# Chapter 7. What Is Agile Design?

Sau khi xem xét vòng đời phát triển phần mềm như tôi đã hiểu, tôi đã kết luận software documentation duy nhất thực sự có vẻ thỏa mãn các tiêu chí của một thiết kế kỹ thuật là danh sách source code. - Jack Reeves

Năm 1992, Jack Reeves đã viết một bài báo chuyên đề "Thiết kế phần mềm là gì?" Trên Tạp chí C ++. Trong bài viết này, Reeves lập luận rằng thiết kế của một hệ thống phần mềm được ghi lại chủ yếu bằng source code của nó, rằng các diagrams đại diện cho source code là phụ trợ cho thiết kế và không phải là bản thân thiết kế. Hóa ra, bài báo của Jack là điềm báo cho agile development

Trong các trang tiếp theo, chúng ta thường nói về "the design". Bạn không nên coi điều đó có nghĩa là một tập hợp các sơ đồ UML tách biệt với code. Một tập hợp các sơ đồ UML có thể đại diện cho các phần của thiết kế, nhưng các sơ đồ đó không phải là thiết kế. Thiết kế của một dự án phần mềm là một khái niệm trừu tượng. Nó phải làm với hình dạng và cấu trúc tổng thể của chương trình, cũng như hình dạng và cấu trúc chi tiết của từng mô-đun, lớp và phương thức. Thiết kế có thể được đại diện bởi nhiều phương tiện khác nhau, nhưng phương án cuối cùng của nó là source code. Cuối cùng, source code là thiết kế.

## Design Smells

Nếu bạn may mắn, bạn bắt đầu một dự án với một bức tranh rõ ràng về những gì bạn muốn hệ thống trở thành. Thiết kế của hệ thống là một hình ảnh quan trọng trong tâm trí của bạn. Nếu bạn vẫn còn may mắn hơn, sự rõ ràng của thiết kế đó làm cho nó được phát hành đầu tiên.

Nhưng sau đó, một cái gì đó đi sai. Phần mềm bắt đầu thối rữa như một miếng thịt xấu. Thời gian trôi qua, sự thối rữa tiếp tục. Xấu xí, lở loét và nhọt tích tụ trong code, làm cho nó ngày càng khó khăn hơn để duy trì. Cuối cùng, nỗ lực tuyệt đối cần thiết để làm cho ngay cả những thay đổi đơn giản nhất cũng trở nên quá tệ đến nỗi các nhà phát triển và quản lý tuyến đầu khóc vì thiết kế lại.

Thiết kế lại như vậy hiếm khi thành công. Mặc dù các nhà thiết kế bắt đầu với ý định tốt, họ thấy rằng họ đang bắn vào một mục tiêu đang di chuyển. Hệ thống cũ tiếp tục phát triển và thay đổi, và thiết kế mới phải theo kịp. Các mụn cóc và vết loét tích tụ trong thiết kế mới trước khi nó được phát hành lần đầu tiên.

### Design SmellsThe Odors of Rotting Software

Bạn biết rằng phần mềm đang mục nát khi nó bắt đầu thể hiện bất kỳ mùi nào sau đây.

* Rigidity
* Fragility
* Immobility
* Viscosity
* Needless complexity
* Needless repetition
* Opacity

### Rigidity

Rigidity (Độ cứng ) là xu hướng phần mềm khó thay đổi, ngay cả trong những cách đơn giản. Một thiết kế cứng nhắc nếu một thay đổi duy nhất gây ra một loạt các thay đổi tiếp theo trong các mô-đun phụ thuộc. Càng nhiều mô-đun phải thay đổi, thiết kế càng cứng nhắc.

Hầu hết các nhà phát triển đã phải đối mặt với tình huống này bằng cách này hay cách khác. Họ được yêu cầu thực hiện những gì dường như là một thay đổi đơn giản. Họ xem xét sự thay đổi và ước tính hợp lý công việc cần thiết. Nhưng sau đó, khi họ thực hiện thay đổi, họ thấy rằng có những hậu quả không lường trước được đối với sự thay đổi. Các nhà phát triển thấy mình theo đuổi sự thay đổi phần lớn code, sửa đổi nhiều mô-đun hơn so với ước tính ban đầu và phát hiện ra thread sau các thread thay đổi khác mà họ phải nhớ thực hiện. Cuối cùng, những thay đổi mất nhiều thời gian hơn so với ước tính ban đầu. Khi được hỏi tại sao ước tính của họ quá kém, họ lặp lại lời than thở của các nhà phát triển phần mềm truyền thống: "Nó phức tạp hơn tôi nghĩ rất nhiều!"

### Fragility

Tính mong manh là xu hướng của một chương trình bị phá vỡ ở nhiều nơi khi một thay đổi duy nhất được thực hiện. Thông thường, các vấn đề mới là trong các khu vực không có mối quan hệ với khu vực đã được thay đổi. Khắc phục những vấn đề đó dẫn đến nhiều vấn đề hơn nữa và nhóm phát triển bắt đầu giống như một con chó đuổi theo đuôi của nó.

Khi độ mong manh của mô-đun tăng lên, khả năng thay đổi sẽ đưa ra các vấn đề bất ngờ tiếp cận sự chắc chắn. Điều này có vẻ vô lý, nhưng các mô-đun như vậy hoàn toàn không hiếm.

Đây là những mô-đun liên tục cần sửa chữa, những mô-đun không bao giờ nằm ngoài danh sách lỗi.

Các mô-đun này là những mô-đun mà các nhà phát triển biết cần phải được thiết kế lại, nhưng không ai muốn đối mặt với hậu quả của việc thiết kế lại chúng. Những mô-đun này là những mô-đun trở nên tồi tệ hơn khi bạn sửa chúng.

### Immobility

Một thiết kế là “bất động” khi nó chứa các phần có thể hữu ích trong các hệ thống khác, nhưng nỗ lực và rủi ro liên quan đến việc tách các phần đó khỏi hệ thống ban đầu là quá lớn. Đây là một điều đáng tiếc nhưng rất phổ biến.

