp khi phải xử lý hàng tá công việc là *tham lam*: làm nhiều việc một lúc. Xem kia, *An thật giỏi, anh ta có thể vừa lập trình, vừa trả lời email, đồng thời lướt Facebook quan tâm tới bạn bè*; nếu bạn hâm mộ một anh chàng như vậy thì nên biết rằng, anh ta còn giỏi hơn những gì bạn tưởng; hoặc anh ta đang được trả lương không tương xứng với năng lực lập trình hiện có. Bởi anh ta đang chỉ đạt được tối đa 60% hiệu quả lập trình của mình mà thôi.

Xem nào, tôi cá rằng bạn cũng đã từng như An: An đến văn phòng vào 8:30, bắt đầu công việc lập trình; bắt cứ khi nào có email hoặc và dấu chấm đỏ trên Facebook, dừng lại và trả lời, đọc link dẫn tới những bài báo được chia sẻ trên mạng; rời khỏi văn phòng lúc 17:30 và vui mừng rằng mình đã hoàn thành một ngày đầy hiệu quả khi vừa lập trình được 3 method, biết hết mọi sự kiện xã hội trong ngày và trả lời mọi email từ đồng nghiệp. Thật đáng tiếc khi tôi phải nói rằng, một ngày như vậy thật đáng quên, vì lẽ ra bạn đã có thể xem thêm 1 bộ phim và đọc hết một chương sách. An đã bị rơi vào *bẫy đa nhiệm*.

Chúng ta thường nhầm lẫn rằng khi thực hiện đa nhiệm (thực hiện nhiều công việc trong cùng một thời điểm) là cách tốt nhất để hoàn thành nhiều công việc hơn trong một khoảng thời gian hữu hạn. Thực tế không phải vậy. Đã có rất nhiều nghiên cứu chỉ ra rằng bộ não con người không thể hoạt động theo cơ chế song song, cùng suy nghĩ về nhiều vấn đề trong một thời điểm. *Chuyển đổi công việc (task switching)* liên tục giúp chúng ta thực hiện từng phần mỗi công việc một

cách tuân tự khiến chúng ta ngỡ như mình đang thực hiện những công việc này đồng thời. Song chuyển đổi công việc thường là cái giá cao đến kinh hoàng: mỗi khi chuyển đổi lại công việc trước đó, chúng ta không thể bắt nhịp ngay với trạng thái trước đó, việc này lấy đi một khoảng thời gian để chúng ta “nhớ” lại mình đã làm gì; đôi khi chúng ta còn chẳng nhớ mình đã làm gì. Tôi xin lỗi nếu bạn thấy đoạn văn này thật lủng củng và tệ hại. Chắc chắn rồi, tôi đang viết trong lúc lướt Facebook.

Tập trung lại nào. Hãy xem xét ví dụ sau với việc trả lời tin nhắn Facebook trong khi bạn đang lập trình một method:



Hình 10.4: Phân chia thời gian cho công việc

Nếu bạn là một lập trình viên, ngay khi quay lại viết một method đang dang dở, bạn phải đọc lại method đó từ đầu? Hãy trả lời thành thật, vì tôi cũng là một lập trình viên. Điều này thực sự tệ hại. Lời khuyên chân thành của tôi là bạn hãy dừng ngay việc thực hiện đa nhiệm và ngưng ảo tưởng rằng mình có thể thực hiện đa nhiệm. Có thể câu trả lời của bạn sẽ là *tôi vẫn đang thực hiện tốt công việc theo cách đó đấy thôi* thì hãy cùng tôi làm một thí nghiệm sau đây:

- Liệt kê tất cả các công việc bạn làm từ 8:30 tới 17:30: lập trình, lướt Facebook, đọc báo...
- Ước lượng tỉ lệ thời gian bạn dành cho từng công việc: lập trình: 3 giờ, lướt Facebook: 2 giờ, đọc báo: 1 giờ...
- Thực hiện tuần tự công việc theo thời gian đã xác định: lập trình, lướt Facebook, đọc báo...: 3 giờ, 2 giờ, 1 giờ...

Và hãy cho tôi biết, liệu bạn có thể lập trình được nhiều hơn, đọc được nhiều tin hơn, biết nhiều bạn bè hơn... theo cách mới hay không.

Nếu bạn vẫn không tin, hãy tham gia những cuộc thi Hackathon, lập trình trong



Hãy cùng tôi tham gia thử thách trên; bạn sẽ cần 2 ngày cam kết thực hiện nghiêm túc và ghi nhận số liệu một cách chính xác. Tôi biết chắc chắn đây sẽ là 2 ngày *thực sự khó khăn*, song sẽ là 2 ngày thay đổi hoàn toàn cuộc đời bạn.

24 giờ và nhìn lại kết quả của chính mình. Bạn sẽ thấy bất ngờ vì một sản phẩm mất không dưới 1 tuần làm việc lại được tạo ra bởi chính bạn chỉ trong 24 giờ (chỉ tương đương 3 ngày làm việc). Bạn sẽ thấy cái giá của chuyển đổi công việc lớn đến cỡ nào.

Và Kanban cá nhân cung cấp một nguyên lý hay để đạt hiệu quả cao thông qua sự tập trung: *WIP* (*work in progress*), bất kỳ thời điểm nào, chỉ một lượng công việc nhất định được duy trì trong trạng thái đang thực hiện. Nói cách khác, số thẻ công việc trong cột *In Progress* luôn bị hạn chế. Việc này giúp chúng ta luôn tập trung nỗ lực vào những công việc hiện tại, tránh việc chuyển đổi công việc gây ra những lãng phí không cần thiết. Với mỗi người, giới hạn số lượng WIP có thể khác nhau tùy vào khả năng cũng như đặc thù công việc; song tôi khuyến nghị bạn nên bắt đầu bởi số 2 và không thoả hiệp với con số này trong một thời gian trước khi bạn biết cách đánh giá lại để tăng hoặc giảm số lượng WIP cho phù hợp.

<i>TODO</i>	<i>In Progress</i> (2)	<i>DONE</i>

Hình 10.5: Bảng có chứa WIP

2 công việc đang thực hiện ư? Thật điên rồ. Đúng vậy, mặc dù 2 công việc đang thực hiện trong mỗi thời điểm đúng với đa số mọi người; một số không cảm thấy ổn, đó là những người có nhiều công việc nằm trong góc phần tư thứ tư khi họ thực hiện rất nhiều việc uỷ quyền. Vậy thì không có lý do gì để chờ đợi công việc được hoàn thành (bởi người được uỷ quyền) trước khi bắt tay vào một công việc khác. Ví dụ, việc chuẩn bị bản báo cáo, bạn uỷ quyền cho nhân viên của mình vào buổi

sáng, nhận lại kết quả vào buổi trưa để hoàn thành bài trình bày vào buổi chiều. Và đương nhiên bạn sẽ không đợi tới buổi chiều mới hoàn thành chỉ một công việc này do giới hạn WIP. Cách tốt nhất là tạo ra thêm những cột tương ứng với trạng thái công việc.

<i>TODO</i>	<i>Định nghĩa Hoàn thành</i>	<i>Ủy quyền</i>	<i>Thực hiện</i>	<i>Kiểm tra</i>	<i>DONE</i>

Hình 10.6: Bảng Kanban có nhiều cột

Hoặc nếu không thể tìm điểm chung giữa luồng của những công việc này, bạn có thể tách nhỏ công việc ra như sau:

<i>TODO</i>	<i>In Progress</i>	<i>DONE</i>

Hình 10.7: Bảng Kanban với công việc được chia nhỏ

Tôi khuyến nghị sử dụng việc định nghĩa thêm các cột như giải pháp thứ nhất; tuy vậy vấn đề chính gặp phải là chúng ta phải tìm ra được luồng công việc chung nhất cho mọi công việc. Trong trường hợp việc này không khả thi, chúng ta có thể tạo ra nhiều dòng trong bảng Kanban tương ứng với những nhóm công việc có chung đặc điểm (về luồng công việc) như sau:

<i>TO DO</i>	<i>In Progress</i>		<i>DONE</i>
	<i>Ủy quyền</i>	<i>Kiểm tra</i>	
	<i>Thảo luận</i>	<i>Thực hiện</i>	<i>Đánh giá</i>

Hình 10.8: Bảng Kanban phân chia theo nhóm công việc

Và rồi chúng ta lại không thể chụp lại được trạng thái công việc của bảng Kanban trên trong 15 giây? Mỗi phương pháp đều có điểm lợi cũng như những hạn chế nhất định, tôi khuyến khích bạn thử nghiệm nhiều phương pháp khác nhau để tìm ra phương pháp phù hợp nhất cho chính mình. Đó chính là tinh thần Agile.

DỌN DẸP VÀ ĐÁNH GIÁ

Chúng ta sẽ làm gì với bảng Kanban vào cuối ngày? Trường hợp lý tưởng nhất là mọi công việc xuất phát từ cột *TO-DO* giờ đây đã nằm tại cột *DONE*, đồng nghĩa với mọi công việc trong ngày đã được hoàn thành. Tuy vậy, trong giai đoạn đầu thực hành Kanban cá nhân, trường hợp này rất hiếm khi xảy ra; thậm chí, chỉ một số ít công việc tại cột *TO-DO* được hoàn thành. Dù thế nào, tôi tin rằng chúng ta xứng đáng dành cho mình một phút nghỉ ngơi và cảm nhận thành quả từ những công việc đã hoàn thành. Đó là việc rất quan trọng giúp chúng ta có động lực thực hiện những công việc phía trước dựa trên những thành quả (dù rất nhỏ) đã đạt được. Và 5 hoặc 10 phút tiếp theo đây còn quan trọng hơn nhiều: *dọn dẹp và đánh giá*.

Dọn dẹp là nhín lại những công việc còn tồn đọng và quyết định xem những công việc nào cần giữ lại và thực hiện tiếp. Đôi khi chúng ta có thể thẳng tay loại bỏ những công việc này hoặc chuyển chúng xuống backlog dành cho tháng thay vì nằm ở *TO-DO* của ngày hôm sau; bởi những công việc này không còn hoặc còn rất ít giá trị. Ví dụ, công việc *gửi CV đến nhà tuyển dụng* gần như không còn giá trị nếu ngày hôm nay bạn đã có một cuộc phỏng vấn tuyệt vời.

Đánh giá gồm hai lĩnh vực; *một là, đánh giá lại mức độ ưu tiên của những công việc còn tồn đọng; hai là, đánh giá cách chúng ta đã thực hiện công việc trong ngày*. Như tôi đã nói, mức độ ưu tiên của mỗi công việc luôn biến động theo tình hình hiện

tại; nếu bạn đã thực hành tốt việc đánh giá mức độ ưu tiên liên tục trong ngày, bạn không cần phải quan tâm tới chúng vào cuối ngày. Việc đánh giá cách thực hiện thì khác, chúng ta nên làm việc này vào cuối ngày để tìm ra những điểm không hợp lý.

Chúng ta đã thực hiện khoa học chưa? Giới hạn WIP là 4 có khiến hiệu quả tập trung kém đi?