### Viscosity

Độ nhớt có hai dạng: độ nhớt của phần mềm và độ nhớt của môi trường. Khi phải đối mặt với một sự thay đổi, các nhà phát triển thường tìm thấy nhiều hơn một cách để thực hiện thay đổi đó. Một số cách bảo tồn thiết kế; những người khác thì không (tức là họ là hack). Khi các phương pháp bảo quản thiết kế khó sử dụng hơn các hack, độ nhớt của thiết kế cao. Thật dễ để làm điều sai nhưng khó làm điều đúng. Chúng tôi muốn thiết kế phần mềm của chúng tôi sao cho những thay đổi bảo tồn thiết kế dễ thực hiện.

Độ nhớt của môi trường xuất hiện khi môi trường phát triển chậm và không hiệu quả. Ví dụ: nếu thời gian biên dịch rất dài, các nhà phát triển sẽ bị cám dỗ thực hiện các thay đổi không bắt buộc biên dịch lại lớn, mặc dù những thay đổi đó không bảo toàn thiết kế. Nếu hệ thống kiểm soát source code yêu cầu hàng giờ để kiểm tra chỉ một vài tệp, các nhà phát triển sẽ sẵn sàng thực hiện các thay đổi yêu cầu càng ít đăng ký càng tốt, bất kể thiết kế có được bảo tồn hay không.

Trong cả hai trường hợp, một dự án nhớt là một trong đó thiết kế của phần mềm rất khó bảo tồn. Chúng tôi muốn tạo ra các hệ thống và môi trường dự án giúp dễ dàng bảo quản và cải thiện thiết kế.

### Needless Complexity

Một thiết kế có mùi phức tạp không cần thiết khi nó chứa các yếu tố hiện không hữu ích. Điều này thường xảy ra khi các nhà phát triển dự đoán các thay đổi đối với các yêu cầu và đưa vào phần mềm để đối phó với những thay đổi có thể xảy ra sau này. Lúc đầu, điều này có vẻ như là một điều tốt. Rốt cuộc, chuẩn bị cho những thay đổi trong tương lai sẽ giữ cho code của chúng tôi linh hoạt và ngăn chặn những thay đổi về sau.

Thật không may, hiệu quả thường chỉ là ngược lại. Bằng cách chuẩn bị cho nhiều tình huống, thiết kế trở nên ngổn ngang với các cấu trúc không bao giờ được sử dụng. Một số trong những chuẩn bị có thể được đền đáp, nhưng nhiều người khác thì không. Trong khi đó, thiết kế mang trọng lượng của các yếu tố thiết kế không sử dụng này. Điều này làm cho phần mềm phức tạp và khó hiểu.

### Needless Repetition

Cắt và dán có thể là các hoạt động chỉnh sửa văn bản hữu ích, nhưng chúng có thể là các hoạt động chỉnh sửa code thảm họa. Tất cả quá thường xuyên, các hệ thống phần mềm được xây dựng trên hàng chục hoặc hàng trăm yếu tố code lặp đi lặp lại. Nó xảy ra như thế này: Ralph cần phải viết một số code. Anh ta nhìn xung quanh trong các phần khác của code nơi anh ta nghĩ rằng nó phù hợp và tìm thấy một đoạn code phù hợp. Anh ta cắt và dán code đó vào mô-đun của mình và thực hiện các sửa đổi phù hợp.

Khi cùng một code xuất hiện lặp đi lặp lại, trong các hình thức hơi khác nhau, các nhà phát triển đang thiếu một sự trừu tượng hóa. Tìm tất cả sự lặp lại và loại bỏ nó với một sự trừu tượng thích hợp có thể không cao trong danh sách ưu tiên của họ, nhưng nó sẽ đi một chặng đường dài để làm cho hệ thống dễ hiểu và bảo trì hơn.

Khi có code rác trong hệ thống, công việc thay đổi hệ thống có thể trở nên khó khăn. Lỗi được tìm thấy trong một đơn vị lặp lại như vậy phải được sửa chữa trong mỗi lần lặp lại. Tuy nhiên, vì mỗi lần lặp lại hơi khác nhau, nên cách khắc phục không phải lúc nào cũng giống nhau.

### Opacity

Độ mờ là xu hướng của một mô-đun là khó hiểu. Code có thể được viết một cách rõ ràng và dễ hiểu, hoặc nó có thể được viết theo cách mờ đục và phức tạp. Code phát triển theo thời gian có xu hướng ngày càng mờ đục theo tuổi tác. Một nỗ lực liên tục để giữ cho code rõ ràng và dễ hiểu là cần thiết để giữ độ mờ ở mức tối thiểu.

Khi các nhà phát triển lần đầu tiên viết một mô-đun, mã có thể rõ ràng đối với họ. Rốt cuộc, họ đã đắm mình trong đó và rất hiểu nó. Sau đó, sau khi sự gắn bó đã biến mất, họ có thể quay lại mô-đun đó và tự hỏi làm thế nào họ có thể viết bất cứ điều gì khủng khiếp như vậy. Để ngăn chặn điều này, các nhà phát triển cần đặt mình vào vị trí của độc giả và nỗ lực phối hợp để cấu trúc lại mã của họ để độc giả của họ có thể hiểu được.

## Why Software Rots

Trong môi trường không phải agile, các thiết kế xuống cấp vì các yêu cầu thay đổi theo cách mà thiết kế ban đầu không lường trước được. Thông thường, những thay đổi này cần được thực hiện nhanh chóng và có thể được thực hiện bởi các nhà phát triển không quen với triết lý thiết kế ban đầu. Vì vậy, mặc dù sự thay đổi trong thiết kế hoạt động, nhưng bằng cách nào đó nó vi phạm thiết kế ban đầu. Từng chút một, khi những thay đổi tiếp tục, những vi phạm này tích lũy cho đến khi bệnh ác tính bắt đầu.

Tuy nhiên, chúng ta không thể đổ lỗi cho sự trôi dạt của các yêu cầu đối với sự xuống cấp của thiết kế. Chúng tôi, là nhà phát triển phần mềm, biết rõ rằng các yêu cầu thay đổi. Thật vậy, hầu hết chúng ta nhận ra rằng các yêu cầu là yếu tố dễ bay hơi nhất trong dự án. Nếu các thiết kế của chúng tôi không thành công do sự thay đổi yêu cầu liên tục, thì đó là thiết kế và thực hiện của chúng tôi có lỗi. Chúng ta phải bằng cách nào đó tìm cách làm cho các thiết kế của chúng ta trở nên linh hoạt trước những thay đổi như vậy và thực hành bảo vệ chúng khỏi mục nát.

Một agile team phát triển mạnh về sự thay đổi. Nhóm đầu tư ít lên phía trước và do đó không được trao cho một thiết kế ban đầu cũ. Thay vào đó, nhóm giữ cho thiết kế của hệ thống sạch sẽ và đơn giản nhất có thể và sao lưu nó với rất nhiều unit tests và acceptance tests. Điều này giữ cho thiết kế linh hoạt và dễ dàng thay đổi. Nhóm tận dụng sự linh hoạt đó để liên tục cải tiến thiết kế; do đó, mỗi lần lặp kết thúc với một hệ thống có thiết kế phù hợp như nó có thể dành cho các yêu cầu trong lần lặp đó.

## The Copy Program

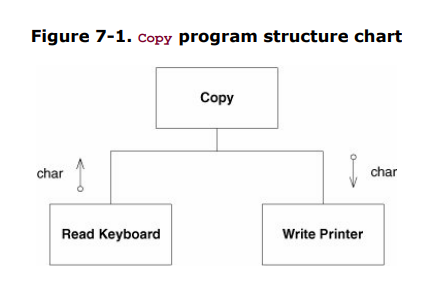
### A Familiar Scenario

Giả sử sếp đến gặp bạn vào sáng sớm thứ Hai và yêu cầu bạn viết chương trình sao chép các ký tự từ bàn phím sang máy in. Thực hiện tính toán nhanh chóng trong đầu, bạn kết luận rằng đây sẽ là ít hơn mười dòng mã. Thời gian thiết kế và viết code nên ít hơn 1 giờ. Với group meetings, quality education meetings, daily group progress meetings là ba cuộc khủng hoảng hiện tại trong lĩnh vực này, chương trình này phải khiến bạn mất khoảng một tuần để hoàn thành nếu bạn ở lại sau nhiều giờ. Tuy nhiên, bạn luôn nhân ước tính của mình với 3.

"Ba tuần," bạn nói với sếp của bạn. Anh ta quấy rối và bỏ đi, để lại cho bạn nhiệm vụ của bạn.

#### The initial design

Bạn có một chút thời gian ngay bây giờ trước khi cuộc họp đánh giá quá trình (process review meeting ) bắt đầu, vì vậy bạn quyết định vạch ra một thiết kế cho chương trình. Sử dụng thiết kế có cấu trúc, bạn đưa ra biểu đồ cấu trúc trong Hình 7-1.



Có ba mô-đun, hoặc chương trình con, trong ứng dụng. Mô-đun Copy gọi hai cái kia. Chương trình Copy tìm nạp các ký tự từ mô-đun Read Keyboard và định tuyến chúng đến mô-đun Write Printer.

Bạn nhìn vào thiết kế của bạn và thấy rằng nó là tốt. Bạn mỉm cười và sau đó rời khỏi văn phòng của bạn để đi review đó. Ít nhất bạn sẽ có thể ngủ một chút.

Vào thứ ba, bạn đến sớm một chút để có thể hoàn thành chương trình Copy. Thật không may, một trong những cuộc khủng hoảng trong lĩnh vực này đã ấm lên chỉ sau một đêm, và bạn phải đến phòng thí nghiệm và giúp gỡ lỗi một vấn đề. Vào giờ nghỉ trưa, cuối cùng bạn mất lúc 3 giờ chiều, bạn quản lý để nhập code cho chương trình Copy.



Vào thứ Tư, bạn đến sớm một lần nữa và lần này dường như không có gì là không ổn. Vì vậy, bạn bắt đầu biên dịch chương trình Copy. Lần đầu tiên nó biên dịch không có lỗi! Đó cũng là một điều tốt, bởi vì sếp của bạn gọi bạn vào một cuộc họp đột xuất về nhu cầu bảo tồn mực máy in laser.

Vào thứ năm, sau khi dành 4 giờ trên điện thoại, một kỹ thuật viên dịch vụ ở Rocky Mount, Bắc Carolina, thông qua các lệnh gỡ lỗi và ghi nhật ký lỗi từ một trong những thành phần khó hiểu hơn của hệ thống, bạn lấy Hoho và sau đó kiểm tra chường trình Copy. Nó hoạt động, lần đầu tiên! Quá tốt. Bởi vì sinh viên hợp tác mới của bạn vừa xóa master source code khỏi serve và bạn phải đi tìm các băng sao lưu mới nhất và khôi phục nó. Tất nhiên, bản sao lưu đầy đủ cuối cùng đã được thực hiện 3 tháng trước và bạn có 94 bản sao lưu gia tăng để khôi phục.

Thứ sáu là hoàn toàn không có việc. Cũng tốt, vì phải mất cả ngày để chương trình Copy được tải thành công vào source code control system của bạn.

Tất nhiên, chương trình là một thành công hoành tráng và được triển khai trong toàn công ty của bạn. Danh tiếng của bạn như là một lập trình viên ace một lần nữa được khẳng định, và bạn đắm chìm trong vinh quang của những thành tựu của bạn. Nếu may mắn, bạn thực sự có thể sản xuất 30 dòng code trong năm nay!

#### The requirements they are a'changin

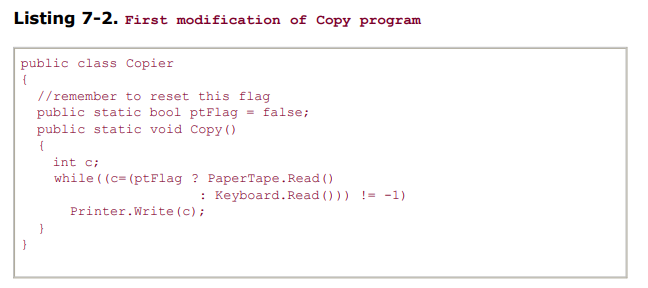
Vài tháng sau, sếp của bạn đến gặp bạn và nói rằng chương trình Copy cũng có thể đọc được từ paper tape reader. Bạn nghiến răng và đảo mắt. Bạn tự hỏi tại sao mọi người luôn thay đổi các yêu cầu. Chương trình của bạn không được thiết kế cho paper tape reader! Bạn cảnh báo sếp rằng những thay đổi như thế này sẽ phá hủy sự thanh lịch trong thiết kế của bạn. Tuy nhiên, ông chủ của bạn rất kiên quyết, nói rằng người dùng thực sự cần phải đọc các ký tự từ đầu đọc băng giấy theo thời gian.