- Nếu câu trả lời là có, hãy nhanh chóng thay đổi;
- Nếu câu trả lời là *không*, hãy xem xét lại kỳ vọng, có thể chúng ta đang kỳ vọng quá cao vào bản thân khi lên kế hoạch thực hiện 10 công việc trong ngày trong khi năng lực hiện có chỉ đủ thực hiện 5 công việc. Điều chỉnh lại kỳ vọng đôi khi tốt hơn nhiều so với việc cố gắng nâng cao giới hạn của bản thân; chúng ta không muốn bị stress cũng như bị những công việc hàng ngày làm xao nhãng những chuẩn bị cho tương lai lâu dài; như dành hết thời gian gấp gáp 10 khách hàng và không còn thời gian đọc sách để tìm hiểu thị trường.



Retrospective có thể không mang lại một cách đo đạc chính xác, song bạn có thể áp dụng phương pháp *Value Stream Mapping* cho một công việc cụ thể như sau:

1. Sử dụng một tờ giấy lớn.
2. Liệt kê tất cả những bước thực hiện công việc, mỗi bước trong một hình vuông.
3. Điền giá trị thời gian cần thiết để thực hiện tại mỗi bước, gọi là T1, T2...
4. Kết nối những bước này bằng những mũi tên tương ứng với workflow thực hiện; điền giá trị độ trễ thời gian giữa hai bước tương ứng, gọi là D1, D2...
5. Đánh dấu tất cả những bước mang lại giá trị, và tính tổng thời gian của chúng, gọi là TN. Đây chính là những bước tối thiểu chúng ta cần thực hiện để hoàn thành công việc. Tính tổng thời gian của những bước khác, gọi là TW, đây chính là *thời gian lãng phí* khi thực hiện theo workflow cũ.
6. Thực hiện tương tự với thời gian trễ, gọi là DN và DW.
7. Giờ đây tôi tin rằng bạn đã tìm ra công thức về năng suất có thể tăng lên do cải tiến về workflow.

HIỂU ĐÚNG

Kanban cá nhân chỉ tập trung vào luồng công việc, không phải thời gian.

Kanban cá nhân tối đa hoá giá trị những công việc được thực hiện theo thứ tự ưu tiên; và không có gì mâu thuẫn nếu chúng ta lựa chọn thời gian là độ ưu tiên (có trọng số lớn). Hãy nhớ lại những góc phần tư, thời gian luôn là một tiêu chí về độ ưu tiên.

Việc xây dựng DoD và đánh giá độ ưu tiên cho mỗi công việc tốn quá nhiều thời gian và không cần thiết.

Những người mới thực hành Kanban cá nhân thường gặp vấn đề này, họ tốn rất nhiều thời gian xây dựng DoD và xác định độ ưu tiên cho mỗi công việc. Song việc này sẽ trở nên đơn giản và rất nhanh chóng khi bạn đã thực hành thành thục. Tôi khẳng định rằng, xác định DoD chính là 30% công việc; và đánh giá đúng độ ưu tiên đảm bảo 50% hiệu quả công việc.

Việc dọn dẹp và đánh giá có thể thực hiện vào ngày hôm sau.

Hoàn toàn có thể; nhưng sẽ tốt hơn nếu bạn thực hiện vào cuối ngày. Có ít nhất 2 lý do: một là, chúng ta sẽ dễ dàng nhớ lại những gì đã xảy ra trong ngày để rút kinh nghiệm; hai là, chuẩn bị vào buổi tối tức là bạn đã đi trước mọi người 1 bước.

Khi không còn công việc trong TO-DO chúng ta nên nghỉ ngơi.

Kanban cá nhân không mô tả việc này. Nghỉ ngơi: tốt; tiếp tục lên kế hoạch: tốt. Điều này hoàn toàn phụ thuộc mục đích và cách chúng ta định nghĩa giá trị cuộc sống; một số người luôn thích thử thách, một số người thích cân bằng. Nhưng dù với cá nhân nào, Kanban cá nhân cũng cung cấp cách thức *thông minh hơn* để tối ưu giá trị công việc.

TỔNG KẾT

Kanban cá nhân cung cấp một cách quản lý (công việc) cá nhân tập trung vào *luồng công việc nhằm tối đa hóa giá trị*.

Trực quan hóa là nguyên lý nổi bật nhất. *Tập trung* là nguyên lý quan trọng nhất. *Ưu tiên* tạo ra hiệu quả. *Luồng công việc tạo ra kiểm soát*. *Trạng thái công việc tạo ra động lực*. Hãy luôn đặt ra giới hạn WIP và cam kết thực hiện cho tới khi bạn tìm thấy lý do chính đáng để điều chỉnh.

Một bảng Kanban cá nhân thường bao gồm 2 cột cố định là *TO-DO* và *DONE*; xen giữa là những cột khác thể hiện luồng công việc; song không nên có quá nhiều cột. Việc chụp lại tình trạng mọi công việc trong 15 giây là quan trọng nhất.

1

TỐI ƯU
HIỆU QUẢ

Cũng như Phần II về phát triển phần mềm theo phương pháp Agile, tôi muốn dành chương cuối cùng của Phần III để nói thêm về những phương pháp, kỹ thuật giúp bạn tối ưu hoá hiệu quả công việc.

Kanban cá nhân là một phương pháp tuyệt vời giúp chúng ta bắt đầu với việc tối ưu hoá những công việc giá trị, nhưng những nguyên tắc đứng sau Kanban cá nhân cũng như những công cụ bổ sung sau đây thậm chí còn giúp chúng ta tăng hiệu quả lên gấp đôi.

ĐỊNH NGHĨA CÔNG VIỆC

Nếu phải chọn một hành động quan trọng nhất trước khi bắt tay làm bất cứ một công việc nào, bạn sẽ chọn hành động gì? Tôi đã hỏi rất nhiều người và câu trả lời tôi thường nhận được là *lập kế hoạch*. Thật đáng tiếc, đó không phải là câu trả lời tốt. Lập kế hoạch chỉ nên là hành động đứng thứ hai bởi nó phải nhường chỗ cho hành động *định nghĩa công việc*. An bắt đầu việc học tiếng Anh bằng cách lên một lịch trình chi tiết về việc đăng ký một khoá học, dành mỗi thời gian cuối tuần để luyện tập thêm; sau 6 tháng thực hiện kế hoạch hoàn hảo, An có thể đọc và viết các tài liệu bằng tiếng Anh, song An vẫn thấy thất vọng vì khả năng giao tiếp của mình. Một kế hoạch hoàn hảo cùng những nỗ lực tuyệt vời đã được An thực hiện nhưng không mang lại hiệu quả như An mong đợi; chỉ vì An đã bỏ đi hành động đầu tiên, và cũng là hành động quan trọng nhất, định nghĩa hoàn thành cho việc học tiếng Anh.

An không phải là một trường hợp đặc biệt. Theo nghiên cứu của Đại học Scranton, chỉ 8% những người được khảo sát nói rằng họ đạt được mục tiêu của mình trong năm 2014, phần nhiều nói rằng họ *không chắc chắn* đạt được mục tiêu hay chưa. Điều này nghe thật nực cười, nhưng sự thực là, hầu hết chúng ta không thể đánh giá kết quả công việc của mình chỉ vì đã bỏ qua hoặc thực hiện rất sơ sài hành động quan trọng nhất *định nghĩa công việc*.

Bạn có thể nhớ lại phương pháp định nghĩa hoàn thành (Definition of Done – DoD) đã được chúng ta nhắc tới trong phần II với tiêu chí INVEST (Independent, Negotiable, Valuable, Estimable, Small, Testable) để áp dụng cho công việc của cá nhân. Ngoài ra, tôi muốn giới thiệu thêm một phương pháp định nghĩa công việc khoa học được giới thiệu từ những năm 1980 nhưng vẫn cho thấy hiệu quả tuyệt vời trong thế giới hiện nay: SMART.

Phương pháp định nghĩa công việc SMART dựa trên 5 yếu tố tạo nên từ này: S.M.A.R.T = Specific, Measurable, Achievable, Relevant, Time-Bound. Trong đó:

Specific (Cụ thể). Nếu công việc mông lung, làm sao chúng ta biết phải làm gì? Một trong những sai lầm chúng ta hay gặp phải là xác định công việc một cách mơ hồ; và mơ hồ nhất là việc chúng ta bước ra khỏi nhà vào buổi sáng chỉ với một công việc duy nhất: đi làm. Nhưng cụ thể đến mức *lập trình chức năng A, gấp đối tác B...* thì không nhiều người xác định được trước khi chúng ta bước chân ra khỏi nhà vào 8 giờ sáng.

Measurable (Đo được). Nếu công việc không đo được, làm sao chúng ta biết khi nào mình đã hoàn thành? Đây chính là yếu tố chúng ta dễ dàng bỏ qua và rồi vắt kiệt sức mình với một công việc mơ hồ; và mơ hồ nhất có lẽ là việc *kiếm tiền*. Nhưng nếu không định nghĩa rõ ràng việc kiếm 5 triệu hay 10 triệu thì bao giờ chúng ta sẽ dùng lại đây? *Trở thành tỉ phú đô la dù sao cũng khoa học hơn kiếm được nhiều tiền.*

Achievable (Khả thi). Nếu công việc không khả thi, làm sao chúng ta thực hiện? Chúng ta có thể nghĩ lớn, đặt ra những mục tiêu vĩ đại cho cuộc đời mình. Nhưng hãy lưu ý, đó chỉ là *tâm nhìn*; khi định nghĩa một công việc cụ thể, chúng ta cần chắc chắn rằng công việc đó khả thi tại thời điểm hiện tại hoặc tương lai gần. *Gặp tổng thống Obama vào tuần sau để bàn về giải pháp chống khủng bố* là một công việc bất khả thi với 99.999% dân số thế giới; tại sao không bắt đầu bởi việc *gửi thư cho tổng thống Obama để nêu quan điểm của mình về giải pháp chống khủng bố vào tuần sau?* Thật bất ngờ là công việc đó khả thi với ít nhất 30% dân số, những người sử dụng Internet; và chúng ta có thể thực hiện.

Relevant (Liên quan). Nếu công việc không có sự liên quan (tới cuộc sống, những mục tiêu khác), chúng ta thực hiện công việc có ý nghĩa gì? Đây là điểm đặc biệt quan trọng, bởi trong thế giới cá nhân, chúng ta có thể có hàng trăm việc để làm mỗi ngày, hãy lựa chọn đúng công việc cần thiết.

Time-Bound (Có thời hạn). Nếu công việc không giới hạn thời gian, khi nào nó được bắt đầu và kết thúc? Hãy nhớ lại phương pháp quản lý thời gian; bởi thời gian của chúng ta là hữu hạn, ai quản lý thời gian tốt hơn, người đó có khả năng thành công cao hơn.