Vì vậy, bạn thở dài và lập kế hoạch sửa đổi của bạn. Bạn muốn thêm một đối số Boolean vào chức năng Copy. Nếu TRue, bạn sẽ đọc từ paper tape reader; nếu sai, bạn sẽ đọc từ bàn phím như trước.

Thật không may, bây giờ rất nhiều chương trình khác sử dụng chương trình Copy mà bạn không thể thay đổi interface. Thay đổi interface sẽ gây ra vài tuần biên dịch lại và kiểm tra lại. Các kỹ sư kiểm tra hệ thống một mình sẽ làm bạn lo lắng, chưa kể bảy người trong configuration control group. Và process police sẽ có một ngày thực địa, buộc tất cả các loại đánh giá mã cho mọi mô-đun gọi là Copy!

Không, thay đổi giao diện là không được. Nhưng sau đó, làm thế nào bạn có thể cho chương trình Copy biết rằng nó phải đọc từ paper tape reader? Tất nhiên! Bạn sẽ sử dụng global! Bạn cũng sẽ sử dụng tính năng tốt nhất và hữu ích nhất của ngôn ngữ C, toán tử ?:!

kết quả.



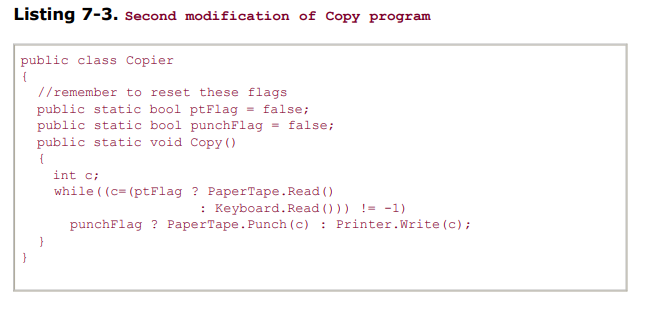
Copy đặt ptFlag thành true khi có người nhập bằng paper tape reader. Sau đó, họ có thể gọi Copy, và nó sẽ vui vẻ đọc từ paper tape reader. Khi Copy return giá trị, ptFlag được đặt lại; mặt khác, lần gọi tiếp theo có thể đọc nhầm từ đầu đọc băng giấy thay vì từ bàn phím. Để nhắc nhở các lập trình viên về nhiệm vụ của họ để thiết lập lại cờ này, bạn đã thêm một nhận xét thích hợp.

Một lần nữa, bạn phát hành phần mềm của bạn để được hoan nghênh. Nó thậm chí còn thành công hơn trước và hàng loạt lập trình viên háo hức đang chờ đợi một cơ hội để sử dụng nó. Cuộc sống thật tốt.

#### Give 'em an inch

Vài tuần sau, ông chủ của bạn vẫn là ông chủ của bạn mặc dù có ba lần tổ chức lại toàn công ty trong nhiều tháng và ông ấy báo bạn rằng khách hàng đôi khi muốn chương trình Copy xuất ra paper tape punch. Khách hàng! Họ luôn làm hỏng thiết kế của bạn. Viết phần mềm sẽ dễ dàng hơn rất nhiều nếu nó không dành cho khách hàng. Bạn nói với sếp của bạn rằng những thay đổi không ngừng này có ảnh hưởng tiêu cực sâu sắc đến sự thanh lịch trong thiết kế của bạn, cảnh báo rằng nếu thay đổi tiếp tục với tốc độ khủng khiếp này, phần mềm sẽ không thể duy trì trước khi kết thúc năm. Sếp của bạn gật đầu có chủ ý và sau đó nói với bạn để thực hiện thay đổi.

Sự thay đổi thiết kế này tương tự như trước đây. Tất cả chúng ta cần gobal và khác toán tử ?: Bạn đặc biệt tự hào về thực tế rằng bạn đã nhớ để thay đổi nhận xét. Tuy nhiên, bạn lo lắng rằng cấu trúc chương trình của bạn đang bắt đầu lật đổ. Bất kỳ thay đổi nào nữa đối với thiết bị đầu vào chắc chắn sẽ buộc bạn phải cơ cấu lại hoàn toàn vòng lặp while có điều kiện. Có lẽ đã đến lúc phủi bụi sơ yếu lý lịch của bạn.



### Expect changes

Tôi sẽ để lại cho bạn để xác định mức độ cường điệu trước đó là bao nhiêu. Điểm chính của câu chuyện là cho thấy cách thiết kế chương trình có thể nhanh chóng xuống cấp trước sự thay đổi.

Thiết kế ban đầu của chương trình Copy rất đơn giản và thanh lịch. Tuy nhiên, chỉ sau hai thay đổi, nó đã bắt đầu cho thấy những dấu hiệu của sự cứng nhắc, mong manh, bất động, phức tạp, dư thừa và mờ đục. Xu hướng này chắc chắn sẽ tiếp tục, và chương trình sẽ trở thành một mớ hỗn độn.

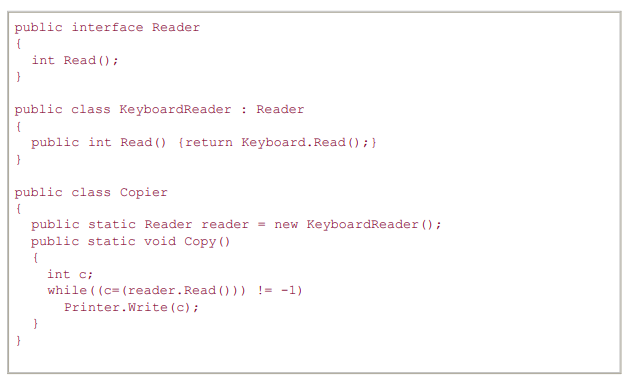
Chúng tôi có thể ngồi lại và đổ lỗi cho điều này về những thay đổi. Chúng tôi có thể phàn nàn rằng chương trình được thiết kế tốt cho thông số ban đầu và những thay đổi tiếp theo đối với thông số kỹ thuật khiến thiết kế xuống cấp. Tuy nhiên, điều này bỏ qua một trong những sự thật nổi bật nhất trong phát triển phần mềm: Yêu cầu luôn thay đổi!

Hãy nhớ rằng, những thứ dễ bay hơi nhất trong hầu hết các dự án phần mềm là các yêu cầu. Các yêu cầu liên tục trong tình trạng thông lượng. Đây là một thực tế mà chúng tôi, là nhà phát triển, phải chấp nhận!