Thật là phức tạp. Tôi hoàn toàn không bất ngờ nếu bạn nhận xét như vậy về phương pháp SMART; hàng chục người tôi gặp đều nói như vậy khi mới tìm hiểu qua về phương pháp này. Song với những người kiên trì thực hiện, họ nhanh chóng cảm thấy sự tự nhiên mỗi khi định nghĩa một công việc. Nếu bạn cảm thấy việc sử dụng phương pháp SMART tốn quá nhiều thời gian cho từng đầu công việc, hãy bắt đầu với việc sử dụng SMART cho một mục tiêu lớn trong năm, rồi mục tiêu tháng, và đến mục tiêu tuần. Lâu dần, SMART trở thành một phần tất yếu trong mỗi công

việc hàng ngày của bạn. Hãy bắt đầu với một ví dụ về mục tiêu năm, mọi người hay đặt mục tiêu *năm nay kiếm được nhiều tiền*; và mục tiêu đó thực sự không khoa học. Thế nào là nhiều (measurable)? 1 tỷ USD là nhiều (achievable)? Tiền để làm gì (relevant)? Một mục tiêu tốt hơn là: *để phục vụ cho công việc (relevant), năm nay (time-bound) kiếm được thêm 50 triệu (measurable) để mua laptop mới (specific) nhờ việc bán hàng online (achievable)*. Yếu tố achievable là khó nhất, nếu chúng ta đã có kỹ năng bán hàng online tốt, đã có vốn thì mục tiêu này khả thi. Ngược lại, mục tiêu không khả thi. Hãy thử SMART với một công việc hàng ngày: *gặp ông Bình, khách hàng tiềm năng của công ty thực sự là một định nghĩa công việc tôi*. Chúng ta cần một định nghĩa tốt hơn như: *để mở rộng mạng lưới khách hàng (relevant), gặp ông Bình trong 30 phút (time-bound), giới thiệu về 2 giải pháp công ty có thể cung cấp (specific) nhằm xác định khả năng tiến tới đàm phán hợp đồng (measurable)*. Với một định nghĩa công việc rõ ràng như vậy, chúng ta hoàn toàn tự đánh giá được mức độ hoàn thành công việc của mình. Nếu chúng ta kết thúc buổi họp và cung cấp được 2 giải pháp tới khách hàng, chúng ta đã hoàn thành công việc; nếu hợp đồng được ký kết ngay trong buổi họp, chúng ta đã thành công ngoài mong đợi.

Chúng ta đã nghe về tầm quan trọng của việc đặt những câu hỏi 5W1H khi tiếp cận một công việc cụ thể; thì theo tôi, thực hành phương pháp SMART cho chúng ta chính xác câu trả lời của những câu hỏi 5W1H.

SMART	5W	Ý nghĩa
Specific	What	Công việc cụ thể.
Measurable	Where	Đích đến, kết quả của công việc chúng ta muốn đạt được.
Achievable	How	Yếu tố này không thực sự trả lời được câu hỏi <i>how</i> về cách thực hiện. Nhưng nói chung, khi chúng ta đánh giá được công việc là khả thi, yên tâm, chúng ta sẽ biết cách làm.
Relevant	Why	Lý do thực hiện công việc.
Time-bound	When	Khi nào công việc được bắt đầu, kết thúc, và đánh giá.

Gần đây, một phát triển mới từ phương pháp S.M.A.R.T là S.M.A.R.T.E.R, được thêm vào 2 yếu tố nữa E.R. Có nhiều biến thể của phương pháp này, nhưng phiên bản tôi thấy thích nhất là *E.R = Evaluate, Re-Adjust*. Phiên bản S.M.A.R.T.E.R với E.R như trên không thường cần thiết với những công việc nhỏ, song lại rất hữu ích với

những công việc hoặc mục tiêu lớn.

Evaluate (Đánh giá). Một mục tiêu khoa học cần có trong đó khả năng đánh giá được qua từng giai đoạn hoặc tiến trình đạt được mục tiêu.

Re-Adjust (Điều chỉnh). Trong quá trình thực hiện, nếu mục tiêu đã lỗi thời hoặc cách tiếp cận của chúng ta không còn đúng, mục tiêu ngay lập tức được điều chỉnh lại cho phù hợp với tình hình hiện tại.

Ví dụ, với mục tiêu chúng ta đã có theo S.M.A.R.T ở trên, có thể bổ sung như sau: *để phục vụ cho công việc (relevant), năm nay (time-bound) kiểm được thêm 50 triệu (measurable) theo từng tháng (evaluate) để mua laptop mới (specific) nhờ việc bán hàng online (achievable)*. Như vậy, cứ sau mỗi tháng, chúng ta lại đánh giá quá trình đạt được mục tiêu. Nếu đến tháng 5, chúng ta chỉ kiểm được 10 triệu thì cần phải điều chỉnh lại mục tiêu hoặc cách làm. Cũng có thể, đến tháng 5 chúng ta vẫn chưa phải điều chỉnh gì nếu mục tiêu là *để phục vụ cho công việc (relevant), năm nay (time-bound) kiểm được thêm 50 triệu (measurable), tập trung vào tháng 6,7 (Euro 2016) để mua laptop mới (specific) nhờ việc bán hàng online (achievable)*. Cũng có thể, tháng 4, chúng ta trúng xổ số 100 triệu, thì mục tiêu cũng đã lỗi thời, có thể nên loại bỏ (mục tiêu *kiểm thêm 50 triệu mua laptop* đã hoàn thành, nhưng nếu chúng ta đặt mục tiêu *kiểm thêm 50 triệu mua laptop và thử sức trong lĩnh vực kinh doanh* thì việc trúng xổ số có rất ít liên quan đến mục tiêu này bị loại bỏ).

S.M.A.R.T.E.R = Specific, Measurable, Achievable, Relevant, Time-Bound, Evaluate, Re-Adjust

Xin nói thêm lý do khiến tôi thích phiên bản S.M.A.R.T.E.R này vì nó rất *agile*: *chúng ta luôn phải đánh giá và điều chỉnh liên tục để có cách làm phù hợp nhất, nhằm hoàn thành phần công việc có giá trị nhất giúp đạt được mục tiêu có ý nghĩa nhất*. Dù vậy, có thể bạn cho rằng yếu tố *Evaluate* và *Re-Adjust* gắn với quá trình thực hiện mục tiêu, không gắn với việc xây dựng mục tiêu ban đầu. Nhưng mục tiêu đặt ra là để đạt được và có ý nghĩa, đâu phải chỉ để “đặt ra”, phải không? Hơn nữa, qua ví dụ trên bạn cũng thấy yếu tố *Evaluate* hoàn toàn có thể được đặt ra ngay trong lúc xây dựng mục tiêu.

Hãy tập cho mình cách định nghĩa công việc một cách rõ ràng trước khi bắt tay vào từng công việc cụ thể. Bằng việc dành ra 15-20 phút cho việc viết ra định nghĩa công việc trước khi bắt đầu, tin tôi đi, bạn có thể tiết kiệm được ít nhất 20% thời gian và có thể tăng đến 80% hiệu quả công việc. 20% thời gian dành cho việc gửi một email có thể không phải là điều bạn quan tâm, nhưng định nghĩa

rõ ràng công việc cập nhật và gửi CV đến nhà tuyển dụng có thể thay đổi công việc của bạn trong một vài năm, thậm chí là bước ngoặt của sự nghiệp. Thậm chí, ngay sau 10 phút dành cho việc định nghĩa công việc, bạn thấy ngay bước tiếp theo là *ném công việc vào sọt rác* bởi thật ngớ ngẩn nếu thực hiện một công việc không có ý nghĩa, hoặc không khả thi; điều hoàn toàn có thể xảy ra sau khi bạn đã tiêu tốn rất nhiều thời gian và công sức mới nhận ra; có lẽ chiếm đến 50% số lượng công việc của bạn.



Sử dụng phương pháp tính toán độ ưu tiên $P = E / D$ để tìm ra công việc có độ ưu tiên cao nhất của bạn vào ngày mai.

Sử dụng phương pháp S.M.A.R.T để định nghĩa rõ công việc này.

TẬP TRUNG

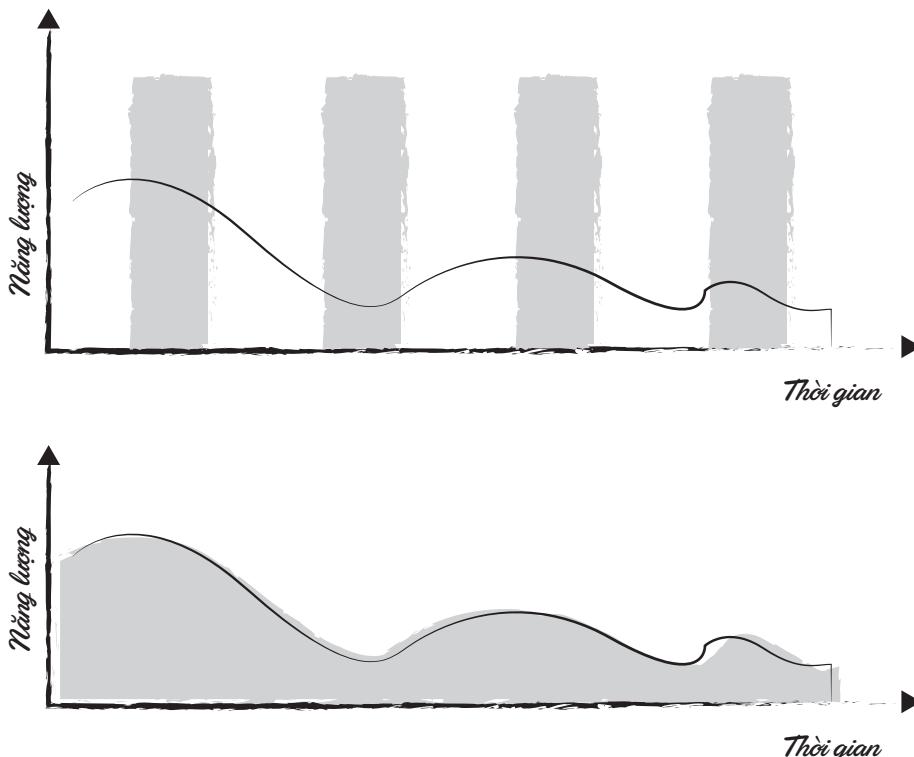
Nếu phải chọn một hành động quan trọng nhất khi nghĩ đến bất cứ một công việc nào, bạn sẽ chọn việc gì? Tất nhiên, không phải định nghĩa công việc đâu, bạn mới “nghĩ” đến công việc thôi. Hãy ngay lập tức nghĩ tới ý nghĩa của công việc. Trả lời được câu hỏi Why bạn cũng sẽ xác định được yếu tố Relevance trong phương pháp S.M.A.R.T.

Nếu ngày hôm nay, bạn chỉ thực hiện được 1 công việc, đó là công việc nào? Nếu ngày hôm nay, bạn chỉ thực hiện 2 công việc, công việc tiếp theo là gì? Nếu ngày hôm nay, bạn chỉ thực hiện 3 công việc, công việc tiếp theo là gì?

Sau khi trả lời 3 câu hỏi đó, tôi nghĩ bạn nên dừng lại. Đây được coi là *số 3 kỳ diệu* được đưa ra trong cuốn sách *Thuật quản lý thời gian* của Brian Tracy. Người tốt nhất không phải là người thực hiện nhiều công việc nhất; người tốt nhất thực hiện ít công việc nhưng mang lại hiệu quả cao nhất.

Tôi đã trình bày trong chương đầu tiên, chúng ta đã đi qua thời kỳ *quản lý thời gian* để cố gắng làm nhiều công việc nhất trong thời gian hạn chế hàng ngày; giờ đây chúng ta đang ở trong thời kỳ *quản lý năng lượng* để đạt được hiệu quả cao nhất trong một không gian xác lập bởi lượng công việc và thời gian hàng ngày. Và chúng ta không có giải pháp nào khác ngoài kỹ năng *quản lý sự tập trung*.

Hãy cùng nhìn lại *diện tích hiệu quả* tôi đã đưa ra:



Hình 11.1: *Diện tích hiệu quả*

Bằng việc giới hạn lượng công việc được thực hiện nhưng mở rộng sự tập trung vào từng công việc giúp cho năng lượng bạn dành cho từng công việc là lớn nhất có thể; chúng ta có thể giúp cho *diện tích hiệu quả* tăng lên. Và ngay cả khi *diện tích hiệu quả* không tăng lên, chúng ta vẫn cảm thấy thoải mái vì tiết kiệm được thời gian. Nếu bạn chưa biết, tôi xin giới thiệu qua một chút về nguyên tắc 80/20. Đây là một nguyên tắc cơ bản trong kinh tế học và đã được chứng minh trong cuộc sống (nguyên tắc Pareto). Hiểu đơn giản, nguyên tắc này cho thấy 80% hiệu quả chúng ta có đến từ 20% nỗ lực, 80% nỗ lực còn lại chỉ mang về 20% hiệu quả. Hay, 80% những gì bạn thấy hay nhất trong cuốn sách này chỉ được viết ra bởi 20% nỗ lực của tôi mà thôi, 20% còn lại chiếm tới 80% công sức của tôi trong việc gõ văn bản, chỉnh sửa câu chữ... Và trong một núi công việc chúng ta có hàng ngày, ai tìm ra 20% công việc mang lại hiệu quả cao nhất chính là người chiến thắng.

Nhưng tôi vẫn có hàng trăm việc khác cần hoàn thành. Đúng. Và chúng ta sẽ biết cách hoàn thành hàng trăm công việc còn lại.

UỶ QUYỀN

Ai cũng cần được giúp đỡ. Thế kỷ 17 bắt đầu với học thuyết về quy trình của Adam Smith bằng việc phân định mỗi người làm một công việc cụ thể trong một dây chuyền sản xuất đã làm năng suất lao động tăng tới 20.000% chẳng qua cũng là một phép ánh xạ cách cuộc sống vận hành vào từng hệ thống sản xuất: *chuyên môn hoá*. Cuối thế kỷ 20 là cuộc bùng nổ của những công cụ trợ giúp công việc hàng ngày: hãy để hộp thư thoại tự động nhận cuộc gọi mỗi khi bạn đang bận họp. Thế kỷ 21 sẽ là cuộc cách mạng tiếp theo của sự trợ giúp dựa trên những thành tựu công nghệ và giao tiếp giữa con người.

Dù chúng ta chọn 3 hay 20% số công việc mang lại 80% hiệu quả, hãy nhớ rằng chúng ta còn 80% nữa cần hoàn thành; và tất cả những gì chúng ta cần là *uỷ quyền*. Uỷ quyền là giao phó công việc cho một công cụ, cá nhân hay tổ chức khác thực hiện nhằm đảm bảo công việc đó được thực hiện với chỉ một chút nỗ lực của bản thân. Và chúng ta tiết kiệm 80% thời gian.

Nhưng tôi không phải là sép, tôi không thể yêu cầu người khác làm việc thay tôi. Đó là suy nghĩ đáng tiếc nhất khiến chúng ta luôn quay cuồng trong hàng ngàn công việc. Hãy biết rằng, bất cứ ai cũng có thể thực hành uỷ quyền: một người đánh giày uỷ quyền việc nấu ăn bằng cách mua cho mình một suất cơm bụi; một giáo sư ở trường đại học uỷ quyền việc chuẩn bị chỗ ăn nghỉ cho phòng hành chính để tập trung vào bài thuyết trình tại hội thảo. Hãy nhớ rằng, bất cứ ai cũng có thể uỷ quyền nếu có một trong bốn yếu tố: *tiền bạc, công nghệ, quan hệ, và quyền hạn*. Nếu bạn đang đọc cuốn sách này, tôi tin chắc chắn rằng, bạn có ít nhất một trong bốn yếu tố đó. Vậy bạn còn chờ đợi gì để không giải thoát 80% thời gian cho mình?

Uỷ quyền dựa trên tiền bạc là cách uỷ quyền đơn giản nhất. Bất cứ công việc nào bạn không muốn thực hiện nhưng lại cần thiết, hãy tìm đến những dịch vụ tốt nhất và uỷ quyền. Hàng ngày chúng ta thực hiện vô thức hàng trăm việc uỷ quyền như vậy: mua một suất ăn, một ly cafe, sửa chiếc điện thoại... Nấu một bữa cơm, pha một ly cafe, thậm chí sửa chiếc điện thoại, chúng ta đều có thể tự làm được; nhưng chúng ta muốn có thêm thời gian cho những việc khác và lâu dần thói quen trả tiền cho những dịch vụ được chúng ta thực hiện. Nhưng đừng dừng lại ở đây, hãy thực hiện nhiều hơn việc uỷ quyền một cách *có ý thức*; hãy phân tích thiệt hơn mỗi khi chúng ta thực hiện một công việc hoặc uỷ quyền cho một dịch vụ khác. *Tôi không thích công việc đơn dẹp nhà cửa, nó chiếm quá nhiều thời gian*; nhưng nếu bạn vẫn muốn sống trong một ngôi nhà sạch đẹp, hãy tìm một người giúp việc. Hãy tìm người giúp việc tốt nhất mà bạn có thể chi trả, bạn uỷ quyền cho họ để có thời gian nghỉ ngơi hoặc tập trung vào công việc khác; và bạn chắc chắn không muốn bị phân tán bởi hàng tá những câu hỏi như *làm sao để lau sạch bếp*

chỉ vì tiết kiệm được thêm một chút tiền. Đừng bị chi phổi quá nhiều bởi chi phí, ngay cả khi bạn chỉ kiếm được 50.000 đồng / giờ làm việc mà phải chi trả 100.000 đồng / giờ giúp việc. Chẳng phải bạn đang có thời gian làm việc, đọc sách, phát triển cá nhân đó sao? 100.000 đồng đó là chi phí nếu bạn đang chi trả và ngồi không một cách vô lý; ngược lại, đó là khoản đầu tư cần phải có nếu bạn đang dùng 1 giờ đó để làm việc và phát triển bản thân.

Uỷ quyền dựa trên công nghệ là một trong những kỹ năng không thể thiếu của thế kỷ 21. Bạn muốn một ly cafe? Hãy sử dụng Foody để tìm kiếm và đặt hàng. Bạn không muốn dành cả ngày để đọc email? Hãy đặt chế độ trả lời tự động *nếu có việc cần, hãy gọi cho tôi;* và bạn có thể xoá tất cả những email của mình vào cuối ngày mà không cần đọc. Không, tôi đang rất nghiêm túc, chỉ những người đã gọi cho bạn, đó là những email bạn cần lưu ý, bạn không cần quan tâm tới số email còn lại. Hãy biết rằng, thành tựu công nghệ hiện tại cho chúng ta muôn vàn công cụ để tiết kiệm thời gian và công sức: đặt lịch thanh toán tự động cho hoá đơn tiền điện từ ngân hàng, tự động thanh toán khoản tín dụng khi đến hạn, tự động gửi điện hoa tới khách hàng vào dịp sinh nhật... Và nếu những dịch vụ như vậy chưa tồn tại, điều tuyệt vời nhất một lập trình viên có thể làm, là viết ra những chương trình: tự động bật máy pha cafe, tự động bật bình nước nóng... vào mỗi buổi sáng. Có rất nhiều lập trình viên biết cách duy trì hạnh phúc gia đình chỉ với một chương trình đơn giản, tự động gửi SMS "*vợ à, anh có việc cần làm thêm nên về muộn*" vào 18:30 nếu laptop của anh ta chưa tắt.

Ngay cả khi không có tiền, không biết sử dụng công nghệ, chúng ta vẫn có thể *uỷ quyền dựa trên quan hệ*. Bất cứ ai trong cuộc sống đều có những mối quan hệ nhất định, rất nhiều người trong số đó sẵn sàng giúp đỡ chúng ta khi cần. Khi bạn muốn mua một chiếc laptop, chẳng phải việc đầu tiên là nhắc điện thoại lên, và gọi cho một người bạn làm trong lĩnh vực CNTT đó sao? Năm nay, tôi dọn về căn nhà mới để sống cùng gia đình. Thú thật, cho tới trước khi nhận nhà, tôi chưa bao giờ đặt chân tới đó, dù chính tôi là người ký các hợp đồng mua bán, tín dụng với ngân hàng. Tôi chỉ quyết định thời điểm mua, các khoản vay dựa trên tình hình tài chính cá nhân; mọi công việc khác đều do những thành viên khác trong gia đình tôi trợ giúp. Có thể bạn không tin, đây thậm chí là căn nhà riêng đầu tiên của tôi. Nhưng rõ ràng, uỷ quyền cho những người khác trong các công việc về tìm kiếm thông tin, lựa chọn dự án, thủ tục... giúp tôi có nhiều thời gian hơn tập trung vào những công việc có giá trị về tài chính. Và cách làm này hiệu quả hơn nhiều. Cuộc sống ngày nay dựa trên sự giao tiếp và tinh thần trợ giúp, cộng tác cao hơn; hãy tìm tới đúng người trong những mối quan hệ mà chúng ta có, và uỷ quyền cho họ để có kết quả tuyệt vời. Tôi không vô tình khi xếp uỷ quyền dựa trên quan hệ ở vị trí thứ 3; hãy biết rằng, thời gian của ai cũng đều quan trọng như nhau; chúng ta chỉ nên uỷ quyền dựa trên quan hệ nếu không thể thực hiện 2 giải pháp uỷ quyền

dựa trên tiền bạc và công nghệ.

Và cuối cùng, một người lãnh đạo hiệu quả thường là người có kỹ năng uỷ quyền *dựa trên quyền hạn* tốt. Ngay khi chúng ta có vị trí, quyền hạn nhất định, chúng ta đã sẵn sàng chết ngập trong đó nếu không biết cách sử dụng quyền hạn hiệu quả. 70% là con số được Brian Tracy đưa ra trong cuốn *Thuật quản lý thời gian*: Hãy uỷ quyền cho bất cứ nhân viên nào có thể thực hiện được công việc và có thể mang lại ít nhất 70% hiệu quả như bạn thực hiện công việc đó. Đừng quá lo lắng, tỉ lệ đó sẽ tăng lên nhanh chóng qua mỗi lần thực hiện. Nếu những nhân viên bán hàng có thể mang về 7 hợp đồng cho công ty thay vì 10 hợp đồng với tự bạn thực hiện việc đàm phán, hãy tiếp tục để họ làm. Cái bạn cần là thời gian để tìm ra phương pháp để mang về 30 hợp đồng, chứ không phải là trực tiếp làm thay những nhân viên của mình để có thêm 3 hợp đồng.