Chúng ta sống trong một thế giới thay đổi các yêu cầu và công việc của chúng ta là đảm bảo rằng phần mềm của chúng ta có thể tồn tại trong những thay đổi đó. Nếu thiết kế phần mềm của chúng tôi xuống cấp vì các yêu cầu đã thay đổi, chúng tôi sẽ không nhanh nhẹn.

## Agile Design of the Copy Program

Một nhóm agile có thể bắt đầu chính xác theo cùng một cách, với mã trong Liệt kê 7-1. Khi ông chủ yêu cầu làm cho chương trình đọc từ đầu đọc băng giấy, các nhà phát triển sẽ phản hồi bằng cách thay đổi thiết kế để trở nên linh hoạt theo kiểu đó của sự thay đổi. Kết quả có thể giống như Liệt kê 7-4.



Thay vì cố gắng vá thiết kế để làm cho yêu cầu mới hoạt động, nhóm đã nắm bắt cơ hội để cải tiến thiết kế để nó sẽ linh hoạt với loại thay đổi đó trong tương lai. Từ giờ trở đi, bất cứ khi nào ông chủ yêu cầu một loại thiết bị đầu vào mới, nhóm sẽ có thể chỉnh sửa theo cách không gây ra sự xuống cấp cho chương trình

Nhóm đã tuân theo Open/Closed Principle (OCP), mà chúng tôi mô tả trong Chương 9. Nguyên tắc này hướng dẫn chúng tôi thiết kế các mô-đun để chúng có thể được mở rộng mà không cần sửa đổi. Đó chính xác là những gì nhóm đã làm. Mỗi thiết bị đầu vào mới mà ông chủ yêu cầu có thể được cung cấp mà không cần sửa đổi chương trình Copy

Tuy nhiên, lưu ý rằng khi lần đầu tiên thiết kế mô-đun, nhóm đã không cố gắng dự đoán chương trình sẽ thay đổi như thế nào. Thay vào đó, nhóm đã viết mô-đun theo cách đơn giản nhất có thể. Chỉ đến khi các yêu cầu cuối cùng thay đổi, nhóm mới thay đổi thiết kế mô-đun để có khả năng phục hồi với loại thay đổi đó

Người ta có thể lập luận rằng nhóm chỉ làm một nửa công việc. Mặc dù các developer đã tự bảo vệ mình khỏi các thiết bị đầu vào khác nhau, nhưng họ cũng có thể tự bảo vệ mình khỏi các thiết bị đầu ra khác nhau. Tuy nhiên, nhóm thực sự không biết liệu các thiết bị đầu ra có bao giờ thay đổi hay không. Để thêm sự bảo vệ bổ sung bây giờ sẽ không đáp ứng được mục địc hiện tại. Rõ ràng là nếu cần sự bảo vệ như vậy, nó sẽ dễ dàng được thêm vào sau này. Vì vậy, thực sự không có lý do để thêm nó ngay bây giờ.

## Following agile practices

Các nhà phát triển agile trong ví dụ của chúng tôi đã xây dựng một lớp trừu tượng để bảo vệ khỏi những thay đổi đối với dữ liệu đầu vào. Làm thế nào mà họ biết làm thế nào để làm điều đó? Câu trả lời nằm ở một trong những nguyên lý cơ bản của thiết kế hướng đối tượng.

Thiết kế ban đầu của chương trình Sao chép không linh hoạt do hướng phụ thuộc của nó. Nhìn lại vào Hình 7-1. Lưu ý rằng mô-đun Copy phụ thuộc trực tiếp vào KeyboardReader và the PrinterWrite. Mô-đun Copy là mô-đun cấp cao trong ứng dụng này. Nó đặt chính sách của ứng dụng. Nó biết cách sao chép ký tự. Thật không may, nó cũng đã được thực hiện phụ thuộc vào các chi tiết cấp thấp của bàn phím và máy in. Do đó, khi các chi tiết cấp thấp thay đổi, nó cũng bị ảnh hưởng.

Khi tính không linh hoạt được phát hiện, các agile developer đã biết rằng sự phụ thuộc từ mô-đun Copy vào thiết bị đầu vào cần phải được đảo ngược, sử dụng Dependency Inversion Principle (DIP) trong Chương 11, do đó Sao chép sẽ không còn phụ thuộc vào thiết bị đầu vào. Sau đó, họ đã sử dụng STRATEGY pattern , được thảo luận trong Chương 22, để tạo ra sự đảo ngược mong muốn.

Vì vậy, trong ngắn hạn, các agile developer biết phải làm gì vì họ đã làm theo các bước này.

* Họ đã phát hiện ra vấn đề bằng cách làm theo các agile practice.
* Họ chẩn đoán vấn đề bằng cách áp dụng các nguyên tắc thiết kế.
* Họ đã giải quyết vấn đề bằng cách áp dụng một mẫu thiết kế phù hợp.

Sự tương tác giữa ba khía cạnh phát triển phần mềm này là hành động của thiết kế.

## Keeping the design as good as it can be

Các nhà phát triển Agile giữ cho thiết kế phù hợp và sạch sẽ nhất có thể. Đây không phải là một cam kết hỗn loạn hoặc dự kiến. Các nhà phát triển Agile không "dọn dẹp" thiết kế mỗi vài tuần. Thay vào đó, họ giữ cho phần mềm sạch sẽ, đơn giản và biểu cảm nhất có thể mỗi ngày, mỗi giờ và mỗi phút. Họ không bao giờ nói, "Chúng tôi sẽ quay lại và sửa nó sau." Họ không bao giờ để sự thối rữa bắt đầu.

Thái độ mà các agile developer đối với việc thiết kế phần mềm cũng giống như thái độ của các bác sĩ phẫu thuật đối với quy trình vô trùng. Thủ tục vô trùng là những gì làm cho phẫu thuật xảy ra. Không có nó, nguy cơ nhiễm trùng sẽ quá cao để có thể chịu đựng được. Các nhà phát triển Agile cảm thấy như vậy về thiết kế của họ. Nguy cơ để cho ngay cả một chút thối nhỏ nhất bắt đầu là quá cao để chịu đựng

## Conclusion

Vậy, thiết kế nhanh là gì? Thiết kế Agile là một quá trình, không phải là một sự kiện. Đây là ứng dụng liên tục của các principles, patterns, và practices để cải thiện cấu trúc và khả năng đọc của phần mềm. Nó là sự cống hiến để giữ cho thiết kế của hệ thống đơn giản, sạch sẽ và rõ ràng có thể mọi lúc.

Trong các chương tiếp theo, chúng ta sẽ nghiên cứu các nguyên tắc và mô hình thiết kế phần mềm. Khi bạn đọc, hãy nhớ rằng một agile developer không áp dụng các nguyên tắc và mẫu đó cho một thiết kế lớn, phía trước. Thay vào đó, chúng được áp dụng từ lặp đi lặp lại đến lặp đi lặp lại trong nỗ lực giữ mã và thiết kế mà nó thể hiện, sạch sẽ.

# Chapter 8. The Single-Responsibility Principle (SRP)

Không ai khác ngoài chính Đức Phật phải có trách nhiệm đưa ra những bí mật huyền bí. . .

Nguyên tắc này đã được mô tả trong tác phẩm của Tom DeMarco [1] và Meilir Page-Jones. [2] Họ gọi đó là sự gắn kết, mà họ định nghĩa là sự liên quan chức năng của các yếu tố của một mô-đun. Trong chương này, chúng tôi sửa đổi ý nghĩa đó một chút và liên kết sự gắn kết với các lực gây ra một mô-đun, hoặc một lớp, để thay đổi.

The Single-Responsibility Principle

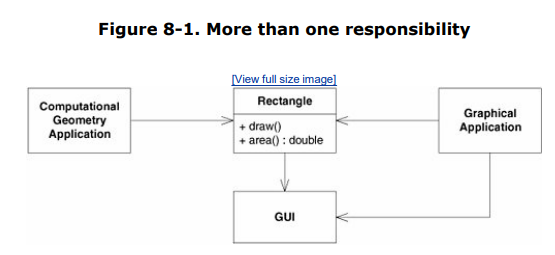
A class should have only one reason to change.

Hãy xem xét trò chơi bowling từ Chương 6. Trong phần lớn sự phát triển của nó, lớp Game đã xử lý hai trách nhiệm riêng biệt: theo dõi khung hiện tại và tính điểm. Cuối cùng, RCM và RSK đã tách hai trách nhiệm này thành hai lớp. Game giữ trách nhiệm theo dõi các khung và Scorer có trách nhiệm tính điểm.

Tại sao điều quan trọng là tách hai trách nhiệm này thành các lớp riêng biệt? Lý do là mỗi trách nhiệm là một trục thay đổi. Khi các yêu cầu thay đổi, sự thay đổi đó sẽ được thể hiện thông qua sự thay đổi trách nhiệm giữa các lớp. Nếu một lớp nhận nhiều hơn một trách nhiệm, thì lớp đó sẽ có nhiều hơn một lý do để thay đổi

Nếu một lớp có nhiều hơn một trách nhiệm, các trách nhiệm sẽ được kết hợp. Thay đổi một trách nhiệm có thể làm giảm hoặc ức chế khả năng gặp gỡ của các lớp khác. Loại khớp nối này dẫn đến các thiết kế mỏng manh bị phá vỡ theo những cách bất ngờ khi thay đổi.

Ví dụ, hãy xem xét thiết kế trong Hình 8-1. Lớp hình chữ nhật có hai phương thức được hiển thị. Một là vẽ hình chữ nhật trên màn hình, và hình kia tính diện tích hình chữ nhật.



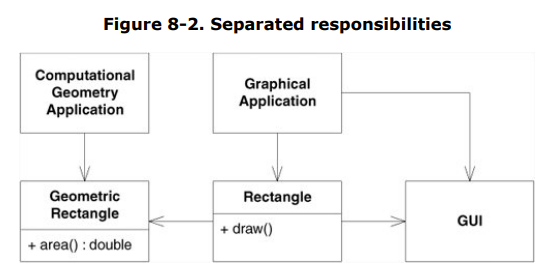
Hai ứng dụng khác nhau sử dụng lớp Rectangle. Một ứng dụng không tính toán hình học. Sử dụng Rectangle để giúp nó với các phép toán học của các hình dạng hình học nhưng không bao giờ vẽ hình chữ nhật trên màn hình. Ứng dụng khác có bản chất đồ họa và cũng có thể thực hiện một số hình học tính toán, nhưng nó chắc chắn vẽ hình chữ nhật trên màn hình.

Thiết kế này vi phạm SRP. Lớp Rectangle có hai trách nhiệm. Trách nhiệm đầu tiên là cung cấp một mô hình toán học về hình học của hình chữ nhật. Trách nhiệm thứ hai là kết xuất hình chữ nhật trên GUI.

Việc vi phạm SRP gây ra một số vấn đề khó chịu. Đầu tiên, chúng ta phải bao gồm GUI trong computational geometry application. Trong .NET, việc lắp ráp GUI sẽ phải được xây dựng và triển khai với computational geometry application.

Thứ hai, nếu một sự thay đổi đối với GraphicalApplication khiến cho Rectange thay đổi vì một số lý do, sự thay đổi đó có thể buộc chúng ta phải xây dựng lại, kiểm tra lại và triển khai lại ComputationalGeometryApplication. Nếu chúng ta quên làm điều này, ứng dụng đó có thể bị hỏng theo những cách không thể đoán trước.

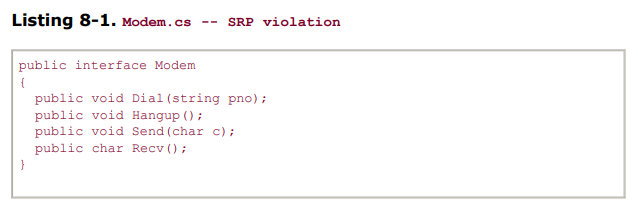
Một thiết kế tốt hơn là tách hai trách nhiệm thành hai lớp hoàn toàn khác nhau, như trong Hình 8-2. Thiết kế này di chuyển các phần tính toán của Rectangle vào lớp GeometricRectangle. Bây giờ các thay đổi được thực hiện theo cách hiển thị hình chữ nhật không thể ảnh hưởng đến ComputationalGeometryApplication