Trong cuộc sống, bạn có thể thấy hàng tá những công việc có thể uỷ quyền hàng ngày một cách đơn giản, từ những công việc gia đình như dọn dẹp đến di chuyển hay sắp xếp một chuyến đi... Tất cả đều hướng đến một mục tiêu tối thượng là dành lấy thời gian để chúng ta tập trung vào những công việc có lợi ích hơn. Nhưng hãy cân nhắc, bởi dù sử dụng phương thức uỷ quyền nào, chúng ta cũng đều phải đánh đổi với những phần tương ứng. Một chỉ dẫn để chúng ta quyết định thực hiện hay uỷ quyền một công việc thường có sau khi trả lời những câu hỏi sau:

- Công việc này có cần những kiến thức, kỹ năng, thông tin gì thực sự đặc biệt mà chỉ bạn mới có thể thực hiện không?
- Công việc này có giúp phát triển kỹ năng bản thân mà bạn quan tâm không?
- Công việc này sẽ không lặp lại trong tương lai?
- Công việc này sẽ ảnh hưởng tới những thành công sau này và cần sự quan tâm đặc biệt?
- Bạn không đủ thời gian để uỷ quyền một cách hiệu quả?

Nếu bạn trả lời có với một hoặc một vài câu hỏi trên, thì vấn đề uỷ quyền cần được cân nhắc. Ví dụ, việc gọi điện hỏi thăm gia đình thì chắc chắn không thể thực hiện được bởi người khác; tuyển dụng nhân sự cấp dưới cũng sẽ ảnh hưởng tới thành công sau này của tổ chức cũng như của cá nhân khiến chúng ta cần sự quan tâm đặc biệt. Gặp gỡ 20% khách hàng đặc biệt, những người mang lại 80% doanh thu cho tổ chức nên được bạn trực tiếp thực hiện; nhưng việc gửi thiếp chúc mừng tới 80% khách hàng còn lại nên được uỷ quyền cho người khác.

Để tránh vướng vào tình thế khó của câu hỏi thứ 5, chúng ta nên thực hiện việc uỷ quyền ngay khi còn có thể thực hiện. Bởi khi thực hiện uỷ quyền, chúng ta cần

định nghĩa rõ công việc với người được uỷ quyền; đó là lý do chúng ta cần định nghĩa cho bất kỳ công việc nào. Khi công việc được định nghĩa rõ ràng, chúng ta sẽ nhanh chóng nhận biết đó là công việc của mình; hoặc tìm ra cách thức và người phù hợp để uỷ quyền. Khi chúng ta vướng vào một mó hỗn độn những công việc không rõ ràng, chúng ta không đủ thời gian để thực hiện uỷ quyền và nhanh chóng nằm trong vòng xoáy bận rộn và không hiệu quả.

Những người quản lý cuộc sống tốt nhất chính là những người có khả năng uỷ quyền tuyệt vời; đây là một vấn đề không đơn giản, nhưng bạn có thể tìm thấy nhiều chỉ dẫn hơn trong cuốn sách *Thuật uỷ quyền và giám sát* của Brian Tracy.

SỐNG THEO SPRINT

Cuộc đời không phải là cuộc thi marathon, cuộc đời là những vòng lặp phát triển. Hãy nhớ rằng, dù chúng ta sống ở đâu trên trái đất này, chúng ta đều định nghĩa thời gian theo ngày, tháng, năm. Và chẳng phải vào những ngày đầu năm, chúng ta đều chúc nhau một năm mới an lành và hỏi nhau có gì mới vào cuối năm đó sao? Nhưng giờ đây, thế giới biến đổi quá nhanh, chúng ta cần đi những phân đoạn nhỏ hơn một năm.

Trong khoảng 10 năm qua, hàng năm, tôi thường dành trọn 2 ngày nghỉ đầu năm; một ngày để nhìn lại những gì đã làm được trong năm cũ, vẽ lại sơ đồ về giá trị bản thân; một ngày để lên kế hoạch cho năm tiếp theo. Phương pháp này đã giúp tôi rất nhiều, song càng ngày càng tỏ ra kém hiệu quả; rất nhiều kế hoạch của tôi đã偏离 hướng vào giữa năm; rất nhiều những cơ hội và công việc tình cờ buộc tôi phải phản ứng mềm dẻo hơn với kế hoạch đã đặt ra. Do vậy, khoảng 2 năm gần đây, bên cạnh việc thực hiện 2 ngày nghỉ theo phương pháp cũ, tôi dành thêm 30 phút vào mỗi ngày thứ 7; đây là khoảng thời gian bất khả xâm phạm để tôi ngồi nhìn lại những gì đã thực hiện được trong tuần, và những gì sẽ làm trong tuần tiếp theo.

Nếu bạn còn nhớ tầm quan trọng của sự kiện Retrospective của nhóm, bạn cũng sẽ hiểu tại sao mỗi cá nhân lại cần một sự kiện như vậy cho riêng mình.

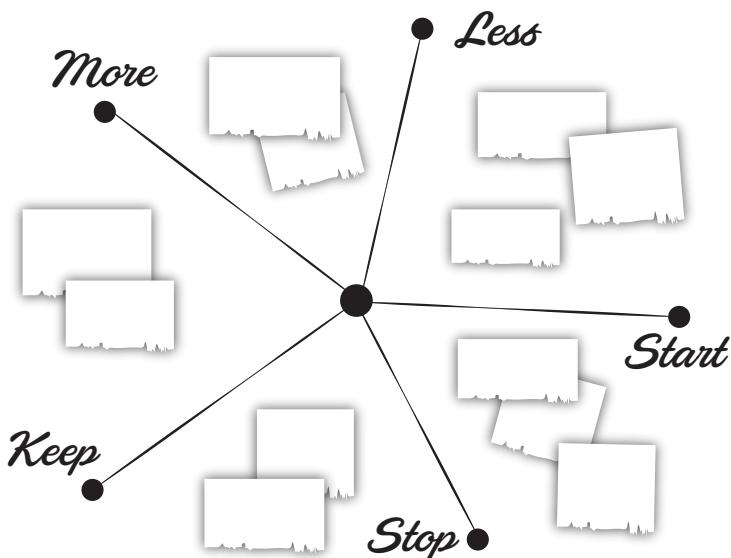
Một trong những vấn đề lớn nhất chúng ta hay gặp phải là bị cuốn theo quán tính của thời gian và mất kiểm soát trong việc quản lý sự thay đổi của bản thân cho phù hợp với thực trạng hiện tại. Và Retrospective cá nhân chính là khoảng lặng để chúng ta nhìn lại và không bị cuốn theo quán tính. Dù chúng ta là ai, làm gì, thì bản chất hàng tuần vẫn là những vòng lặp: ăn, ngủ, giao tiếp với đồng nghiệp, gặp gỡ khách hàng, đọc sách, lướt web, thanh toán hoá đơn... Hàng trăm công việc mỗi tuần, là hàng trăm công việc có thể cải tiến. Làm sao để ăn ngon hơn, ngủ sâu hơn, giao tiếp với đồng nghiệp tốt hơn, đọc sách nhanh hơn, thanh toán hoá

đơn giản hơn... Nếu một tuần, chúng ta tăng hiệu suất làm việc chỉ 1%, sau một năm, hiệu suất làm việc của chúng ta sẽ tăng lên bao nhiêu, bạn có hình dung được không? Công thức đơn giản lắm:

$$(101\%)^{52} = 167\%$$

Hình 11.2: Cải tiến trong năm theo tuần

Thật sự không thể tin nổi, một con số quá lớn. Nếu một tuần chúng ta nâng cao khả năng đọc sách chỉ 1%, ngày này sang năm chúng ta đã đọc được gấp rưỡi số sách trong năm trước. Và 1% là con số khả thi? Chắc chắn rồi. Vấn đề là chúng ta cần một khoảng lặng để nhìn lại và tìm ra 1% đó. Và không cách nào tốt hơn là thực hiện Retrospective cá nhân hàng tuần. Bạn có thể tìm thấy một số kỹ thuật Retrospective trong phần II của cuốn sách này, như kỹ thuật starfish dưới đây:



Hình 11.3: Retrospective theo phương pháp Starfish

Một trong những người thành công nhất hiện nay, COO của Facebook, Sheryl Sandberg là người áp dụng thành thực phương pháp này. Bà đã giao kèo với Mark Zuckerberg rằng hai người cần có buổi nói chuyện hàng tuần về những gì bà có thể cải tiến bất kể Mark Zukenburg có tham gia vào công việc của bà hay không. Và khi việc cải tiến theo tuần ít hiệu quả, họ thực hiện *bất cứ khi nào*.

HIỂU ĐÚNG

Không nên sử dụng SMART để định nghĩa hoàn thành cho những công việc nhỏ, vì nó khiến thời gian hoàn thành công việc nhiều hơn.

Tôi hoàn toàn đồng ý, với những công việc nhỏ như dọn nhà, lau bàn ghế... thì việc sử dụng SMART là rất *ngớ ngẩn*. Tuy vậy, trong giai đoạn đầu thực hành, tôi nghĩ bất cứ ai cũng nên sử dụng một phương pháp cụ thể để định nghĩa hoàn thành công việc, điều này giúp tạo lập một thói quen tốt. Và tốt nhất, là định nghĩa hoàn thành nên được viết ra. Sau khi định nghĩa hoàn thành đã trở thành một thói quen trong bất cứ công việc nào, chúng ta có thể lưu trữ chúng trong đầu với những công việc đơn giản và cần ít thời gian

Uỷ quyền dựa trên quyền hạn hoặc quan hệ là tốt nhất.

Rất nhiều người nghĩ như vậy, bởi chúng ta có thể tiết kiệm thời gian và tiền bạc. Song tôi cần nhấn mạnh lại rằng, thời gian của mỗi người là quý giá như nhau, đừng lạm dụng mối quan hệ tốt đẹp để sử dụng uỷ quyền dựa trên quan hệ; hãy sử dụng chúng như phương án sau cùng. Nếu bạn làm trong lĩnh vực CNTT và uỷ quyền dựa trên quan hệ cho bạn của mình để mua chiếc xe máy, hãy chuẩn bị một khoảng thời gian để đáp lễ bằng việc cài đặt Windows miễn phí; sẽ không có bất cứ *bữa ăn trưa miễn phí* nào. Tuy vậy, nếu bạn sẵn lòng, uỷ quyền dựa trên quan hệ sẽ cho chất lượng công việc cao hơn rất nhiều.

TỔNG KẾT

Hãy luôn có định nghĩa hoàn thành công việc trước khi bắt đầu lên kế hoạch cho bất cứ công việc nào; điều này giúp chúng ta xác định rõ kỳ vọng, tiến độ công việc vào bất cứ thời điểm nào.

Luôn tập trung, hãy làm những công việc giá trị nhất hơn là làm nhiều việc.

Cố gắng uỷ quyền với những công việc không mang lại nhiều giá trị cho bản thân, đặc biệt không giúp ích trong việc phát triển cá nhân. Hãy cố gắng uỷ quyền dựa trên công nghệ và tiền bạc với những dịch vụ tốt nhất có thể; sau đó là uỷ quyền dựa trên quan hệ và quyền hạn.