## Defining a Responsibility

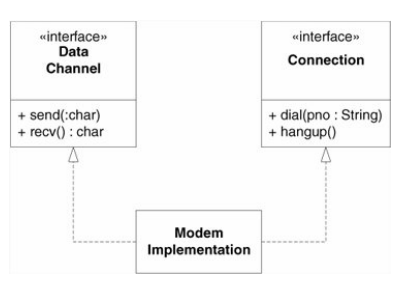
Trong bối cảnh của SRP, chúng tôi xác định trách nhiệm là một lý do để thay đổi. Nếu bạn có thể nghĩ về nhiều hơn một động lực để thay đổi một lớp, thì lớp đó có nhiều hơn một trách nhiệm. Điều này đôi khi rất khó nhìn thấy. Chúng tôi đã quen với việc nghĩ về trách nhiệm trong các nhóm. Ví dụ, hãy xem xét giao diện Modem trong Liệt kê 8-1. Hầu hết chúng ta sẽ đồng ý rằng giao diện này trông hoàn hảo

hợp lý. Bốn chức năng mà nó tuyên bố chắc chắn là các chức năng thuộc về modem.



Tuy nhiên, có hai trách nhiệm đang được hiển thị ở đây. Trách nhiệm đầu tiên là quản lý kết nối. Thứ hai là giao tiếp dữ liệu. Các chức năng Dial và Hangup quản lý kết nối của modem; các chức năng Send và Recv truyền dữ liệu.

Có nên tách rời hai trách nhiệm này? Điều đó phụ thuộc vào cách ứng dụng đang thay đổi. Nếu ứng dụng thay đổi theo cách ảnh hưởng đến chữ ký của các chức năng kết nối, thiết kế sẽ có mùi cứng nhắc, bởi vì các lớp gọi Send và Recv sẽ phải được biên dịch lại và triển khai thường xuyên hơn. Trong trường hợp đó, hai trách nhiệm nên được tách ra, như trong Hình 8-3. Điều này giữ cho client applications không ghép hai trách nhiệm



Mặt khác, nếu ứng dụng không thay đổi theo cách khiến hai trách nhiệm thay đổi vào các thời điểm khác nhau, thì không cần phải tách chúng ra. Thật vậy, tách chúng ra sẽ có mùi phức tạp không cần thiết.

Có một đường cong ở đây. Trục thay đổi là trục thay đổi chỉ khi thay đổi xảy ra. Sẽ không khôn ngoan khi áp dụng SRP theo bất kỳ nguyên tắc nào khác, vì vấn đề đó nếu không có triệu chứng.

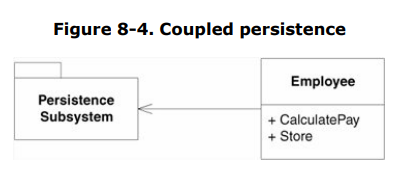
## Separating Coupled Responsibilities

Lưu ý rằng trong Hình 8-3, tôi giữ cả hai trách nhiệm được ghép nối trong lớp ModemImcellenceation. Điều này là không mong muốn, nhưng nó có thể là cần thiết. Thường có những lý do, liên quan đến các chi tiết của phần cứng hoặc hệ điều hành, buộc chúng ta phải ghép đôi những thứ mà chúng ta không muốn kết hợp. Tuy nhiên, bằng cách tách các giao diện của chúng, chúng tôi đã tách các khái niệm về phần còn lại của ứng dụng.

Chúng tôi có thể xem lớp ModemImcellenceation dưới dạng bùn hoặc mụn cóc; tuy nhiên, lưu ý rằng tất cả các dependency nên tránh từ nó. Không ai muốn phải phụ thuộc vào lớp này. Không ai ngoại trừ main để biết rằng nó tồn tại. Vì vậy, chúng tôi đã đặt một chút xấu xí đằng sau một hàng rào. Độ xấu của nó không cần phải rò rỉ ra ngoài và làm ô nhiễm phần còn lại của ứng dụng.

## Persistence

Hình 8-4 cho thấy một vi phạm phổ biến của SRP. Lớp Employee chứa các quy tắc kinh doanh và persistence control. Hai trách nhiệm này gần như không bao giờ được trộn lẫn. Quy tắc kinh doanh có xu hướng thay đổi thường xuyên và mặc dù persistence có thể không thay đổi thường xuyên, nhưng nó thay đổi vì những lý do hoàn toàn khác nhau. Liên kết các quy tắc kinh doanh với persistence subsystem có thể gây ra sự cố



May mắn thay, như chúng ta đã thấy trong Chương 4, thực tiễn phát triển dựa trên thử nghiệm thường sẽ buộc hai trách nhiệm này phải được tách ra từ lâu trước khi thiết kế bắt đầu có mùi. Tuy nhiên, nếu các thử nghiệm không buộc tách và nếu bắt đầu có mùi của độ cứng và sự dễ vỡ , thiết kế nên được tái cấu trúc, sử dụng các mẫu FACADE, DAO (Đối tượng truy cập dữ liệu) hoặc PROXY để phân tách hai trách nhiệm.

## Conclusion

Nguyên tắc đơn trách nhiệm là một trong những nguyên tắc đơn giản nhất nhưng là một trong những nguyên tắc khó nhất để có làm đúng. Các trách nhiệm liên kết là điều mà chúng tôi làm một cách tự nhiên. Tìm kiếm và phân tách các trách nhiệm đó là phần lớn những gì thiết kế phần mềm thực sự hướng tới. Thật vậy, phần còn lại của các nguyên tắc chúng ta thảo luận trở lại vấn đề này bằng cách này hay cách khác.

# Chapter 9. The Open/Closed Principle (OCP)

Cửa Hà Lan: Danh từ. Một cánh cửa được chia làm hai theo chiều ngang để một trong hai phần có thể được mở hoặc đóng.

Như Ivar Jacobson đã nói, "Tất cả các hệ thống thay đổi trong vòng đời của chúng. Điều này phải được ghi nhớ khi phát triển các hệ thống dự kiến sẽ tồn tại lâu hơn phiên bản đầu tiên." Làm thế nào chúng ta có thể tạo ra các thiết kế ổn định khi đối mặt với sự thay đổi và điều đó sẽ kéo dài hơn phiên bản đầu tiên? Bertrand Meyer đã cho chúng tôi hướng dẫn từ năm 1988 khi ông đưa ra open/closed principle nổi tiếng hiện nay. Để diễn giải anh ta:

The Open/Closed Principle (OCP)

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification

Khi một thay đổi duy nhất cho một chương trình dẫn đến một loạt các thay đổi cho các mô-đun phụ thuộc, thiết kế có mùi cứng nhắc. OCP khuyên chúng tôi nên cấu trúc lại hệ thống để những thay đổi tiếp theo của loại đó sẽ không gây ra nhiều sửa đổi. Nếu OCP được áp dụng tốt, những thay đổi tiếp theo sẽ đạt được bằng cách thêm mã mới, không phải bằng cách thay đổi mã cũ đã hoạt động. Điều này có vẻ giống như tình mẹ và quả táo vàng, lý tưởng không thể chối cãi trên thực tế, có một số chiến lược tương đối đơn giản và hiệu quả để tiếp cận lý tưởng đó

## Description of OCP

Các mô-đun phù hợp với OCP có hai thuộc tính chính.

- They are open for extension. Điều này có nghĩa là hành vi của mô-đun có thể được mở rộng. Khi các yêu cầu của ứng dụng thay đổi, chúng ta có thể mở rộng mô-đun với các hành vi mới thỏa mãn những thay đổi đó. Nói cách khác, chúng ta có thể thay đổi những gì mô-đun làm.

- They are closed for modification. Việc mở rộng hành vi của một mô-đun không dẫn đến thay đổi mã nguồn hoặc nhị phân của mô-đun. Phiên bản thực thi nhị phân của mô-đun trong một thư viện có thể liên kết, DLL hoặc tệp .EXE không được chạm tới.

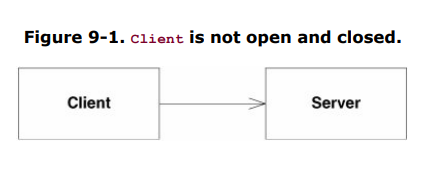
Có vẻ như hai thuộc tính này là bất hòa. Cách thông thường để mở rộng hành vi của một mô-đun là thay đổi mã nguồn của mô-đun đó. Một mô-đun không thể thay đổi thường được cho là có hành vi cố định.

Làm thế nào có thể các hành vi của một mô-đun có thể được sửa đổi mà không thay đổi mã nguồn của nó? Không thay đổi mô-đun, làm thế nào chúng ta có thể thay đổi những gì một mô-đun làm?

Câu trả lời là abstraction. Trong C # hoặc bất kỳ ngôn ngữ lập trình hướng đối tượng nào khác (OOPL có thể tạo ra các trừu tượng được cố định và đại diện cho một nhóm các hành vi có thể không bị ràng buộc.. Abstractions là các lớp cơ sở trừu tượng và nhóm các hành vi có thể không bị ràng buộc được đại diện bởi tất cả các lớp phát sinh có thể.

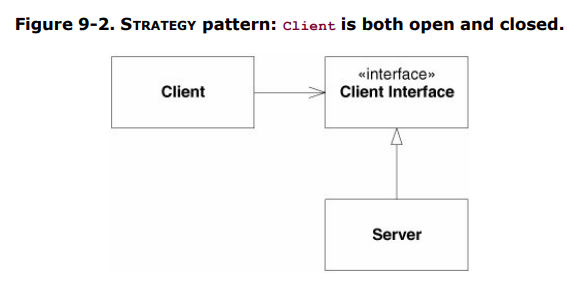
Có thể cho một mô-đun để dùng một sự trừu tượng. Một mô-đun như vậy có thể được đóng lại để sửa đổi, vì nó phụ thuộc vào sự trừu tượng được cố định. Tuy nhiên, hành vi của mô-đun đó có thể được mở rộng bằng cách tạo ra các dẫn xuất mới của abstraction.

Hình 9-1 cho thấy một thiết kế đơn giản không phù hợp với OCP. Cả hai lớp Client và Server đều cụ thể. Lớp Client sử dụng lớp Server. Nếu chúng ta muốn một đối tượng Client sử dụng một đối tượng máy chủ khác, thì lớp Client phải thay đổi tên lớp máy chủ mới.



Hình 9-2 cho thấy thiết kế tương ứng phù hợp với OCP bằng cách sử dụng mẫu STRATEGY (xem Chương 22). In this case, the ClientInterface class is abstract with abstract member functions.

The Client class uses this abstraction. However, objects of the Client class will be using objects of the derivative Server class. If we want Client objects to use a different server class, a new derivative of the ClientInterface class can be created. The Client class can remain unchanged.

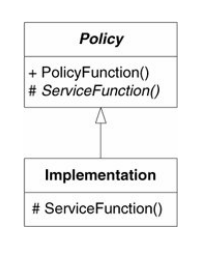


Client có một số công việc cần hoàn thành và có thể mô tả công việc đó theo giao diện trừu tượng được trình bày bởi ClientInterface. Các kiểu con của Client interface có thể implement interface đó theo bất kỳ cách nào họ chọn. Do đó, hành vi được chỉ định trong Serve có thể được mở rộng và sửa đổi bằng cách tạo các kiểu con mới của ClientInterface.

Bạn có thể tự hỏi tại sao tôi đặt tên ClientInterface. Tại sao tôi không gọi nó là AbstractServer thay thế? Lý do, như chúng ta sẽ thấy sau này, là các lớp trừu tượng được liên kết chặt chẽ hơn với các máy khách của chúng hơn là các lớp thực hiện chúng.

Hình 9-3 cho thấy một cấu trúc thay thế bằng cách sử dụng mẫu TEMPLATE METHOD (xem Chương 22). Lớp Policy có một tập hợp các hàm công khai cụ thể thực hiện một chính sách, tương tự như các hàm của Client trong Hình 9-2. Như trước đây, các policy function này mô tả một số công việc cần được thực hiện theo một số abstract interfaces. Tuy nhiên, trong trường hợp này, các abstract interfaces là một phần của chính lớp Policy. Trong C #, chúng sẽ là các phương thức trừu tượng. Các chức năng này được implement trong các kiểu con của Policy. Do đó, các hành vi được chỉ định trong Policy có thể được mở rộng hoặc sửa đổi bởi tạo các dẫn xuất mới của lớp Policy

Figure 9-3. TEMPLATE METHOD pattern: Base class is open and closed.



Hai mẫu này là những cách phổ biến nhất để đáp ứng OCP. Chúng thể hiện sự tách biệt rõ ràng của chức năng chung với việc thực hiện chi tiết chức năng đó.

## The Shape Application