Sống theo Sprint. Luôn nhìn lại và đánh giá cách làm trong của những công việc đã qua theo một chu kỳ đều đặn; lựa chọn một số ít cải tiến song hãy có cam kết chặt chẽ.

PHỤ LỤC

THAM KHẢO

Agile Y chỉ là sự khởi đầu.

Cảm ơn bạn đã đọc đến phần cuối cùng của cuốn sách. Như bạn đã biết, mục đích chính của cuốn sách này là mang đến những kiến thức, khái niệm cơ bản và sơ khai về Agile: tại sao việc phát triển phần mềm hiện đại cần tới Agile, những kỹ thuật cũng như vấn đề gặp phải khi thực hành Agile trong cá nhân, nhóm phát triển phần mềm và tổ chức phát triển phần mềm. Sau gần 2 thập kỷ được cổ vũ, triển khai và chứng minh tính hiệu quả, Agile đang dần hoàn thiện về chiều sâu với nhiều công cụ chất lượng, cũng như chiều rộng với nhiều lĩnh vực ngoài phần mềm.

Agile Y đã phần nào cung cấp được góc nhìn theo bề rộng của việc thực hành Agile trong việc phát triển phần mềm; nhưng không thể đi sâu vào một vấn đề cụ thể trong một vài chương sách ngắn. Phụ lục này giới thiệu một số tài liệu tham khảo; là một phần tôi đã tìm hiểu và tóm lược cho những nội dung trong cuốn sách; cùng những đánh giá cơ bản của cá nhân. Tôi hy vọng bạn có thể tìm thấy nhiều kiến thức bổ ích và đi sâu hơn vào thế giới Agile qua những tài liệu này.

Những tài liệu ở dưới được phân nhóm theo những phần của cuốn sách và được sắp xếp theo quan điểm của tôi theo tiêu chí từ *phải đọc đến nên đọc*. Do đó bạn có thể tìm hiểu dần theo thứ tự này.

CHO CÁ NHÂN

Time Management

Cuốn sách này được dịch ra tiếng việt với tên gọi *Thuật quản lý thời gian*. Brian Tracy là bậc thầy về việc tăng hiệu suất làm việc cá nhân với một loạt những đầu sách như *Thuật quản lý thời gian*, *Thuật uỷ quyền và giám sát...*; những cuốn sách của ông luôn ngắn gọn, tập trung và dễ thực hành. Cuốn sách này đưa ra nhiều kỹ thuật quản lý thời gian kiểu mới, hiện đại và chứa những nội dung liên quan tới việc quản lý năng lượng và quản lý sự tập trung; hoàn toàn không chỉ giới hạn trong việc quản lý thời gian truyền thống như tiêu đề.

Getting Things Done: The Art of Stress-Free Productivity

Một cuốn sách thuộc hàng best seller của David Allen, được tái bản và dịch ra nhiều ngôn ngữ trên thế giới, được xuất bản ở Việt Nam với tựa *Hoàn thành mọi việc không hề khó: Nghệ thuật thực thi không căng thẳng*; nên nằm trong danh sách phải đọc của bạn. David Allen đưa ra cách tiếp cận rất bài bản, từ nhu cầu cần một hệ thống để quản lý cá nhân một cách hiệu quả, tới những phương pháp cụ thể và kết quả đạt được; xuyên suốt theo 3 phần trong cuốn sách. Bạn cũng có thể tìm thấy những tài nguyên khác tại website của riêng cuốn sách này <http://gettingthingsdone.com/>.

Personal Kanban: Mapping Work | Navigating Life

Rất ít người biết đến cuốn sách này của Jim Benson và Tonianne DeMaria Barry; đa phần chưa biết tới sức mạnh của Kanban, một số dù thực hành Kanban trong nhóm hay những dự án cụ thể nhưng không nghĩ rằng việc áp dụng Kanban cho cuộc sống cá nhân lại hiệu quả. Phần 3 của cuốn sách này cũng dựa trên những tư tưởng được trình bày trong Personal Kanban; theo tôi là một phương pháp đơn giản, dễ thực hành nhưng mang lại hiệu quả lớn trong việc tối ưu hoá giá trị những công việc trong cuộc sống. Tất nhiên, bạn sẽ còn tìm thấy nhiều điều bổ ích hơn nữa về những nguyên lý đứng sau cũng như cách thực hiện Kanban cá nhân một cách hiệu quả thông qua cuốn sách.

Website của tác giả cùng nhiều tài nguyên hơn về Personal Kanban: <http://www.personalkanban.com/>

CHO NHÓM PHÁT TRIỂN PHẦN MỀM

Scrum guide

Mọi thành viên trong nhóm thực hành Scrum phải đọc *Scrum guide*, cuốn sách giáo khoa về Scrum, và thậm chí nên liên tục đọc lại để làm theo đúng chỉ dẫn Scrum – chính là yếu tố giúp nhóm có hiệu suất cao và tránh những vấn đề gặp phải do hiểu sai Scrum khiến nhóm thực hành ScrumBUT.

Bạn có thể tải Scrum guide tại địa chỉ <http://www.scrumguides.org/> với khoảng 30 ngôn ngữ, bao gồm tiếng Việt với bản dịch đầu tiên được thực hiện bởi nhóm Hanoi Scrum.

Scrum primer

Nếu *Scrum guide* được coi là sách giáo khoa, *Scrum primer* được coi là sách tham

khảo bổ sung trực tiếp; mặc dù rất ngắn gọn nhưng lại mang những chỉ dẫn thực tế và hướng thực hành nhiều hơn cho nhóm thực hành Scrum. Bạn có thể tải Scrum primer tại địa chỉ <http://www.scrumprimer.org/>, bản dịch đầu tiên cũng được thực hiện bởi nhóm Hanoi Scrum.

Scrum reference card

Tương tự *Scrum primer*, *Scrum reference card* cũng là một nguồn tài liệu bổ trợ trực tiếp, tuy rất ngắn gọn nhưng hữu ích, có tại <http://scrumreferencecard.com/>

Software in 30 Days: How Agile Managers Beat the Odds, Delight Their Customers, And Leave Competitors In the Dust

Cuốn sách hay nhất về Scrum, do chính tác giả của Scrum là Ken Schwaber và Jeff Sutherland chấp bút. Bằng hiểu biết sâu sắc, các tác giả đưa ra những vấn đề về lý thuyết cũng như thực hành nhằm thay đổi quan niệm rằng phát triển phần mềm là vấn đề phức tạp và tốn nhiều thời gian. Chúng ta hoàn toàn có cách làm khác, từ cách tiếp cận studio tới nhóm thực sự và mức doanh nghiệp lớn. Và *Software in 30 days* không chỉ phục vụ vấn đề tạo cảm hứng, đó là việc khả thi với những lý thuyết cụ thể.

Scrum and XP from the Trenches

Một cuốn sách rất hay của Henrik Kniberg, được xuất bản bởi InfoQ, được Hanoi Scrum dịch năm 2012 với tựa *Scrum và XP từ những chiến hào* là những chỉ dẫn thực hành tuyệt vời. Thông qua những ví dụ, case study cụ thể cho từng công việc, *Scrum and XP from the Trenches* đưa người đọc sống trong trải nghiệm của nhóm làm việc cùng Scrum và XP, qua việc áp dụng lý thuyết để giải quyết từng công việc rất cụ thể, từ việc lập kế hoạch, cộng tác... đến việc thực hiện Retrospective; thậm chí những điều nhỏ nhất như sử dụng bảng hay sắp xếp bàn ghế.

Điều hay nhất của cuốn sách là bạn có thể sống trong trải nghiệm của một nhóm thực hành Scrum và XP thực sự qua những vấn đề cụ thể, cách làm cụ thể; và luôn hiểu tại sao với lý thuyết chặt chẽ phía sau.

refactoring.com

Một lập trình viên tốt không thể bỏ qua website <http://refactoring.com/> cũng như cuốn sách *Refactoring: Improving the Design of Existing Code* của huyền thoại Martin Fowler. Như tôi đã đề cập trong Chương 5, code refactoring là một phần không thể thiếu trong những dự án phát triển phần mềm; bởi dù chúng ta có thiết kế ban đầu tốt tới đâu cũng khó có thể đáp ứng được nhu cầu thay đổi của

yêu cầu phần mềm. Website này giống như một cuốn từ điển, tổng hợp những pattern, những dấu hiệu của những đoạn code tồi, cũng như chỉ dẫn cụ thể để khiến chúng tốt hơn.

Clean Code collection

Uncle Bob (Robert C. Martin) có hai cuốn sách mà bạn rất nên đọc: *Clean Code: A Handbook of Agile Software Craftsmanship* và *The Clean Coder: A Code Conduct of Professional Programmer*. Được viết theo đúng phong cách Uncle Bob, rất ngắn gọn và trực tiếp đi vào vấn đề, đây thực sự là handbook, với những chỉ dẫn cụ thể vào từng vấn đề để tạo ra *clean code*; từ rất nhỏ như cách đặt tên biến, cách viết method,... đến lớn hơn như vấn đề kiến trúc, testing. *Clean code* là một phần không thể thiếu nhưng chưa đủ để hình thành *clean coder*; Uncle Bob cũng đưa ra những chỉ dẫn cụ thể cho những hành vi, tư duy rất cụ thể để hình thành *clean coder*, từ việc thực hành TDD, quản lý chất lượng... tới quản lý thời gian và loại bỏ áp lực.

The Art of Agile Development

Tác giả James Shore và Shane Warden là những người đã thực hành Agile từ rất sớm (1999); với bề dày kinh nghiệm đáng nể, những nội dung trong *The Art of Agile Development* vẫn còn nguyên giá trị đến ngày nay dù cuốn sách được xuất bản đã lâu (2007). Với cách tiếp cận và cấu trúc khoa học *The Art of Agile Development* cung cấp những góc nhìn từ việc thực hành Agile qua XP, lập kế hoạch, cộng tác... đến việc thực sự Agile (being Agile) trong phát triển phần mềm. Rất nhiều nội dung trong cuốn sách này vẫn được trích dẫn trong những khóa học, đào tạo Agile ngày nay.

The Pragmatic programmer: From Journeyman to Master

Có rất nhiều tranh cãi đằng sau tư tưởng và cách tiếp cận của David Thomas; song thật khó phủ nhận những tư tưởng và chỉ dẫn trong cuốn sách mang nguyên lý Agile: *pragmatic*. Có đôi khi, bạn sẽ thấy những chỉ dẫn, ví dụ trong cuốn sách không phù hợp với thời đại ngày nay (cuốn sách được xuất bản lần đầu năm 1999) nhưng quan trọng là những tư tưởng phía sau, thực hành những tư tưởng đó cần nhiều thời gian để thực sự trở thành master một cách bền vững. Dù không trực tiếp liên quan tới Agile, đây là một cuốn sách nên đọc với mọi lập trình viên.

Working Effectively with Legacy Code

Làm việc với legacy code thường là vấn đề đau đầu nhất trong mọi dự án phần

mềm; và thật may là chúng ta có thêm một cuốn sách tốt của Michael C. Feather. Cuốn sách tiếp cận rất trực tiếp, dạng handbook, với những vấn đề cụ thể, tinh huống cụ thể mà chúng ta có thể đối mặt trong bất cứ dự án phần mềm nào khi làm việc với legacy code, dù thiết kế ban đầu có ra sao. Và thật may mắn, Michael C. Feather đưa ra những phân tích cũng như model phù hợp, cùng cách tiếp cận *an toàn* bằng việc kết hợp giữa testing, code refactoring... để giải quyết bài toán. Tôi cần nhấn mạnh lại rằng, *an toàn* và *chất lượng* là vấn đề lớn nhất, nỗi sợ hãi lớn nhất của mọi nhóm phát triển phần mềm khi làm việc với legacy code.

CHO TỔ CHỨC

Team Next: A Foundation for Building Agile Innovation Teams

Một cuốn sách *phải đọc* của Erik Thoresen cho bất cứ ai muốn xây dựng nhóm *being Agile*. Với cách tiếp cận rất trực tiếp và sáng tạo, Erik Thoresen đưa ra những mục tiêu cụ thể và từng bước để đạt được mục tiêu trong việc xây dựng nhóm cũng như tổ chức Agile. Tôi không hẳn bị thuyết phục bởi tất cả nội dung trong cuốn sách; nhưng phải thừa nhận rằng những nội dung đó cũng gợi mở nhiều điều cần suy nghĩ và giúp nhóm không chỉ dừng lại ở thực hành Agile.

The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Business

Một cuốn sách *gây bão* của Eric Ries thì không thể không nằm trong danh sách *phải đọc*; được dịch sang tiếng Việt với tựa *Khởi nghiệp tinh gọn*. Chúng ta đều thừa nhận rằng, startup là nơi lý tưởng để những triết lý và kỹ thuật Agile được áp dụng và Lean Startup tiếp tục khẳng định điều đó bằng việc đi từ mô hình tới những kỹ thuật, công cụ quản trị, vận hành phù hợp với startup trong bối cảnh nền kinh tế đang thay đổi với tốc độ cao. Và điều tôi thích nhất, là cuốn sách này theo đúng tư tưởng Lean, ngắn gọn nhưng rất *chất*.

Bạn có thể tìm hiểu nhiều hơn tại website <http://theleanstartup.com/>.

Think one team: An Inspiring Fable and Practical Guide for Managers, Employees and Jelly Bean Lovers

Cuốn sách của Graham Winter không được nhiều người biết đến và không liên quan trực tiếp tới Agile, song rất nên đọc. Đúng như tên gọi, cuốn sách mang đến những chỉ dẫn thực hành cụ thể để xây dựng văn hoá *one team* nơi mọi thành phần chia sẻ chung tầm nhìn, bức tranh toàn cảnh và trách nhiệm, quyết định trong những công việc cụ thể.

Leading Change

Có lẽ tôi không cần phải nói nhiều về cuốn sách này, cuốn sách mà mọi nhà quản trị phải đọc, từ giáo sư John P. Kotter của đại học MIT; được dịch ra tiếng Việt với tựa *Dẫn dắt sự thay đổi*. Đưa Agile vào tổ chức luôn là một vấn đề phức tạp, đặc biệt với những doanh nghiệp Việt Nam; và cần sự chuyển đổi hợp lý. *Leading Change* không liên quan trực tiếp tới Agile, song nếu chúng ta muốn thay đổi sang tổ chức Agile, những nội dung trong cuốn sách thực sự tạo cảm hứng và chứa đựng những bài học rất đáng lưu tâm khi trong thực hành.

Lean Change Management: Innovative Practices for Managing Organizational Change

Một cuốn sách thực hành việc chuyển đổi qua Agile đúng nghĩa, của Janson Little. Cuốn sách cung cấp những chỉ dẫn, thực hành thông qua việc thực nghiệm nhằm tạo ra và quản lý sự thay đổi. Ngoài mô hình Kotter (được đề cập trong *Leading Change*), Janson Little còn sử dụng nhiều mô hình khác như ADKAR, 7S... cùng nhiều chỉ dẫn cụ thể; được tham khảo nhiều trong những khoá huấn luyện, tư vấn việc chuyển đổi tổ chức sang Agile.

PHỤ LỤC

THUẬT NGỮ

Acceptance Criteria	Tiêu chí chấp nhận để một công việc được coi là hoàn thành.
Acceptance Test Driven Development	Một kỹ thuật trong XP, cả nhóm cùng định nghĩa ra Accept Criteria cùng những ví dụ cụ thể và hình thành ra những test case cụ thể trước khi thực hiện việc phát triển; việc phát triển luôn được kiểm tra tính đúng đắn bởi những test case đã đặt.
Agile Release Train	Một artifact trong SAFe, tồn tại trong thời gian dài, được cộng tác bởi nhiều nhóm Agile nhằm liên tục chuyển giao những phần tăng trưởng.
Agile umbrella	Chiếc ô Agile - thường được sử dụng để chỉ sự phủ bóng của phương pháp luận Agile tới những phương pháp cụ thể như Scrum, XP...
Artifact	Tạo tác.
ATDD	Xem Acceptance Test Driven Development.
BDD	Xem Behavior Driven Development.
Behavior Driven Development	Kỹ thuật phát triển từ TDD và ATDD, nhằm hướng tới những giá trị kinh doanh cụ thể, thường được định nghĩa rõ ràng qua những kịch bản cụ thể để điều hướng chức năng tạo ra giá trị kinh doanh.
Blue-green deployment	Phương pháp triển khai hệ thống nhằm đảm bảo tính sẵn sàng, được thực hiện qua hai thành phần: green - đang hoạt động ổn định; và blue - đang được triển khai.
Branch strategy	Chiến lược phân nhánh trong việc quản lý source code.
Burn-down Chart	Biểu đồ đốt cháy; là một artifact của Scrum nhằm trực quan hóa tiến độ công việc được thực hiện và khả năng đạt được mục tiêu (Sprint Goal hoặc Release Goal).

Burn-up Chart	Giống với Burn-down Chart, song biểu đồ có hướng đi lên (thể hiện khối lượng công việc đã được hoàn thành) thay vì đi xuống (thể hiện khối lượng công việc còn tồn đọng) như Burn-down Chart.
CD	Xem Continuous Delivery và Continuous Deployment.
CI server	Xem Continuous Integration server.
Code refactoring	Hoạt động chỉnh sửa source code (thường khiến source code dễ đọc và có cấu trúc tốt hơn) nhưng không ảnh hưởng tới những hành vi (chức năng) hiện tại.
Code review	Hoạt động đánh giá source code.
Command and control	Phương thức quản lý được điều hướng bởi việc ra lệnh (command) và sự điều khiển (control) chặt chẽ.
Common Definition of Done	DoD chung; thường được sử dụng trong những nhóm mở rộng Agile như SAFe, LeSS... khi mỗi nhóm Agile có DoD riêng, là sự mở rộng của một định nghĩa DoD chung. Phương pháp nhằm liên tục chuyển giao những phần thay đổi, từ chức năng mới, bugfix, cấu hình...
Continuous Delivery	Phương pháp nhằm liên tục triển khai những phần thay đổi, từ chức năng mới, bugfix, cấu hình... tới môi trường production.
Continuous Deployment	Phương pháp nhằm liên tục đồng bộ những công việc được thực hiện trong nhóm và tích hợp source code, build... giúp nhanh chóng tìm ra xung đột.
Continuous Integration	Server thực hiện những tác vụ tích hợp liên tục như Jenkins, TeamCity,...
Continuous Integration server	Khoảng thời gian được nhóm chia sẻ chung, làm việc và cộng tác cùng nhau, ngoài khoảng thời gian làm việc cá nhân của mỗi thành viên
Core time	Xem nhóm liên chức năng.
Cross functional team	Tập hợp những phương pháp luận, nằm trong chiếc ô Agile; tập trung vào những nguyên lý đứng sau việc tổ chức phát triển dự án phần mềm hơn là quy trình hay kỹ thuật cụ thể.
Crystal	

Customer centric organization**Daily Scrum**

Tổ chức đặt khách hàng làm trung tâm. Mọi hành vi trong tổ chức được điều hướng bởi giá trị mang lại cho khách hàng.

Một sự kiện hàng ngày của Scrum nhằm đồng bộ công việc giữa những thành viên trong nhóm; thường được tổ chức qua việc mỗi thành viên trả lời 3 câu hỏi trong cuộc họp nhóm ngắn (dưới 15 phút).

**Database versioning
Definition of Done**

Phương pháp quản lý phiên bản cơ sở dữ liệu. Định nghĩa hoàn thành, là một Artifact trong Scrum; thường tồn tại dưới dạng một danh sách kiểm tra việc hoàn thành của một công việc. Definition of Done thường được định nghĩa chung cho mọi công việc và được thống nhất trong toàn nhóm Scrum.

Phần tăng trưởng, có thể chuyển giao được. Xem Increment.

Deliverable

Phương pháp luận nhấn mạnh vào việc cộng tác, trao đổi giữa những nhà phát triển (developer) và chuyên gia vận hành (operation) nhằm liên tục chuyển giao những phần tăng trưởng nhanh nhất thông qua những công cụ tự động hóa.

Xem Definition of Done.

DevOps

Người lái; thuật ngữ sử dụng trong Pair Programming, chỉ người viết source code.

**DoD
Driver**

Phương pháp nằm trong chiếc ô Agile, giải quyết những bài toán có thời gian và ngân sách cố định.

**Dynamic Systems
Development Method****Epic**

Một artifact lớn hơn User Story, không thể hoàn thành trong một hoặc một vài Sprint; Epic có thể liên dự án và có thể phân chia nhỏ thành nhiều User Story.

Extreme Programming

Phương pháp phát triển phần mềm nằm trong chiếc ô Agile, đặt trọng tâm vào kỹ thuật và chất lượng.

Feature Driven Development

Phương pháp phát triển phần mềm nằm trong chiếc ô Agile, giải quyết vấn đề phát triển song song những chức năng trong hệ thống phần mềm.

Forming	Giai đoạn thứ nhất (hình thành) trong mô hình <i>Các trạng thái phát triển của nhóm</i> của Tuckman.
Green build	Bản build thành công; thường được hiểu là việc tích hợp source code, build hệ thống không có lỗi và vượt qua mọi test case.
Increament	Phần tăng trưởng; một chức năng mới đảm bảo DoD và có thể thao tác bởi người dùng, sẵn sàng chuyển giao (vào cuối Sprint).
Innovation and Planning Iteration	Sự kiện trong SAFe, là khoảng thời gian lặp cố định dành cho việc lên kế hoạch, cải tiến và thay đổi.
Integrated Increment	Artifact trong Nexus, chỉ phần tăng trưởng được tạo ra bởi nhóm Nexus, được tích hợp từ nhiều nhóm Scrum.
Iterative development	Phương pháp phát triển phần mềm dựa trên những vòng lặp xác định, tạo ra phần tăng trưởng sau mỗi vòng lặp.
INVEST	Tính chất (nên có của User Story): Independent (độc lập), Negotiable (có thể thương lượng), Valuable (có giá trị), Estimable (có thể ước lượng), Small (nhỏ) và Testable (có thể kiểm thử).
Iteration	Vòng lặp trong phát triển phần mềm theo phương pháp luận Agile; là những khoảng thời gian lặp đi lặp lại (như Sprint trong Scrum).
Kanban	Phương pháp nhằm phản hồi tức thì với những thay đổi trong sản xuất và phát triển phần mềm.
Large-Scale Scrum	Framework mở rộng Agile, được xây dựng bởi Bas Vodde và Craig Larman.
Lead time	Thời gian thực hiện một công việc, từ khi nhận được yêu cầu tới khi hoàn thành.
Lean Software Development	Phương pháp phát triển phần mềm nằm trong chiếc ô Agile, đặt trọng tâm việc mang lại giá trị theo cách tinh gọn.
Learning organization	Tổ chức học tập. Tổ chức có đủ kỹ năng để tạo ra, thu nhận và chuyển giao kiến thức trong tổ chức, từ đó thay đổi hành vi để phù hợp với những kiến thức và quyết định mới.

Legacy code	Source code cũ; những phần source code khiến nhóm phát triển không hiểu hoặc không đủ tự tin để thay đổi.
LeSS Navigator	Xem Large-Scale Scrum. Hoa tiêu; thuật ngữ sử dụng trong Pair Programming, chỉ người theo dõi việc source code, có tầm nhìn định hướng việc viết source code.
Nexus	Framework mở rộng Agile, được xây dựng bởi Ken Schwaber.
Nexus Integration Team	Nhóm chịu trách nhiệm kết nối, huấn luyện việc thực hành Nexus và Scrum; giải quyết những vấn đề liên nhóm về mặt thực hành, kỹ thuật cũng như quy trình được sử dụng.
Nexus Sprint Backlog	Sprint Backlog của nhóm Nexus.
Nexus Sprint Goal	Sprint Goal của nhóm Nexus.
Nhóm liên chức năng	Nhóm tập hợp những thành viên có đầy đủ những kỹ năng cần thiết để hoàn thành một công việc, mục tiêu chung.
Nhóm tự tổ chức	Nhóm tự chọn quy trình, cách thức, kỹ thuật tốt nhất để thực hiện công việc và đạt được mục tiêu đề ra, không phụ thuộc hay chịu bất cứ ảnh hưởng hay quyết định nào của một người ngoài nhóm.
Norming	Giai đoạn thứ ba (ổn định) trong mô hình Các trạng thái phát triển của nhóm của Tuckman.
Open Space	Hoạt động cộng tác, thảo luận, chia sẻ kiến thức... và giải quyết vấn đề trong nhóm.
Pair programming	Lập trình cặp; một kỹ thuật trong XP, trong đó hai lập trình viên làm việc cùng nhau để viết ra một đoạn source code nhưng chia sẻ chung một máy tính và thiết bị ngoại vi.
Peer review	Hoạt động đánh giá source code chéo giữa những thành viên trong nhóm phát triển.
Performing	Giai đoạn thứ tư (hiệu suất) trong mô hình Các trạng thái phát triển của nhóm của Tuckman.
Portfolio	Mức thứ ba trong SAFe, gồm nhiều Strategic Theme.

Product Backlog	Artifact trong Scrum; chứa danh sách những công việc cần hoàn thành của sản phẩm hoặc dự án (thường được sắp xếp theo độ ưu tiên).
Product centric organization	Tổ chức đặt sản phẩm làm trung tâm. Mọi hành vi trong tổ chức được điều hướng bởi sản phẩm.
Product Owner	Vai trò trong Scrum; chịu trách nhiệm về tâm nhìn sản phẩm và chuyển hóa thành những yêu cầu cụ thể thông qua trao đổi với nhóm Scrum và những bên liên quan.
Program	Mức thứ hai trong SAFe, tích hợp công việc giữa những nhóm Agile.
Program Increment	Phần tăng trưởng ở mức Program trong SAFe.
Pull system	Hệ thống kéo. Tổ chức định hình bởi những công việc phải làm, các cá nhân và nhóm chủ động kéo công việc nhằm thực hiện mục tiêu đề ra.
Push system	Hệ thống đẩy. Tổ chức với những vai trò và quy trình được định nghĩa trước, công việc được giao cho một người cụ thể đảm nhiệm.
Red build	Bản build không thành công; thường được hiểu là việc tích hợp source code hoặc build hệ thống có lỗi hoặc không thể vượt qua được các test case.
Release Goal	Mục tiêu triển khai; được quản lý bởi Product Owner trong Scrum.
Releaseable	Phần tăng trưởng có thể chuyển giao được; xem Deliverable.
Requirement Area	Một thành phần trong LeSS Huge để mở rộng với nhóm Agile gồm hàng trăm nhân sự.
Retrospective	Sự kiện trong Scrum nhằm nhìn lại và thúc đẩy việc cải tiến quy trình.
Review	Sự kiện trong Scrum nhằm review những gì đã thực hiện được sau Sprint và nhận những phản hồi từ các bên liên quan.
SAFe	Xem Scaled Agile Framework.
Scaled Agile Framework	Framework mở rộng Agile, được xây dựng bởi Scaled Agile Inc.
Scaling Agile	Mở rộng Agile; phương pháp thực hành Agile cho những nhóm và tổ chức lớn.

Scrum	Framework phát triển phần mềm, nằm trong chiếc ô Agile, đặt trọng tâm vào sự tập trung và tương tác.
ScrumMaster	Vai trò trong Scrum; chịu trách nhiệm đảm bảo Scrum được thực hành đúng trong nhóm và tổ chức.
Scrum of Scrums	Phương pháp mở rộng Agile; hướng trọng tâm vào sự tổ chức và tương tác giữa các nhóm Scrum.
Scrumban	Phương pháp kết hợp giữa Scrum và Kanban nhằm tận dụng lợi thế tối đa giữa Scrum (tập trung, tương tác) và Kanban (phản ứng nhanh với thay đổi).
Self-organised team	Xem nhóm tự tổ chức.
Servant-leader	Lãnh đạo đầy tớ; chỉ người lãnh đạo thông qua việc làm đầy tớ - loại bỏ những trở ngại cho nhân viên.
Shipable	Phần tăng trưởng có thể chuyển giao được; xem Deliverable.
Collective ownership source code	Source code được sở hữu tập thể, có thể hiểu và thay đổi bởi mọi thành viên trong nhóm phát triển.
Sprint Backlog	Backlog cho Sprint, bao gồm những công việc cần hoàn thành trong Sprint để đạt được Sprint Goal.
Sprint Goal	Mục tiêu của Sprint; gồm phần tăng trưởng và những mục tiêu khác (hoạt động cải tiến) trong nhóm Scrum.
Sprint Planning One	Sự kiện trong LeSS, thực hiện việc lên kế hoạch cho nhóm LeSS bằng cách phân chia công việc tới những nhóm Scrum.
Sprint Planning Two	Sự kiện trong LeSS, thực hiện việc lên kế hoạch trong nhóm Scrum.
Starfish	Phương pháp sao biển nhằm thực hiện Retrospective, thông qua việc phân loại những hành động thành: More (thực hiện nhiều hơn); Less (giảm thiểu), Start (bắt đầu thực hiện), Stop (không thực hiện), Keep (tiếp tục thực hiện).

Storming	Giai đoạn thứ hai (bão tố) trong mô hình Các trạng thái phát triển của nhóm của Tuckman.
Story Point	Đơn vị đo lường nỗ lực cần thiết để hoàn thành User Story.
Strategic Theme	Theme ở mức cao, mang ý nghĩa điều hướng tới những giá trị, mục tiêu cụ chung của sản phẩm hơn là những giá trị từ chức năng cụ thể.
System Demo	Sự kiện trong SAFe, demo phần tăng trưởng ở mức Program.
TDD	Xem Test Driven Development.
Team	Nhóm.
Team Board	Bảng, được chia sẻ chung cho toàn nhóm thực hành Agile.
Team velocity	Tốc độ của nhóm; khối lượng (Story Point, giờ...) nhóm thực hiện được trong mỗi Sprint.
Test Driven Development	Phương pháp của XP, đảm bảo chất lượng việc phát triển thông qua định nghĩa trước yêu cầu qua những test case cụ thể.
Test First Development	Phương pháp của XP, đảm bảo chất lượng việc phát triển thông qua định nghĩa trước yêu cầu qua những test case cụ thể (trước khi thực hiện việc cài đặt).
Theme	Gồm nhiều User Story có liên quan.
Unit test	Phần kiểm thử nhỏ nhất trong phát triển phần mềm, kiểm thử một unit (thường là một hàm, phương thức..).
User Story	Câu chuyện người dùng; được sử dụng để mô tả yêu cầu trong Scrum.
Value Stream mapping	Ánh xạ luồng giá trị; phương pháp tìm ra (và ánh xạ) những điểm có giá trị trong luồng công việc.
Waterfall	Phương pháp phát triển phần mềm thác nước.

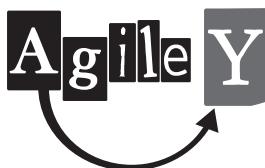
NHÀ XUẤT BẢN TRI THỨC

53 Nguyễn Du - Hà Nội

Tel: (844) 3945 4661; Fax: (844) 3845 4660

E-mail: lienhe@nxbtrithuc.com.vn

Nguyễn Hiển



Chịu trách nhiệm xuất bản: Chu Hảo

Biên tập: Nguyễn Bích Thuỷ

Thiết kế: Hà Tiên

In 1000 bản; khổ: 16 x 24 cm

Tại: Công ty Cổ phần In Hà Nội

Địa chỉ: Lô 6B, CN5, Cụm Công nghiệp Ngọc Hồi, Thanh Trì, Hà Nội

Giấy đăng ký KHXB số: 4254-2016/CXBIPH/2-48/TrT.

ISBN: 978-604-943-452-5

Quyết định xuất bản số: 72/QĐLK-NXBTrT

của Giám đốc NXB Tri thức ngày 08 tháng 12 năm 2016.

In xong và nộp lưu chiểu Quý 4 năm 2016



Agile Y là cuốn sách tiếng Việt về Agile phong phú nhất mà tôi từng đọc... Dành cho những ai đang loay hoay tìm kiếm phương pháp luận hay cách thức thực hành để nâng cao hiệu suất "làm phần mềm". Một món quà đặc biệt và cần thiết dành cho Agile Vietnam cũng như cộng đồng những người làm phần mềm ở Việt Nam.

Huỳnh Lê Tấn Tài

Chủ tịch, Agile Vietnam



NGUYỄN HIỂN

Agile

Cuốn sách này không có những chỉ dẫn kiểu "cẩm nang", nhưng hàm chứa những ý tưởng quan trọng cho hành trình tự học bất tận của những người làm nghề.

Hãy đọc một lèo hết quyển sách, dừng lại suy nghĩ để chiêm nghiệm, và lập một kế hoạch áp dụng những ý tưởng của cuốn sách. Bạn sẽ thấy những thay đổi tuyệt vời.

Dương Trọng Tấn

Giám đốc, Học viện Agile



Nguyễn Văn Hiển
CSM, CSP

🌐 <http://gurunh.com>
<https://linkedin.com/in/hiennvn>



Y

